RICE UNIVERSITY

Hardware-Software Co-Design for Optimizing MPI Programs in Data Center Network

By

Afsaneh Rahbar

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE

T. S. Eugene Ng
T. S. Eugene Ng (Dec 2, 2021 15:54 CST)

T.S. Eugene Ng
Professor of Computer Science and Electrical and Computer Engineering

[illegible]

Keith Cooper
L. John and Ann H. Doerr Professor in Computational Engineering, Professor of Computer Science

Moshe Vardi
Moshe Vardi (Dec 2, 2021 16:05 CST)

Moshe Vardi
University Professor, Karen Ostrum George Distinguished Service Professor in Computational Engineering

HOUSTON, TEXAS

December 2021

ABSTRACT

Hardware-Software Co-Design for Optimizing MPI Programs in Data Center Network

by

Afsaneh Rahbar

High Performance Computing (HPC) systems are critical. A single server/processor cannot handle the heavy computation needs of today's applications. HPC systems are built out of increasing numbers of processors to solve these computation-intensive problems. Communication between machines is essential. These applications may consist of thousands of processes spread across machines coordinating to solve a specific large-scale problem. The critical component of these systems is the network that connects the servers and makes this collaboration between servers possible. The performance of the network has a significant impact on the application performance. To better understand the main issues and improve the communication performance, in this thesis, we investigate data center networks and provide a general overview and analysis of the literature covering various research areas, including data center network architectures, network protocols for data center networks, and state-of-the-art communication frameworks.

We argue that many of the challenges faced by HPC applications in the communication phase can be addressed by augmenting the existing physical network architecture with low-cost optical technologies. However, we observe that integrating physical network/ hardware-based solutions alone would not be adoptable by HPC applications users. It requires some level of software-level application adaptations to the physical network before benefiting from the new characteristics of the network. Without a proper application to network interaction, the network cannot automatically adapt to the application's needs and vice versa. Our goal is to explore co-designing hardware and software solutions that optimize the data center network for MPI-based HPC programs. We propose a static source code analysis solution to identify the different communication patterns and requirements of applications and design algorithms that find the optimal network placement of the tasks to reduce the number of cross-rack communications to the least possible. We implement a prototype of our solution that automates learning the application communication characteristics, application to network interaction, and network to application adaptation (reconfiguring the network). We evaluate our tool and demonstrate the high potential of hardware-software co-design for optimizing HPC programs in the data center network.

Acknowledgments

I would like to express my heartfelt gratitude to the people without whom I could not have written this dissertation. First and foremost, I am grateful to my advisor Eugene Ng for his support, guidance and encouragement. Thank you, Eugene, for believing in me and giving me the freedom to explore, all the while challenging me and guiding me. I hope that one day I can be as effective an advisor as you have been to me. My sincere thanks to Moshe Vardi and Keith Cooper for their valuable insights that helped consolidate my research focus and for serving on my PhD defense committee with valor. Thank you Moshe, for welcoming me into your research group and being present for me from the first day I joined Rice. Keith and Linda Torczon's "Engineering a Compiler" book inspired me to come to Rice. I would also like to thank Philip Taffet and John Mellor-Crummey for their constructive feedback on my research from the very beginning.

I am grateful to Luay Nakhleh for his unflinching support. His sense of fairness and his support of PhD students helped me get through the worst stretch here. In the same spirit, Devika Subramanian stood by me and got me over my self-doubts. I should also thank my collaborators Ang Chen, Xiaoye Sun, Dingming Wu, Sushovan Das, Weitao Wang and all other members of BOLD lab (Simbarashe J. Dzinamarira, Xin Huang, Jiarong Xing, Kuo-Feng Hsu, Zhuang Wang, Xinyu Wu, and Richard Qiu). I would be remiss if I didn't recognize the valuable support and camaraderie of my fellow researchers: Dror Fried, Suguman Bansal, Kuldeep S. Meel, Aditya Shrotri, Eleni Litsa, Jeffery Dudek, Lucas Tabajara, Vu Phan, Kurt Warren, Zhiwei Zhang, Jayvee Abella, Arkabandhu Chowdhury, Ryan Anthony Elworth, and Ken Odegard. Their sharp minds, wit, and sense of humor kept me going. I will always smile when I think back to our coffee breaks, time at Gibbs Recreation Center, and Valhalla (not to forget the beer bike events). Thank you for your help and support.

Special thanks to my friend and mentor Matthias Felleisen, and deep gratitude to my friends from outside of the research community who stayed by my side and supported me throughout this long and challenging journey. Here I will name a few of many: Xavier

Briswalter, Vincent Brochart, Vesta Broumand, Nima Fard, Elaheh Thompson, Caroline Capelle, Alexandr Sarioglo, Nima Radan, Geofrey Prioreau, and Charline Primat.

Finally, to my parents, I owe my biggest debt of gratitude. Their love and continued support has been my source of strength. I dedicate this thesis to them.

Contents

Illustrations

Tables

Chapter 1

Introduction

High-Performance Computing (HPC) systems play an essential role in today's heavily digitized world. HPC systems benefit from parallel computing systems that decompose problems into sub-problems. These sub-problems run on different computational units/processors to solve computation-intensive scientific problems in a reasonable amount of time. The computational units need to collaborate to complete a specific task. If all the computational units reside in the same machine, processes collaborate using the shared memory space. Otherwise, for applications on distributed-memory parallel systems, cross-machine communication is essential. Such applications often communicate using a standard library for message passing defined by the MPI (Message Passing Interface) since the number of processors on one machine is not enough to solve these computation-intensive problems. Therefore the latter case is more commonly seen. These MPI-based HPC programs may consist of thousands of processes spread across machines coordinating to solve a specific large-scale problem.

For instance, consider the distributed machine learning (ML) workloads, including the Logistic Regression algorithm for Twitter spam filtering [1] and the Alternating Least Squares algorithm for Netflix movie rating prediction [2, 3]. They manipulate and analyze massive amounts of data and run on compute clusters and data centers consisting of tens of thousands of machines. Both jobs take hundreds of iterations, and communications account for 30% and 45% of the job completion time, respectively. Consider also data mining workloads (e.g., Apache Hive [4], Spark SQL [5]). In such workloads, one of the most critical and time-consuming operations is the distributed database join, in which one of the input tables is multicast to all workers. These tables are up to 6.2 GB in a popular database benchmark [6]. Another example would be distributed matrix multiplication which is a fundamental linear algebra routine ubiquitous in all areas of science and engineering.

These popular applications run on compute clusters in the data center. The data center is a pool of resources (computational, storage, network) interconnected using a communication network. Data Center Network (DCN) interconnects all data center resources (computational, storage, network) together. Datacenter networks are critical infrastructure behind today's cloud services which accommodate diverse applications [7] including HPC applications. Data and computation-intensive HPC applications are on the rise in DCNs, and they constantly transmit large objects across servers.

The communication characteristic of these HPC applications is as follows:

1. Performing point-to-point communications.
2. Performing group communications.
3. Dynamically changing the communication pattern/ communicating processors in different stages of the program.

The performance of the network connecting these resources has a significant impact on overall application performance. The sensitivity to communication performance increases with the scale of parallelism. In large-scale applications, communication has always been one of the main bottlenecks. Several studies have shown that applications spend a substantial fraction of time moving data between machines rather than performing computations. Typically these applications spend at least 20% of their execution time communicating, and some spend more than 50%. Therefore, the network is becoming the main bottleneck for scaling parallel application [8, 9, 10, 11].

Optimizing network activity is critical for improving large-scale applications performance. The problem to be addressed is how to improve and optimize the communication performance of these applications over data center networks?

From this point forward, we use large-scale HPC applications interchangeably with MPI-based applications. Researchers have focused on optimizing or redesigning the communication algorithms used by MPI to optimize the performance of MPI-based applications [12, 13, 14, 15]. It is recommended to pick different communication algorithms depending on the message size to minimize latency for short messages and minimize bandwidth use for long messages. These efforts and tunings are all at the software level and not in the actual network.

We argue that many of the challenges HPC applications face in the communication phase can be addressed by augmenting the existing physical network architecture with low-cost optical technologies.

We propose co-designing a hardware and software-based solution for optimizing the data center network to the needs of MPI-based applications. The network topology and architecture design impact communication performance, but the network is fixed once installed. A typical data center network topology seen in today's data centers is the traditional multilayer oversubscribed Clos topology [16, 17] which is a tree-based fixed topology. Minor modifications can be done in the data center network occasionally without a hefty cost and downtime; however, without a proper application to network interaction, the network cannot automatically adapt to the application's needs and vice versa. Without an automated adaptation of the application to the optimized network, the applications cannot benefit from the optimized network hardware solution.

1.1 Thesis Contributions

This thesis investigates data center networks and provides a general overview and analysis of the literature covering various research areas, including data center network architectures, network protocols for data center networks, and state-of-the-art communication frameworks.

We present the design and implementation of multiple systems supporting the communication needs of distributed parallel programs (point-to-point and group communications). The common goal behind these systems is to improve HPC application performance while retaining as many of the existing benefits of current data center networks as possible and without significantly changing the network and increasing the hardware cost. We propose a broad scale of hybrid hardware-software solutions to optimize communication performance. We attack the problem from various angles.

In particular, this thesis makes the following contributions.

- First, we evaluate a network augmentation solution called Shufflecast. Shufflecast augments the existing Clos network with a separate network dedicated to multicast transmissions. This architecture leverages inexpensive passive optical splitters and cleverly makes use of edge network bandwidth. We show that by separating multicast traffic from the Clos network to Shufflecast, we can achieve line-rate multicast throughput and low latency. We implement communication routines that are compatible with Shufflecast and adapt the distributed parallel applications. We perform a comprehensive evaluation to observe the benefit of having such solutions to MPI applications.

- Second, we evaluate a network augmentation solution called RDC—Rackless Data Center. The goal of RDC is to remove the bandwidth disparity between intra-rack and cross-rack communications. This architecture logically removes the rack boundary of traditional data centers and the inefficiencies that come with it. RDC achieves this by inserting circuit switches at the network edge between the ToR—Top of Rack switches and the servers and reconfiguring the circuits to regroup servers across racks based on the traffic patterns minimizing cross-rack traffic. Rather than optimizing the MPI applications based on the topology, RDC optimizes the topology to suit the changing workloads of MPI applications.

  We have performed extensive evaluations of RDC in a hardware testbed. RDC can speed up a 4:1 oversubscribed network to achieve nearly non-blocking network performance. Note that to improve the communication performance of an MPI application executing on RDC, users need to understand how the application's communication patterns interact with the network, primarily when those interactions result in congestion and instruct the application with the reconfiguration code.

- Our third contribution is designing a static analysis-based communication pattern detection and optimal server placement identification solution for MPI applications. Automated approaches are needed since understanding application communication patterns and their interaction with the network is complex for users. Our prototype uses the optimal server placement to augment the application source code with the reconfiguration code compatible with RDC. We test the resulting program on the RDC testbed. Our evaluation shows that the optimal server placement automatically identified by our solution can speedup a 4:1 oversubscribed network to achieve nearly non-blocking network performance similar to the manual RDC.

1.2 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 analyzes existing ways to support multicast group communication in DCNs and demonstrates their limitations. Chapter 3 presents the first solution to improve MPI program group communication by using network-level multicast and segregating multicast traffic from unicast traffic. Chapter 4 presents the second solution to improve MPI program point-to-point and group communications using a reconfigurable rackless data center architecture to move servers to the same logical rack and minimize cross-rack communications. Note that this solution relies on application-level multicast. Chapter 5 presents a static analysis-based solution to detect MPI program communication patterns and oscillations in different applications. Furthermore, our solution finds the optimal server placement for the DCN. This solution leads to automating the network reconfiguration for RDC and similar reconfigurable networks. This chapter discusses the solution, implementation, results, and evaluation. Finally, Chapter 6 concludes with a summary and a discussion on possible future directions.

Chapter 2

Analyzing Existing Ways to Support Multicast

Data and computation-intensive applications such as HPC applications are on the rise in Data Center Networks (DCN), and they constantly transmit large objects across servers which leads to more and more traffic that is multicast "in disguise" [18]. Many scientific data analysis jobs [19, 20, 21] perform iterative multicasts using MPI_Bcast [22], which is a primitive in the MPI framework for one-to-many message passing.

2.1 Group communication issues over DCN

Today's multicast service models are either extremely network-centric or application-centric. In the former, the network switches directly support IP-level multicast, and in the latter, applications form a peer-to-peer overlay to spread data via unicast. We analyze both models in the following sections.

2.1.1 Network-centric service model

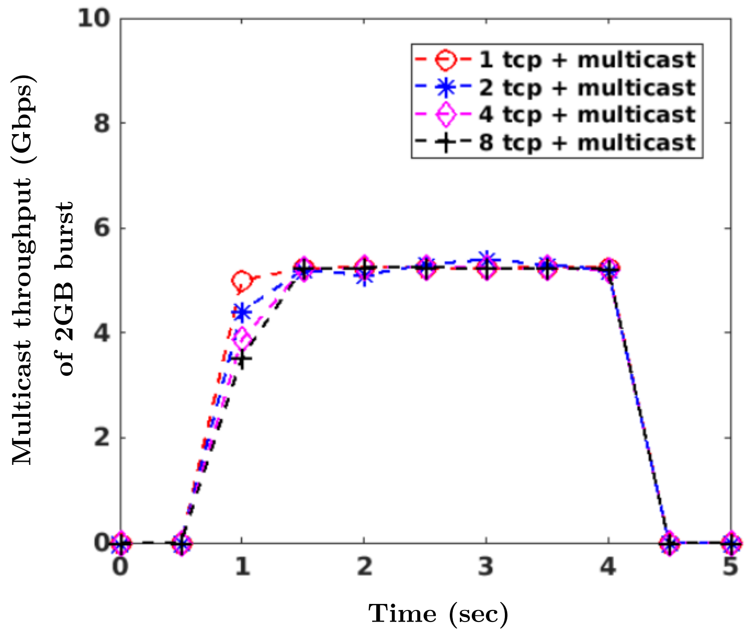
The network-centric service model enables multicast packet forwarding on switches. In practice, this means switch-level multicast over a conventional hierarchical Clos network, which we will refer to as IP multicast in the rest of this thesis. This model ensures efficient use of link bandwidth as there is no data redundancy. Although some recent works have focused on the switch-level improvement of IP multicast [23, 24, 25, 26], the network-centric service model is still unable to satisfy several attributes desirable for both applications and network operators, as shown in Table 2.1. Primarily this service model lacks an easy way to achieve reliability, predictability and fairness for the inter-mingled multicast and unicast traffic. On the one hand, we observe that even a small amount of switch-level multicast traffic without congestion control can cause a significant and

Attributes \ Service model	**Net.-centric**	**App.-centric**
Predictable finish time	N	N
Easy to achieve reliability	N	Y
Friendliness with TCP traffic	N	Y
Efficient use of link bandwidth	Y	N
Scale to large group size	Y	N
Support multiple groups	Y	Y

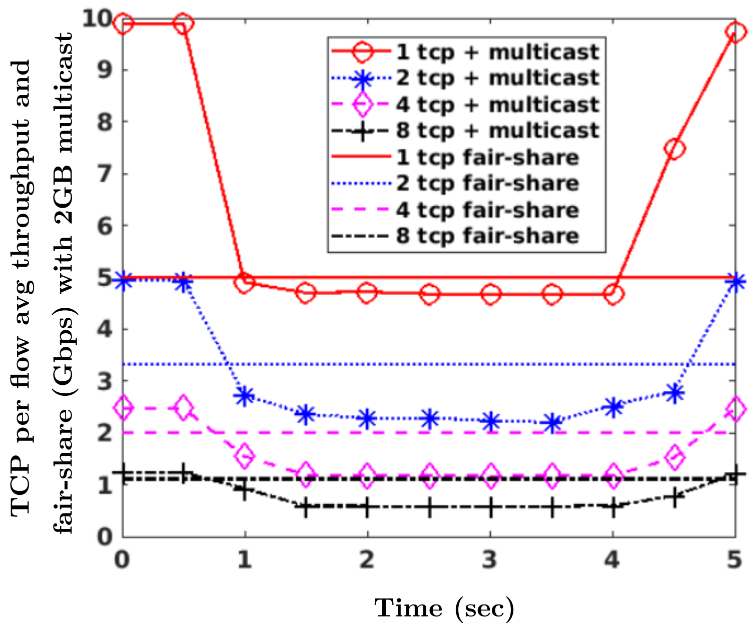
Table 2.1 : Comparison between the service models.

disproportional negative impact on co-existing unicast traffic flows and congestive packet loss of the multicast flow. On the other hand, we also observe that enabling multicast congestion control does not guarantee fairness among multicast and other existing unicast flows.

To observe the effect of multicast congestion control on unicast traffic, we perform experiment on a 10 Gbps full-bisection bandwidth cluster. The servers are equipped with NACK-oriented Reliable Multicast (NORM) protocol [27] which uses the TCPFriendly Multicast Congestion Control (TFMCC) scheme [28, 29]. We configure multicast rules on the switches and perform a 1 : 15 multicast of a 2 GB file in the presence of steady background unicast (TCP) traffic flows sharing a common core-to-ToR switch downlink. We vary the number of competing unicast (TCP) flows from 1 to 8 and observe both the multicast throughput (Figure 2.1(a)) and individual TCP per flow average throughput along with the theoretical fair-share (Figure 2.1(b)). We observe that no matter how many competing TCP flows are there, the multicast flow achieves 5 - 5.2 Gbps of throughput at a steady state and the competing TCP flows get the fair-share from the remaining bandwidth. Clearly, with more competing flows, multicast congestion control scheme cannot achieve fairness guarantee. Hence, using the network-centric service model is very hard to find the right balance between multicast and unicast traffic.



(a)



(b)

Figure 2.1 : (a) Throughput of congestion control enabled multicast in the presence of competing TCP flows, (b) Per flow TCP average throughput and theoretical fair-share in the presence of 2 GB congestion control enabled multicast.

2.1.2 Application-centric service model

Today's DCN does not support IP-level multicast/Network level multicast. Therefore, the only way of doing group communication is application-based, where applications spread data on their own via repeated unicast transmissions, there are significant overheads. The application organizes its processes into an overlay network and sends multicast messages based on unicast traffic flows [30, 31, 32, 33, 26]. Although this model is easy to deploy, the network and the CPU are stressed out since identical packets are pushed multiple times to the network, and packet replication is performed on end-hosts instead of the network [34]. For instance, variants of BitTorrent are commonly used (e.g. [35, 3]). Unfortunately, even when very carefully optimized by experts, data redundancy is still at 39% [34]. This model also suffers from latency inflation and unpredictability under a large multicast group size [36] because fluctuations in relay server performance can cause a collapse in throughput [36]. Moreover, overlays contribute to energy waste [37] because data must be copied in and out of relay servers' memory, and server CPU cycles are consumed to route data and maintain overlays. Hence, the application-centric service model lacks *efficient utilization of link bandwidth* and predictable performance with large multicast group size, desirable features from network operators and applications, respectively.

In Section 2.1.2.1, we illustrate some of the underlying problems of the application-centric model. In Section 2.1.2.2, we run experiments to evaluate the state-of-the-art multicast performance. In Section 2.1.2.3, we run experiments to demonstrate how much multicast performance affects the overall runtime of a variety of real-world applications.

2.1.2.1 BitTorrent over 4-ary fat-tree topology NS3 Simulation

BitTorrent and its variants are commonly used systems for file distribution and application-level multicast. We simulate a 4-ary fat-tree topology with the Network Simulator 3 (ns3) using a BitTorrent extension [38] to help illustrate some of the underlying problems of the application-centric model. Each host in the topology plays a role as a peer in the Bit-Torrent overlay simulation. In BitTorrent, the data is broken down into a large number of equal-sized pieces that are downloaded in rarest-first order. The creator of a torrent determines the size of a piece. The pieces are further split into a set of a few blocks to avoid downloading large chunks from slow or overloaded peers. Peers not only download blocks from others but also serve to other peers. Each peer can start serving other peers as soon as they have downloaded a block.

We assume that one of the hosts initiates the multicast and all the other 15 hosts start requesting the data simultaneously. We measure the download rate of BitTorrent overlay when there is no other traffic in the network while varying link bandwidths (1 Gbps, 10 Gbps), piece sizes (64 MB, 128 MB, 256 MB), and block sizes (512 KB, 1 MB, 16 MB, 32 MB, 64 MB, 128 MB, 256 MB). The data size exchanged among the peers is 4.03 GB.

We observe: 1) the overall performance of BitTorrent is not very good compared to the network bandwidth. For example, the download rate does not exceed 1 Gbps when the link bandwidth is 10 Gbps. 2) It is hard to tune the parameters to obtain the best performance of BitTorrent. Potential reasons behind the significant difference in BitTorrent download rate vs. the available bandwidth are a) Peer finding process, b) Duplicate blocks in the network, c) TCP congestion control mechanism. For each block shared by one peer with any other peer, the TCP transmission suffers from a slow start and may not reach the max congestion window. On one hand, we need bigger blocks to reach the max congestion window. But on the other hand, small block sizes help in efficient sharing. Finding the right combination of piece size, block size for a specific link bandwidth to satisfy such conflicting needs is still an open problem.

2.1.2.2 Multicast performance of state-of-the-art multicast mechanisms

We perform benchmarking experiments to evaluate the state-of-the-art multicast performance. Our baseline mechanisms are state-of-the-art multicast solutions over a full-bisection bandwidth network: a) IP multicast, and b) peer-to-peer mechanisms such as MPI_Bcast [22] and Spark-Cornet [3]. As stated in Section 2.1.2, today's DCN does not support IP-level multicast/Network level multicast; therefore, we manually calculate the IP multicast throughput. However, we run the peer-to-peer mechanisms on our testbed (traditional multi-layer Clos topology). The testbed consists of 16 servers and 4 ToR switches in 4 logical racks as well as one aggregation switch. Each server has 6 3.5 GHz CPU cores with 12 hyperthreads and 128 GB RAM. All connections are 10 Gbps Ethernet.

We perform a 1 : 15 multicast with varying data size (from 200 MB to 1.4 GB) and measure the multicast reading time (i.e., the duration between receiving program issues reading request and finishing reading it). Figure 2.2 shows the application-level multicast throughput defined as the ratio of multicast data size to multicast reading time.

Without any competing traffic, IP multicast (assuming that IP multicast is using a full-bisection bandwidth network) achieves close to line-rate performance irrespective of multicast data size. However, in presence of competing unicast traffic, IP multicast (over full-bisection bandwidth network) without congestion control leads to unreliable and unpredictable performance (recall Section 2.1.1); similarly, IP multicast (over full-bisection bandwidth network) with congestion control (e.g., NORM [27]) fails to achieve fairness with unicast flows (recall Section 2.1.1).

Even without any competing traffic, we observe that both MPI_Bcast and Spark-Cornet achieve application-level throughput only upto 35% and 20% of the line-rate throughput across data size, which is far from optimal. In Spark-Cornet, a node first locates a block of data it needs from another node then performs a block transfer. We observe that although each individual block transfer can reach near line-rate throughput, far more time is taken up by control communications to locate and wait for data blocks, which becomes the bottleneck for overall throughput. MPI_Bcast, adopts a different approach, where the data is pipelined from one node to the next. In this case, the software handling of data from input to output across the pipeline and the need to ensure reliability across the pipeline becomes the bottleneck for overall throughput.

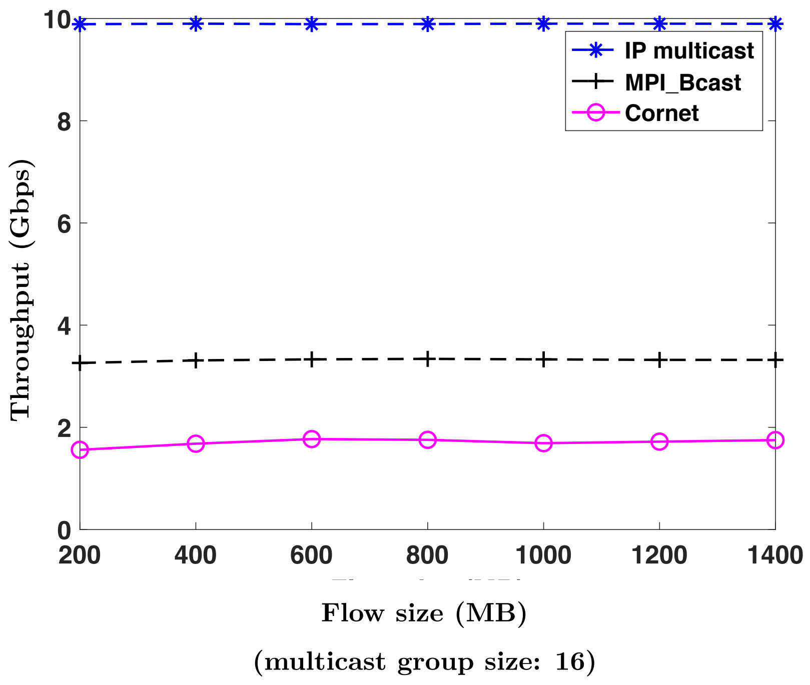


Figure 2.2 : Application-level multicast throughput vs. data size for a 1 : 15 multicast flow.

2.1.2.3 Real-world applications

A variety of applications rely on multicast and point-to-point communications, but their needs are different. Some applications are throughput sensitive, and some are latency sensitive. We study throughput sensitive applications and their characteristics in Section 2.1.2.3.1 and latency sensitive applications and their characteristics in Section 2.1.2.3.2.

2.1.2.3.1 Throughput sensitive

Most big-data applications rely on multicast communication. We briefly discuss the workloads and experimental results of these applications.

Spark ML: Under Spark Machine Learning applications, we focus on two popular iterative machine learning algorithms: Latent Dirichlet Allocation (LDA) and neural word embedding. We use the Spark LDA implementation [39] with the dataset of 20 Newsgroups as the input corpus [40], and do the one-to-all multicast for the training

Application	Multicast Mechanism	Multicast Reading Time (s)	Application Running Time (s)
LDA	Cornet	34.30	105.13
	HTTP	65.74	141.69
Word2Vec	Cornet	25.90	984.00
	HTTP	58.49	1022.10

Table 2.2 : LDA and Word2Vec: Cornet and HTTP multicast reading time inefficiency and their impact on overall application runtime.

vocabulary model (735 MB in size). In addition, we use the Word2Vec [41] neural word embedding application in Spark MLlib, with a training model size of 504 MB. Currently, both of these applications use Spark's native multicast mechanisms like Cornet [3] and HTTP (repeated unicasts to all receivers).

For these applications, we use 8 servers. The application randomly chooses one server with four cores and 88 GB RAM as the master, while the other seven servers with two cores and 44 GB RAM serve as 14 slave executors. Both the Spark ML applications (i.e., LDA and Word2Vec) run for 10 iterations. Finally, we obtain the total multicast reading time and application running time shown in Table 2.2.

For LDA, in an ideal world, IP multicast could speedup multicast reading time by 4× and 7× compared to Cornet and HTTP, respectively. Similarly, for Word2Vec, IP multicast could speedup multicast reading time by 3× and 6× compared to Cornet and HTTP, respectively.

Word2Vec is more computation-intensive, so the multicast reading time plays a smaller role in overall application runtime than LDA.

Spark distributed database: TPC-H is a widely used database benchmark of 22 business-oriented queries with high complexity and concurrent data modifications [6]. We run these queries using the Spark SQL framework [5]. The database tables are 16 GB in size overall, and the multicast data is one of such tables with sizes ranging from 4 MB to 6.2 GB for the distributed database join, making a total of 48.3 GB of multicast data

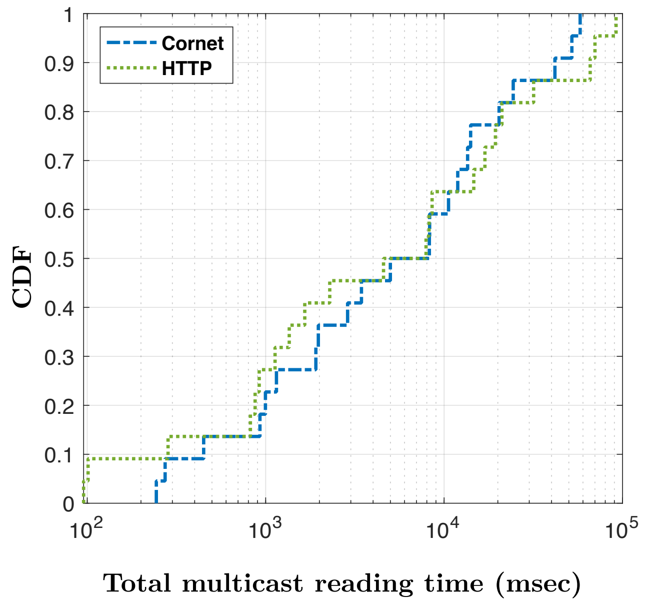


Figure 2.3 : CDF of TPC-H total multicast reading time.

across queries.

We run Spark distributed database through the traditional two-layer fat-tree network (full-bisection bandwidth network). Figure 2.3 shows the multicast reading time of each TPC-H query (queries 1 to 22) averaged over 10 runs. For certain queries (e.g., 1, 4, 6, 14, 15, 22), the amount of multicast data is either very small (under 200 MB) or non-existent. However, for other queries (e.g., 9, 17, 18), multicast data is large (5 GB), so with IP multicast we could get a speedup of 2.7× and 3.5× in multicast reading time compared to Cornet and HTTP respectively.

DMM: Distributed Matrix Multiplication (DMM) is one of the most important linear algebra kernels, and it is widely used in diverse applications, such as machine learning [42, 43, 44, 45], deep neural networks training [46], fluid dynamics [19], climate modelling [20], and molecular dynamics simulation [21]. We use Fox [47] as an example DMM algorithm where processes need to multicast chunks of one matrix in different iterations. DMM is a difficult workload in general, as it involves multiple processes to multicast the matrix chunks in different iterations. Current MPI-based Fox implementation [48] uses MPI_Bcast as the multicast mechanism.

The Fox algorithm considers p servers to be logically mapped to a $\sqrt{p} \times \sqrt{p}$ space on a single plane containing equal-sized blocks of matrices A and B. It then follows a “multicast-multiply-roll” cycle, where the servers a) multicast the blocks of matrix A row-wise, b) multiply with the available blocks of B and finally, c) roll/shift the blocks of matrix B column-wise. As each server multicasts A’s block only once throughout the execution, the network will carry one copy of A matrix. We have used a Fox implementation using the Open MPI library [49] (v1.6.5), which uses a chain-based communication algorithm for multicast.

We benchmarked with different numbers of machines (2/4/8) and processes (16/64/256) on varying matrix sizes (10k×10k-70k×70k) in 4 different scenarios. Each scenario has a different oversubscription ratio created by injecting competing traffic from iPerf3 [50]. The competing traffic leaves different amounts of available bandwidths for the unicast network: 10 Gbps (line-rate), 5 Gbps, 2.5 Gbps, and 1.25 Gbps, respectively.

We compare the basic version of MPI multicast in the 4 scenarios using 8 machines and 64 processes – this combination of machines/processes setting is found to perform the best.

The process to server mapping is done in a way that only multicast traffic is sent outside of the rack, and all unicast traffic is contained within a rack, so only multicast traffic is impacted by oversubscription. For MPI, the multicast rate decreases with heavier congestion. We ran each combination of matrix sizes, scenarios, and multicast mechanisms 10 times and obtained their average running time. Figure 2.4 shows the average multicast reading time over the average total application running time. Here, the multicast reading time is defined as the duration from (1) the time when the program issues the block reading request running in the receiving process to (2) the time when the program finishes reading the block. The total application running time consists of multicast time, shift time, and computation time.

In Figure 2.4 we demonstrate that if in an ideal world we had IP multicast without the disadvantages stated in Section 2.1.1 multicast reading time would improve considerably.

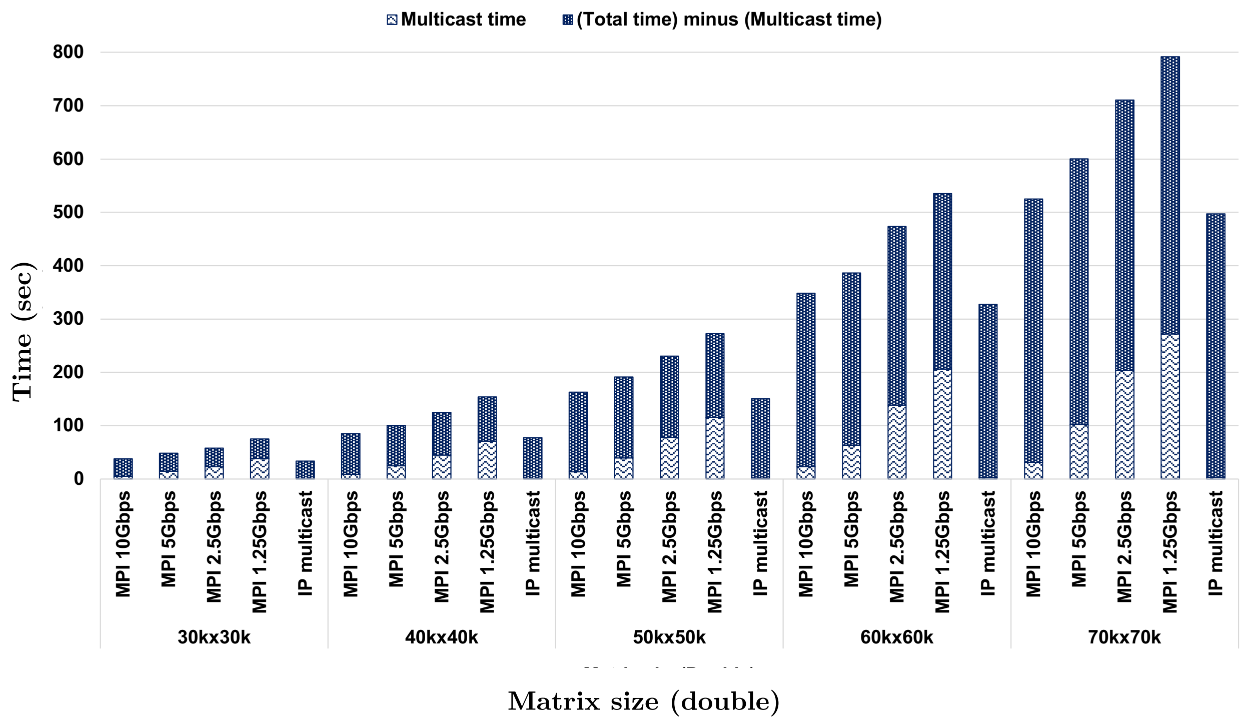


Figure 2.4 : The average multicast time and application running time of MPI and IP multicast in all the four scenarios for matrix sizes 30k×30k to 70k×70k.

For the matrix size 70k×70k, IP multicast theoritically achieves speedups of 8.6×, 28.1×, 56.02×, and 74.8×, compared to MPI with 10 Gbps, 5 Gbps, 2.5 Gbps, and 1.25 Gbps, respectively. Figure 2.4 further shows that this translates to application-level improvement, by 5.3% (10 Gbps), 17.2% (5 Gbps), 30.04% (2.5 Gbps), and 37.21% (1.25 Gbps), respectively.

Even non high throughput applications will suffer from application level multicast, which we will demonstrate in Section 2.1.2.3.2.

2.1.2.3.2 Latency sensitive

Paxos [51, 52, 53] is a consensus protocol for distributed set of unreliable processes. It is used for implementing a fault-tolerant distributed system. The consensus algorithm ensures that a single value is chosen among the proposed values by all the processes, and participants agree on it. It assumes a leader-election oracle. At a high level, Paxos

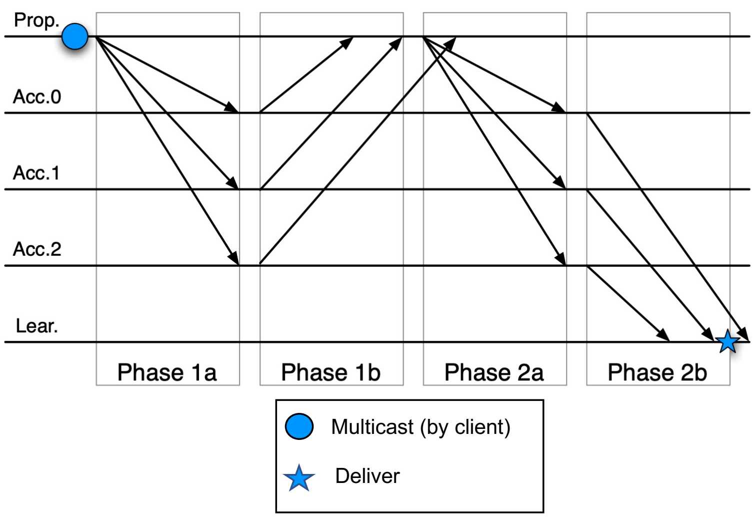


Figure 2.5 : The phases of the PAXOS algorithm.

distinguishes the processes as proposers, acceptors, and learners. First, the proposer multicasts a message with a unique number to all acceptors. If the acceptors have not seen a message with a higher unique number, they confirm the reservation of the ballot and send back a message to the proposer. If the proposer receives confirmations from at least more than half of the acceptors, it will multicast a message including the value to be proposed and the ballot number to all acceptors. Finally, the acceptors store the value and multicast their decision to the proposer and all the learners. Three out of four of the steps above involve multicast. Figure 2.5 shows the sequence of events required in each instance of consensus in order to deliver a value.

Various implementations of Paxos exist, some of them are based on IP multicast (RingPaxos[54], LibPaxos v1 and v2[55], etc.) and some are based on Unicast (Paxos4sb [56], Libpaxos v3 [55]).

Figure 2.6 shows the latency of Unicast based Paxos application level multicast through the traditional two-layer fat-tree network. As the messages tend to be small, the performance of Paxos is sensitive to latency. The proposers send a couple of hundred values of size 1 Byte, one at a time to all acceptors. Figure 2.6 shows the 90th percentile latency of hundreds of values proposed. The latency of Unicast based Paxos scales linearly with the number of acceptors (receivers), while Multicast based Paxos's

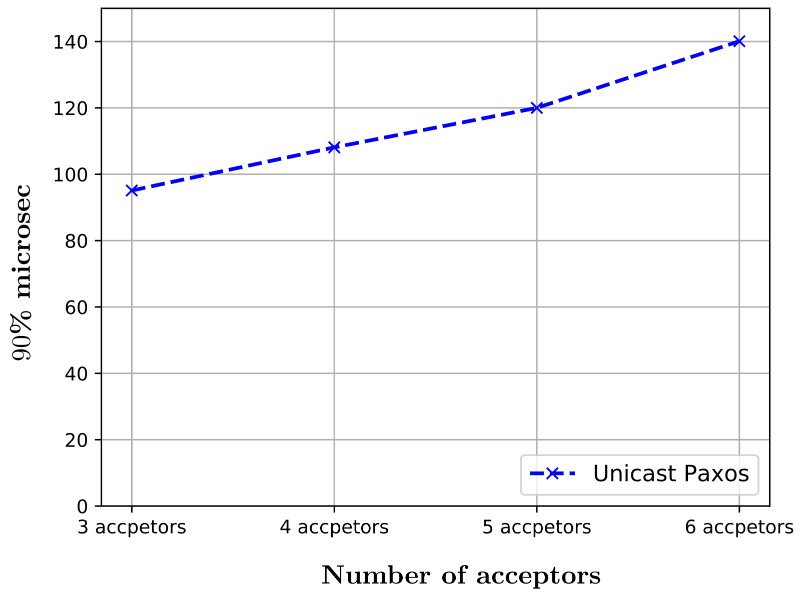


Figure 2.6 : The 90th percentile latency of unicast Paxos over fat-tree with one sender. The latency of Unicast Paxos scales linearly with the number of acceptors (receivers).

latency will be almost constant and minimal due to the use of IP multicast for message dissemination.

2.2 Summary

Applications in compute clusters heavily rely on multicast group communication. Multicast service models today could be characterized as "leave it to the network" or "leave it to the application". Unfortunately, neither model achieves simultaneously the predictability and reliability that applications need, and the efficiency, unicast-friendliness and scalability that network operators want.

Chapter 3

Evaluate the benefits of augmenting DCN with a parallel IP multicast enabled network

In Chapter 2 we saw that both network-centric and application-centric service models could support multicast transmissions to some degree, but neither provides an ideal solution. We demonstrated how application-level multicast could hurt the overall runtime of applications relying on multicast group communications. Continual reliance on application-level overlays for multicast data transmission is not a tenable position in the long run. We also saw that network-level multicast could achieve high throughput and low latency, but it does not come for free and has some disadvantages. Thus it is not supported in today's DCNs.

To take advantage of the high throughput and low latency of network-level multicast, one solution is to reserve multicast trees from the existing Clos-based network; however, this is very costly, as the multicast trees consume a lot of resources in the network core. Consider a one-to-all multicast; we need a cluster-wide multicast tree, which would block a core switch entirely and almost all downlinks of one aggregate switch in each pod. Even in cases where multicast trees are not cluster-wide, the scenario will worsen due to the rise of multicast applications with more frequent arrivals and constraints in job placement within busy clusters. It could become necessary to reserve more than one cluster-wide multicast tree with a proportionally higher core and aggregate resources. Hence, involving the core for multicast resource consumption would not be pragmatic.

[57], proposes an edge-network architecture called Shufflecast, which is responsible for multicast. It restricts the resource consumption to the network edge, which would be more sustainable in the long run. Shufflecast coexists with the unicast network, and multicast and unicast traffic are physically isolated. Separating the communication

classes in this manner enables us to target multicast with the most appropriate network technology and operating mechanism. The key features are a) multicast and unicast traffic are physically segregated; b) the network is responsible for multicast transmission; and c) its architecture supports high scalability in multicast group size and number of groups and achieves high reliability in the presence of multiple multicast groups.

Adopting a parallel multicast dedicated network such as Shufflecast should give us the benefits of IP-level multicast/Network level multicast without harming the unicast traffic. We expect having the ability to do IP multicast would benefit the overall runtime as shown in Chapter 2 Section 2.1.2.3.

For long-lived bulk data transfers, we build a complete hardware and software prototype of Shufflecast (Section 3.2) and perform comprehensive testbed evaluation (Section 3.3). We want to see how much benefit an auxiliary dedicated multicast network can bring to the overall runtime. A tutorial of Shufflecast is given in Section 3.1.

3.1 Shufflecast

3.1.1 Introduction

Shufflecast is a novel optical architecture that supports high-performance multicast in compute clusters or DCNs. Shufflecast segregates multicast and unicast traffic and has a network that has explicit multicast support. The architecture supports high scalability in terms of multicast group size and also reliably supports multiple multicast groups. As we saw earlier allocating multicast trees from the existing hierarchical Clos-based network is very costly as it will consume a lot of resources inside the network core. With multicast-heavy jobs arriving more frequently and busy clusters having less job placement flexibility, the problem will worsen. Instead, Shufflecast restricts multicast resource consumption to the network edge to have minimal impact on unicast traffic.

Shufflecast is a scalable edge-network topology and data plane design that directly connects the ToR switches using inexpensive passive optical splitters, providing physical-layer enabled high-performance multicast without disrupting other unicast traffic. This proposed design supports intra-pod and inter-pod multicast with latency growing sub-linearly while achieving highly desirable properties: no extra rack space, minimal extra power, low cost, high performance, and low latency at scale. This well-defined topology also provides static optimal ToR-level routing. Shufflecast has a highly responsive, lightweight control plane design that enables simple customization of ToR-to-server forwarding based on an application-defined multicast group.

3.1.2 Potential benefits of edge-based design

First, multicast and unicast traffic do not commingle with each other, leading to more predictability, reliability, and fairness guarantee for both types of traffic with much less complexity. Second, the network is responsible for multicast, eliminating application-level data redundancy, and ensuring efficient use of link bandwidths (Section 3.3.1). Third, Shufflecast's data plane design can achieve high scalability in supporting large multicast group sizes. Fourth, only edge switches carry multicast traffic, making the Shufflecast control plane simple while reliably supporting multiple multicast groups. Finally, edge resource consumption will not grow as fast as the core, as future multicast traffic demands may increase.

3.1.3 Network Architecture

In this section, we first describe the key building blocks and then describe the Shufflecast architecture.

3.1.3.1 Building Blocks

In this architecture, every ToR switch is equipped with (i) passive optical splitter(s) and (ii) fixed-wavelength optical transceivers, and it is connected to some other ToR switches via (ii) single-mode optical fibers. We describe the key features of these optical devices below.

Optical Splitters: An optical splitter is a unidirectional passive device with one input and multiple output ports, typically made of multiple fibers twisted and fused together. Splitters can split the incoming optical signal of any wavelength from one input port to multiple output ports by proportionally dividing the signal power. For example, a $1:2$ splitter can evenly split the incoming signal into two output ports, incurring $10\log_{10} 2 = 3$ dB of insertion loss in the ideal case. In the case of planar waveguide circuit (PLC) splitters, there may be an additional $1-1.8$ dB of extra loss [58]. Moreover, being a passive device, it does not consume any active power and works for any data rate. This built-in support for multicast capability, along with its operational simplicity and low cost, makes the optical splitter commercially suitable for Shufflecast's network design.

Optical Transceivers: Optical transceivers contain both the optical transmitter (LED or laser: converting electrical to optical signal with specific wavelength) and receiver (photodetector: converting optical to electrical signal). They are specified with certain transmitter power (dBm) and receiver sensitivity (dBm) along with achievable bit rate. The difference (in dB) of those two power levels dictates the optical link's maximum allowable power budget. For example, a 10 Gbps, 1310 nm, 10 km small form-factor pluggable (SFP) optical transceiver has the maximum possible transmit power of 0.5 dBm and receiver sensitivity of -14.4 dBm [58], resulting in $(0.5 - (-14.4)) = 14.9$ dB of allowable power budget. To make the link practically viable, the sum of all the loss components along the link should be less than such power budget. Optical transceivers require low additional power – a 10 Gbps and 100 Gbps transceiver consumes less than 1.5 W and 3.5 W, respectively [59].

Optical Fiber: Optical fiber is the transmission medium to carry light. Single-mode fiber is widely used due to its low cost and negligible attenuation loss (proportional to its length). Commercially available single-mode fiber optic cables typically have 0.3 dB of insertion loss and 0.36 dB/km of attenuation loss at 1310 nm wavelength [58]. Note that a $1:p$ optical splitter is already manufactured with $1+p$ optical fiber strands as its inputs and outputs.

The optical signal propagating through a ToR-to-ToR link in Shufflecast faces splitter insertion loss, fiber insertion loss, and attenuation loss, with the exact amount depending on the physical length of the link. For example, an optical link with 1 : 8 PLC splitter and 500 m single-mode fiber have total loss $\left(10.6 + 0.3 + 0.36 * \frac{500}{1000}\right) = 11.08$ dB and 10 Gbps, 1310 nm, 10 km SFP optical transceiver has a max allowable power budget 14.9 dB. Thus, there is a $(14.9 - 11.08) = 3.82$ dB margin, which makes the design of such links practically viable.

3.1.3.2 Topology

We discuss how we can build a Shufflecast topology using the optical devices mentioned above. Such a topology can be parameterized by p and k, where p denotes the number of ToRs a single ToR connects to, and k is the number of columns. Such a fabric is called a p, k-Shufflecast, with $N = k \cdot p^k$ ToR switches forming a p-regular graph. Any ToR is connected to p other ToRs of the next column. Each column has p^k ToRs, and this connectivity pattern is called a p-shuffle [60]. Moreover, the ToRs of the last column are connected to those of the first, resulting in the graph wrapping around as a cylinder.

Figure 3.1 shows an example of $2, 2$-Shufflecast, where there are 8 ToRs arranged in 2 columns, with 4 ToRs per column and each ToR equipped with 1 : 2 optical splitter (nodal degree 2).

3.1.3.2.1 Topological Properties

Scalability and Port Counts: Shufflecast can scale to arbitrary network size. For example, a $4, 4$-Shufflecast with 1 : 4 splitters and 4 columns can cover 1024 ToRs in a data center.

Hop Counts: Shufflecast routing can be performed with a low worst-case hop count. For a p, k-Shufflecast, all the ToRs are reachable from a given source by at most $2k - 1$ hops. For example, in Figure 3.1, the maximum hop count is 3. ToR 0's multicast packet reaches ToR 4 and 5 in 1^{st} hop. ToR 4 relays these packets to ToR 1, and ToR 5 relays to ToRs 2 and 3 at the 2^{nd} hop. At 3^{rd} hop, either of ToR 1 or 3 can relay the multicast packets to ToRs 6 and 7.

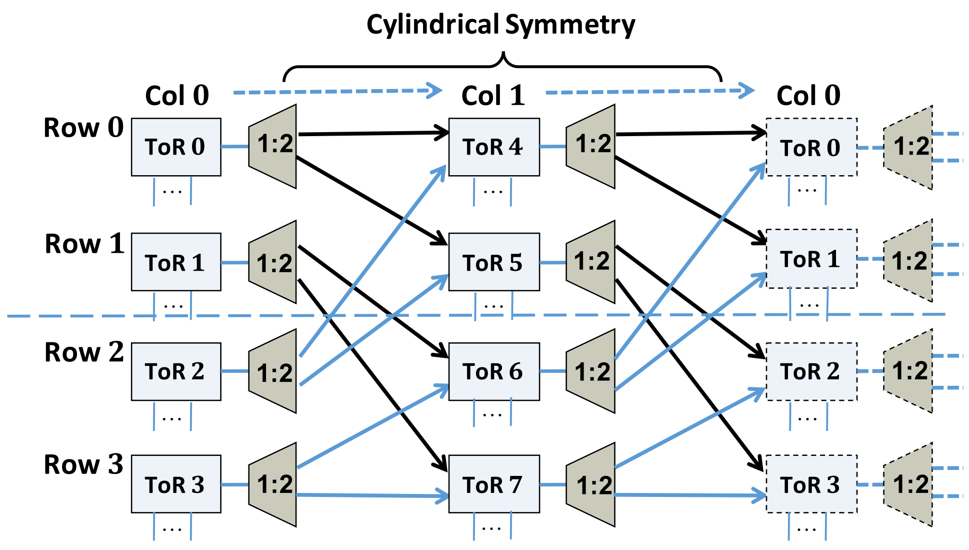


Figure 3.1 : Connectivity of 2, 2-Shufflecast.

3.1.3.3 Routing properties

Shufflecast multicast-aware routing provides static ToR-level relaying rules that depend only on the source ToR ID and has the ability to exploit all degrees of network parallelism.

Implicitly, p dictates the degree of parallelism for Shufflecast fabric. If we divide the rows of the topology by p, a ToR from each division can multicast in parallel at line-rate. In Figure 3.1 the dashed line shows this division. All the outgoing links from the upper division are marked with darker arrows, and those from lower division are marked with lighter arrows. For example, the source ToR 0 and ToR 3 can perform multicast simultaneously at line-rate. Figure 3.2 shows the route to all other ToRs from ToR 0 and ToR 3. We can see that ToR 0 and ToR 3 relay sets are disjoint, $\{0, 1, 4, 5\}$ and $\{2, 3, 6, 7\}$ for this reason, they can operate simultaneously at line-rate.

3.2 Implementation

We implement a prototype of 2, 2-Shufflecast in our testbed. Our setup uses 3 OpenFlow switches (2 ToR switches and one core switch), 8 optical splitters (1 : 2), and 16 servers. We logically divide the 2 ToR switches to emulate 4 ToR switches each, and 2 servers are connected to each logical ToR. We wire the Shufflecast network using optical splitters

Source ToR	Route to all other ToRs	Relay set for one-to-all multicast
0	0→4→1 , 0→5→2 , 0→5→3 , 0→4 , 0→5 , 0→4→1→6 , 0→4→1→7	{ 0 , 1 , 4 , 5 }
3	3→6→0 , 3→6→1 , 3→7→2 , 3→7→2→4 , 3→7→2→5 , 3→6 , 3→7	{ 2 , 3 , 6 , 7 }

Figure 3.2 : Relay sets for ToR 0 and ToR 3 in $2, 2$-Shufflecast.

on these 8 logical ToR switches. The core switch connects to the logical ToRs, creating a 2-layer full-bisection bandwidth network across ToR switches. Each server has 6 3.5 GHz CPU cores with 12 hyperthreads and 128 GB RAM. All connections are 10 Gbps Ethernet. To minimize the number of ports used while wiring the $2, 2$-Shufflecast, at each logical ToR switch, we connect the outgoing fiber (to its splitter) and one of the 2 incoming fibers (from 2 other splitters) to a single transceiver port. Thus, each logical ToR consumes only 2 transceiver ports (optimal for $2, 2$-Shufflecast). The forwarding rules are installed on the switches using the Ryu OpenFlow controller [61], running on one of the servers.

Based on this hardware setup, we use Republic [62], a publicly available platform, to handle packet transportation and to provide a simple unicast-based loss recovery for reliable IP multicast transmission. Shufflecast uses turn-taking as its application-level flow control mechanism among multiple multicast sources.

The controller program consists of two parts. The first part runs the Shufflecast multicast-aware routing algorithm (from Section 3.1.1 of paper [57]) and pre-installs the static ToR-to-ToR forwarding rules for $2, 2$-Shufflecast ($<$ 100 lines of python code written by the first author of Shufflecast). The second part translates application-based multicast group membership information into the ToR-to-server multicast rules and installs them on the switches at runtime ($<$ 30 lines of python code). We make simple modifications to applications to interact with the controller program ($\approx$ 10 lines of C++ code).

3.3 Experimental Results

In this section, we present comprehensive testbed experimental results to demonstrate that Shufflecast can achieve a) optimal and predictable multicast performance with low cost, b) high end-to-end reliability while supporting concurrent multicast groups with negligible overhead, and c) improved application performance for both high-bandwidth and low-latency applications.

3.3.1 Multicast performance of Shufflecast vs. state-of-the-art multicast mechanisms

We perform benchmarking experiments to evaluate the multicast performance of Shufflecast. Our baseline mechanisms are state-of-the-art multicast solutions over a full-bisection bandwidth network: a) IP multicast, and b) peer-to-peer mechanisms such as MPI_Bcast [22] and Spark-Cornet [3]. We perform a 1 : 15 multicast with varying data size (from 200 MB to 1.4 GB) and measure the multicast reading time (i.e. the duration between receiving program issues reading request and finishes reading it). Figure 3.3 shows the application-level multicast throughput defined as the ratio of multicast data size to multicast reading time. We observe that without any competing traffic, Shufflecast and IP multicast (note that IP multicast is using a full-bisection bandwidth network) achieve close to line-rate performance irrespective of multicast data size. However, in presence of competing unicast traffic, IP multicast (over full-bisection bandwidth network) without congestion control leads to unreliable and unpredictable performance (recall Section 2.1.1); similarly, IP multicast (over full-bisection bandwidth network) with congestion control (e.g., NORM [27]) fails to achieve fairness with unicast flows (recall Section 2.1.1). On the other hand, Shufflecast's performance remains unaffected by unicast traffic as it physically isolates multicast and unicast traffic.

We also observe that, even without any competing traffic, both MPI_Bcast and Spark-Cornet achieve application-level throughput only up to 35% and 20% of the line-rate throughput across data size, which is far from optimal. Shufflecast is up to 3× faster

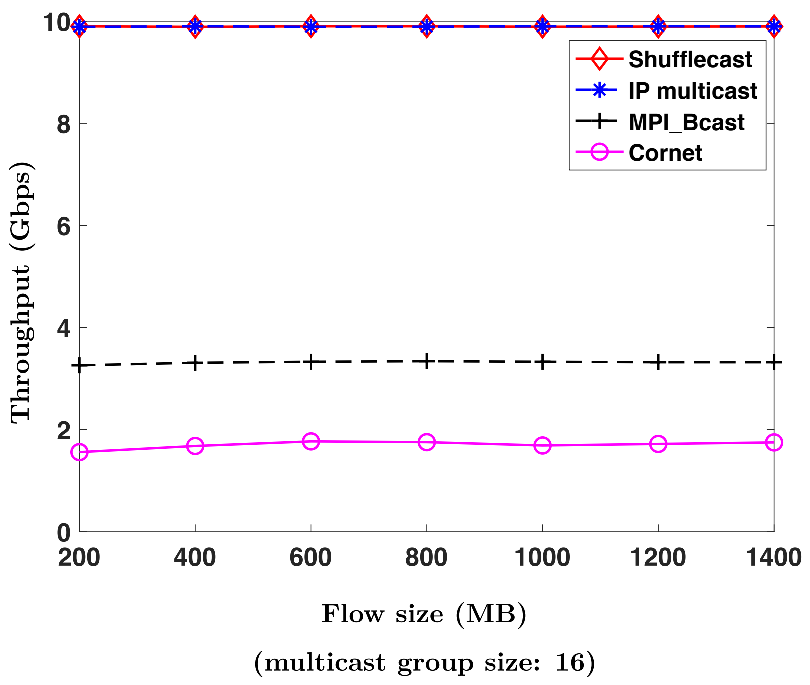


Figure 3.3 : Application-level Multicast throughput vs. data size for a 1 : 15 multicast flow.

than MPI_Bcast and up to 6.3× faster than Spark-Cornet.

3.3.1.1 Real-world applications

To demonstrate if multicast speedup of Shufflecast leads to significant application-level benefit, we use the same case studies as Chapter 2 Section 2.1.2.3.

3.3.1.1.1 Throughput sensitive

MPI: First, we compare the original MPI_Bcast to the modified version that uses Shufflecast. We choose one server under each ToR, and each server is allocated with one process while one of them multicasts a data chunk to seven other processes. Varying the multicast data size from 50 MB to 900 MB, we obtain the average multicast reading time over 10 iterations without any competing traffic. Figure 3.4(a) shows that Shufflecast considerably improves the average multicast reading time compared to MPI_Bcast, with a speedup of 2.33 × −2.8× across different data sizes. Moreover, the multicast reading

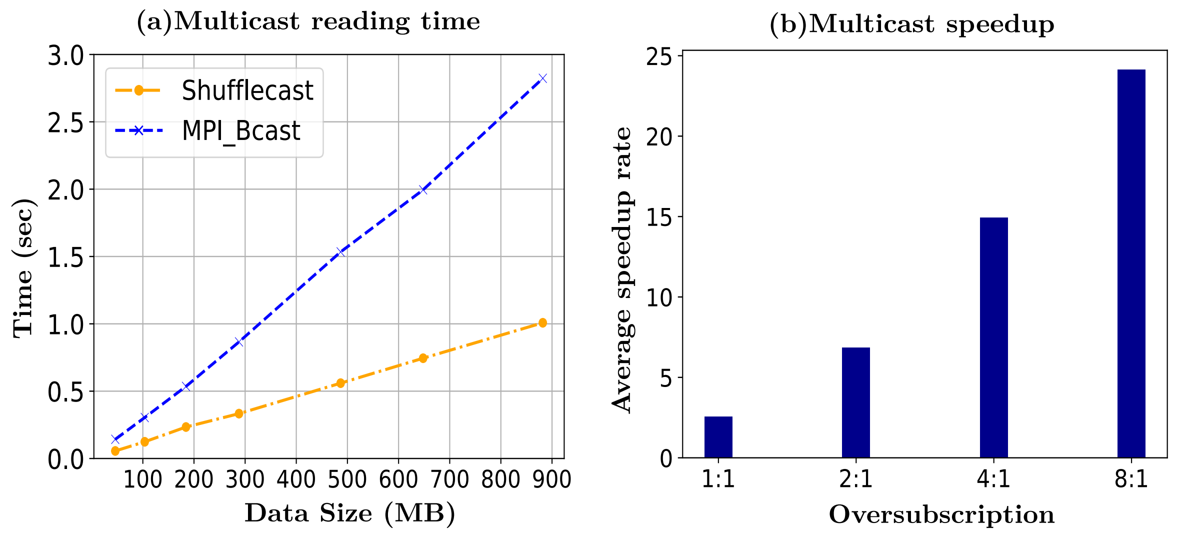


Figure 3.4 : (a)Multicast reading time of Shufflecast vs. MPI_Bcast. Shufflecast improves the multicast reading time considerably, and the speedup is consistent $(2.33\times - 2.8\times)$ across different data sizes. (b)Multicast speedup of Shufflecast over MPI_Bcast under different oversubscription. The average speedups of Shufflecast compared to MPI_Bcast under different oversubscription ratios range from $2.6\times - 24\times$.

time is consistent overall iterations, indicating the predictable performance of Shufflecast. At the lower network level, the multicast performance is close to line-rate. We perform similar experiments with scenarios where the unicast network is oversubscribed. Each scenario has a different oversubscription ratio created by injecting competing traffic using iPerf3 [50], which leaves different amounts of available bandwidth for the unicast network: 5 Gbps, 2.5 Gbps, and 1.25 Gbps. Figure 3.4(b) shows that the average speedups of Shufflecast compared to MPI_Bcast ranges from $2.6\times - 24\times$ across different oversubscription ratios.

Spark ML: Under Spark Machine Learning applications, we focus on two popular iterative machine learning algorithms: Latent Dirichlet Allocation (LDA) and neural word embedding. We use the Spark LDA implementation [39] with the dataset of 20 Newsgroups as the input corpus [40], and do the one-to-all multicast for the training vocabulary model (735 MB in size). In addition, we use the Word2Vec [41] neural word embedding application in Spark MLlib, with a training model size of 504 MB. Currently, both of these applications use Spark's native multicast mechanisms like Cornet [3] and HTTP (repeated unicasts to all receivers).

We compare the performance of Spark ML with and without Shufflecast. We use an extension to Spark that can perform multicast [62] over Shufflecast network. For this application, we use 8 servers. The application randomly chooses one server with four cores and 88 GB RAM as the master, while the other seven servers with two cores and 44 GB RAM serve as 14 slave executors. Both the Spark ML applications (i.e., LDA and Word2Vec) run for 10 iterations.

Finally, we obtain the total multicast reading time and application running time.

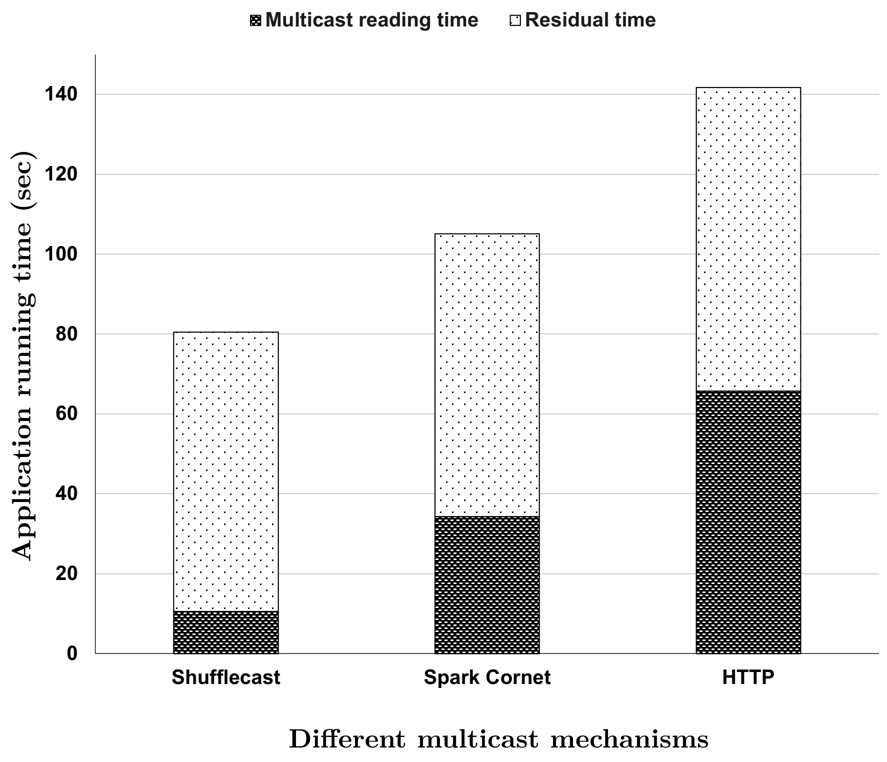


Figure 3.5 : LDA performance improvement of Shufflecast compared to native multicast mechanisms over full-bisection bandwidth network. The speedup in multicast reading time are 3.25× and 6.24× compared to Cornet and HTTP, respectively. The corresponding improvement in the application running time are 23.41% and 43.1%.

Figure 3.5 shows that Shufflecast improves the overall application running time in

Application	Multicast Mechanism	Multicast Reading Time (sec)	Application Running Time (sec)
Word2Vec	Shufflecast	11.43	973.97
	Spark Cornet	25.90	984.00
	HTTP	58.49	1022.10

Table 3.1 : Shufflecast improves the multicast reading time. There are 2.27× and 5.12× speedup in multicast reading time compared to Spark Cornet and HTTP, respectively.

LDA by accelerating the multicast reading time. Shufflecast achieves 3.25× and 6.24× speedup in multicast reading time compared to Cornet and HTTP, respectively. Overall, the application running time improvement is 23.41% and 43.1% compared to Cornet and HTTP, respectively.

Table 3.1 shows that Shufflecast improves the multicast reading time in Word2Vec however, because Word2Vec is more computation-intensive, so the multicast reading time plays a smaller role in overall application runtime. There is 2.27× and 5.12× speedup in the multicast reading time compared to Cornet and HTTP, respectively.

Next, we compare Spark distributed database with and without Shufflecast, keeping the same server configuration as Spark ML.

Figure 3.7 shows the application running time of each TPC-H query (queries 1 to 22) averaged over 10 runs. For certain queries (e.g., 1, 4, 6, 14, 15, 22), the amount of multicast data is either very small (under 200 MB) or non-existent, showing no visible difference between Shufflecast, Cornet, and HTTP. However, for other queries (e.g., 9, 17, 18), multicast data is large (5 GB), so Shufflecast gets speedup of 2.7× and 3.5× in multicast reading time compared to Cornet and HTTP respectively (Figure 3.6. Consequently, the total query running time improvement is 13.7% compared to Cornet and 17% compared to HTTP.

DMM: We use Fox [47, 63] as an example DMM algorithm where processes need to multicast chunks of one matrix in different iterations. Current MPI-based Fox implementation [48] uses MPI_Bcast as the multicast mechanism.

We have used a Fox implementation using the Open MPI library [49] (v1.6.5), which

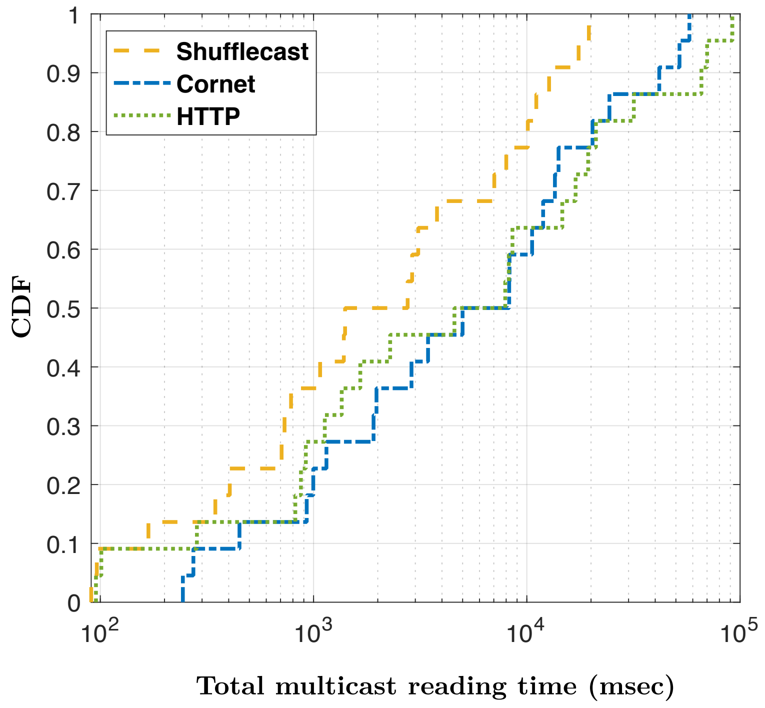


Figure 3.6 : CDF of TPC-H total multicast reading time. Shufflecast achieves 2.7× and 3.5× speedup in multicast reading time compared to Cornet and HTTP, respectively.

uses a chain-based communication algorithm for multicast.

We benchmarked with different numbers of machines (2/4/8) and processes (16/64/256) on varying matrix sizes (10k×10k-70k×70k) in 4 different scenarios. Each scenario has a different oversubscription ratio created by injecting competing traffic from iPerf3 [50]. We replace the multicast module with our multicast module that uses the Shufflecast API. We compare the basic version of MPI and the version supported by Shufflecast (by modifying the MPI multicast module to use Shufflecast APIs) using eight machines and 64 processes – this combination of machines/processes setting performs the best for the basic MPI version.

In Figure 3.8(a) the process to server and ToR mapping is in a way that only multicast traffic is sent outside of the rack, and all unicast traffic is contained within a rack, so only multicast traffic is impacted by oversubscription. Shufflecast performance remains close to line-rate across scenarios, as it uses a dedicated multicast network instead of the oversubscribed unicast network. For MPI, the multicast rate decreases with heavier

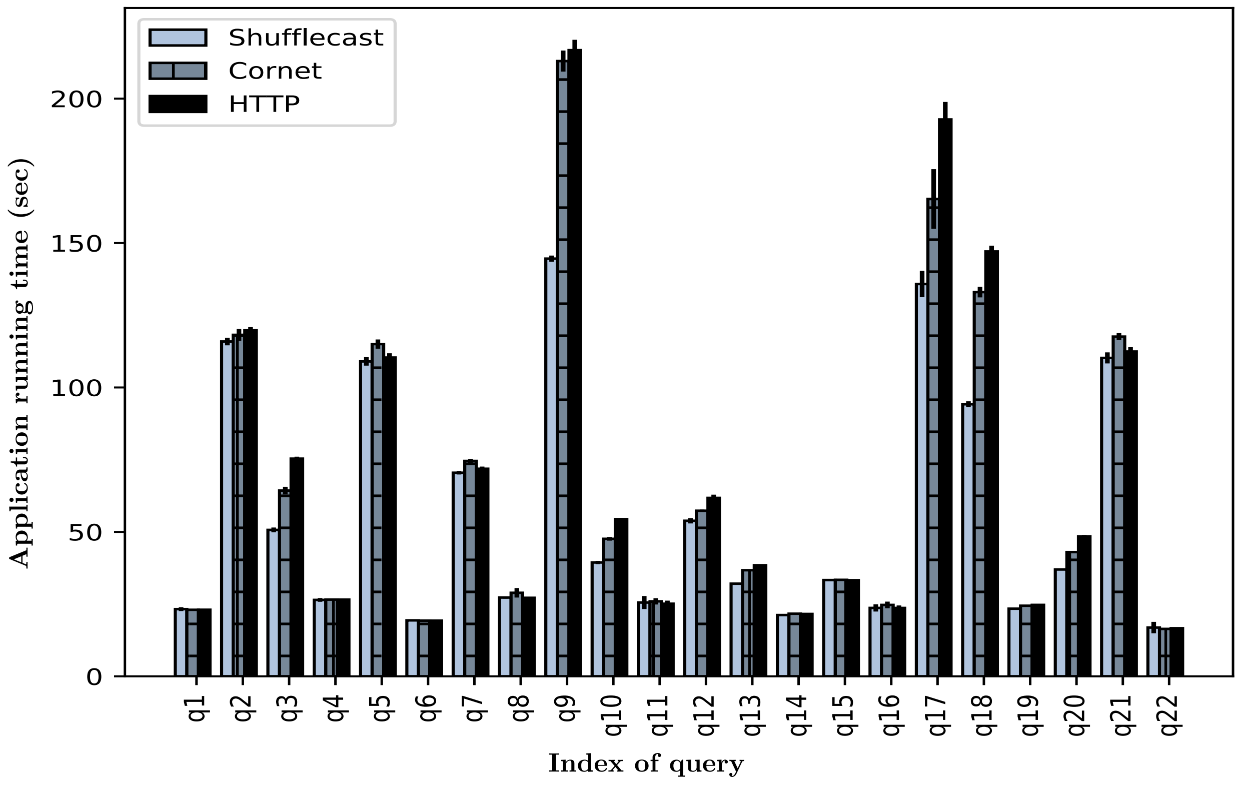


Figure 3.7 : The average running time of each TPC-H query (q1 to q22) over the three multicast mechanisms: For specific queries (e.g., 1, 4, 6, 14, 15, 22), the amount of multicast data is either minimal (under 200 MB) or non-existent, so there is no visible difference between Shufflecast, Cornet, and HTTP. However, for other queries (e.g., 9, 17, 18), the multicast data is large (5 GB), so Shufflecast reduces the multicast reading time significantly.

congestion.

We ran each combination of matrix sizes, scenarios, and multicast mechanisms 10 times and obtained their average running time. Figure 3.9 shows the average multicast reading time over the average total application running time. Here, the multicast reading time is defined as the duration from (1) the time when the program issues the block reading request running in the receiving process to (2) the time when the program finishes reading the block. The total application running time consists of multicast time, shift time, computation time and waiting time. Note that waiting time is only present in

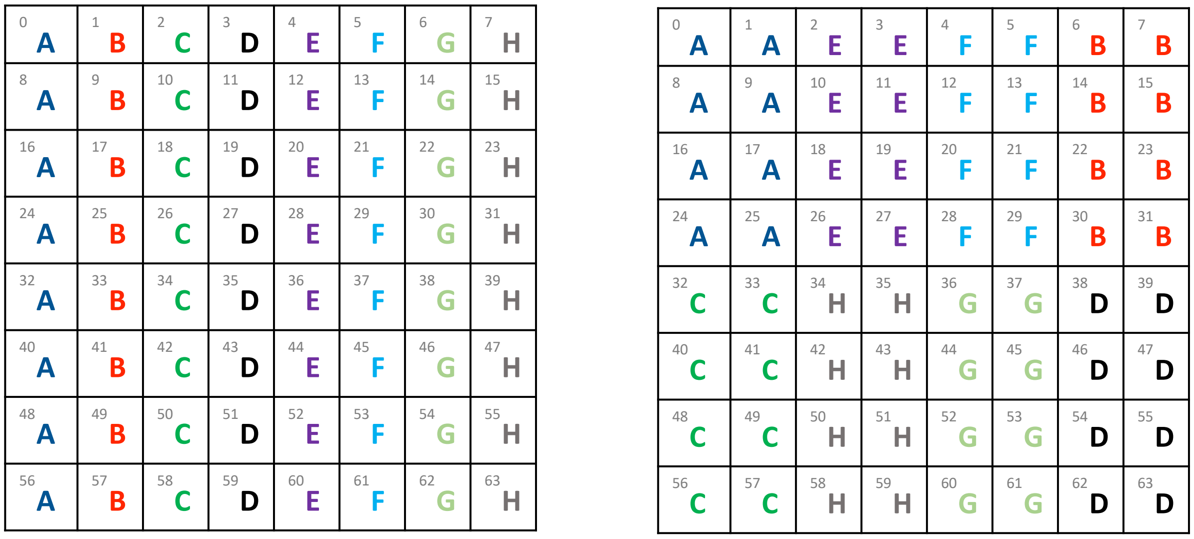

0 A 1 B 2 C 3 D 4 E 5 F 6 G 7 H
8 A 9 B 10 C 11 D 12 E 13 F 14 G 15 H
16 A 17 B 18 C 19 D 20 E 21 F 22 G 23 H
24 A 25 B 26 C 27 D 28 E 29 F 30 G 31 H
32 A 33 B 34 C 35 D 36 E 37 F 38 G 39 H
40 A 41 B 42 C 43 D 44 E 45 F 46 G 47 H
48 A 49 B 50 C 51 D 52 E 53 F 54 G 55 H
56 A 57 B 58 C 59 D 60 E 61 F 62 G 63 H
0 A 1 A 2 E 3 E 4 F 5 F 6 B 7 B
8 A 9 A 10 E 11 E 12 F 13 F 14 B 15 B
16 A 17 A 18 E 19 E 20 F 21 F 22 B 23 B
24 A 25 A 26 E 27 E 28 F 29 F 30 B 31 B
32 C 33 C 34 H 35 H 36 G 37 G 38 D 39 D
40 C 41 C 42 H 43 H 44 G 45 G 46 D 47 D
48 C 49 C 50 H 51 H 52 G 53 G 54 D 55 D
56 C 57 C 58 H 59 H 60 G 61 G 62 D 63 D

Figure 3.8 : Two different process to server/machine mappings for DMM. A-H represent 8 servers. One server is connected to each logical ToR.

Shufflecast due to the turn-taking flow control mechanism when multiple sources multicast data. All but one source need to wait on an $MPI\_Barrier$.

Shufflecast improves multicast reading time considerably. For the matrix size 70k×70k, it achieves speedups of 6.04×, 19.59×, 39.0×, and 52.1×, compared to MPI with 10 Gbps, 5 Gbps, 2.5 Gbps, and 1.25 Gbps, respectively.

Figure 3.9 further shows that this translates to application-level speedups, by 5.8% (5 Gbps), 20.4% (2.5 Gbps), and 28.6% (1.25 Gbps), respectively. However, the overall application runtime gets hurt compared to the 10 Gbps case. Furthermore, due to the turn-taking mechanisms used for Shufflecast implementation, the number of MPI_barrier added to the DMM program causes waiting times not present in the original version of the DMM program. Therefore we see that the upper part of the bar is more significant for Shufflecast.

3.3.1.1.2 Latency sensitive

Paxos-based consensus protocol: The performance of Paxos is sensitive to latency as the messages tend to be small. We choose the data object size as 1 Byte. We run

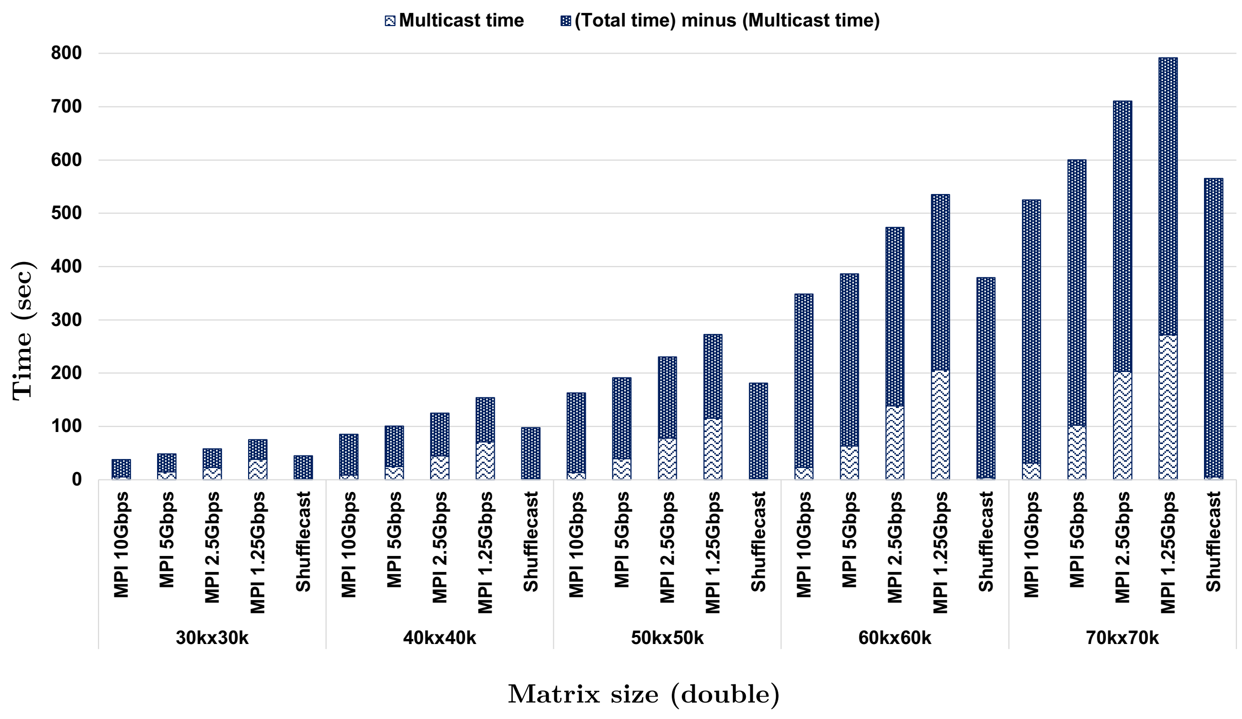


Figure 3.9 : The average multicast time and application running time of MPI and Shufflecast in all the four scenarios for matrix sizes 30k×30k to 70k×70k.

multicast-based Paxos [64] (natively leverage network-level multicast) over Shufflecast (no application modification required) and unicast-based Paxos [65] (repeated-unicasts to realize multicast) over full-bisection bandwidth network.

We run Paxos, where the client repeatedly sends 1 Byte values to the proposer. The client sends the next value as soon as the previous is successful and repeats for one hundred iterations; each iteration provides a latency measurement. All acceptors are placed on different servers. We compare the latency of unicast-based Paxos (over the full-bisection bandwidth network) with that of multicast-based Paxos (over Shufflecast). Figure 3.10 shows the 90th percentile latency of all values proposed. We observe that using Shufflecast, Paxos' 90th percentile latency is very predictable (close to 40 microseconds) and stable even as the number of acceptors increases. In contrast, the 90th percentile latency of unicast-based Paxos increases significantly as the number of acceptors increases and is at least two times worse than Shufflecast-enabled Paxos.

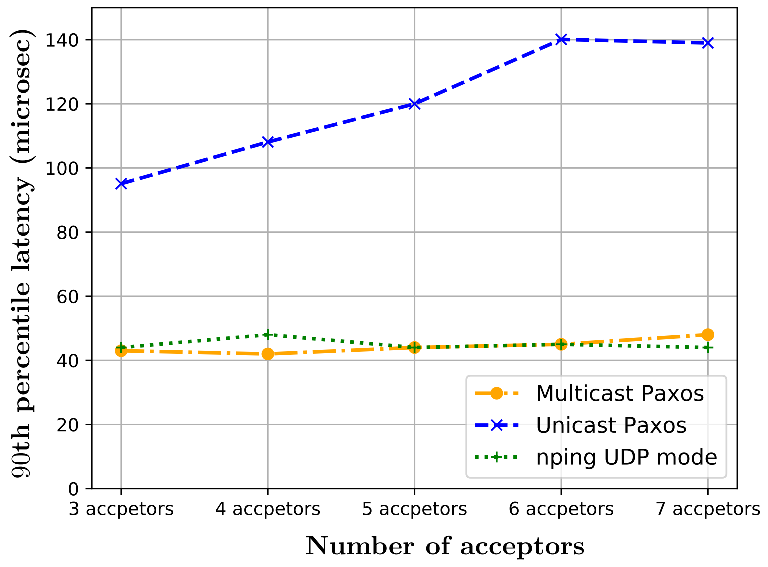
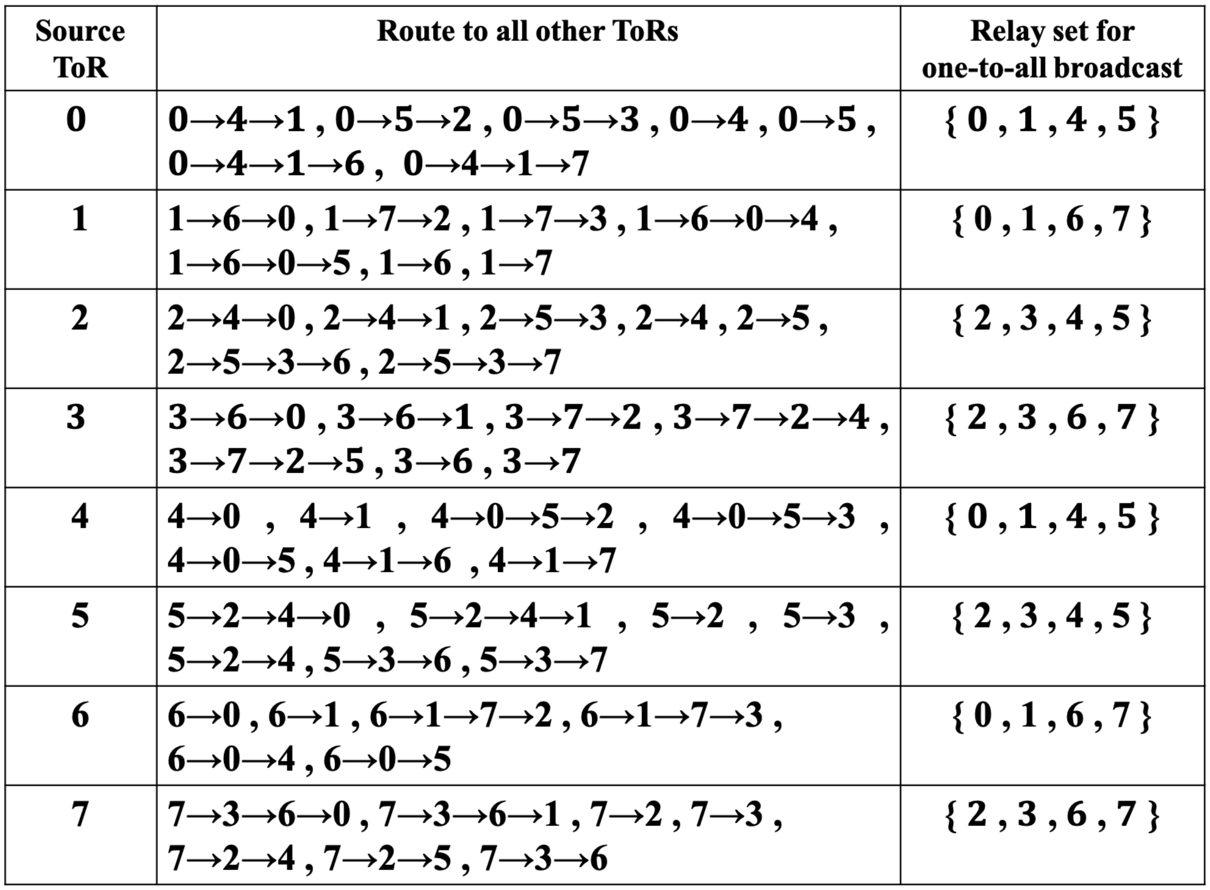


Figure 3.10 : 90th percentile latency of multicast Paxos over Shufflecast versus unicast Paxos over full-bisection bandwidth network with one sender. The latency of Unicast Paxos scales linearly with the number of acceptors (receivers), while multicast based Paxos over Shufflecast has almost constant and low latency.

3.4 Limitations

The limitations come from two different aspects of Shufflecast: 1) Implementation and 2) Architecture.

In MPI programs, multiple processes get assigned to each machine. Those processes have the same IP address. In the implementation of Shufflecast, we use Republic [62] to handle packet transportation and to provide a simple unicast-based loss recovery for reliable IP-multicast transmission. Republic does not allow an IP address to be part of multiple active multicast groups simultaneously. Because of this constraint, we choose to: 1) use turn-taking as our application-level flow control mechanism among multiple multicast sources so that there are no multiple active multicasts at the same time, and 2) place only one process from a multicast group under a machine (or ToR since there is also only one machine per ToR in this Shufflecast testbed) (Figure 3.8(a)). We saw that applications with parallel multicasts such as DMM would suffer from the introduced



Source ToR	Route to all other ToRs	Relay set for one-to-all broadcast
0	0→4→1 , 0→5→2 , 0→5→3 , 0→4 , 0→5 , 0→4→1→6 , 0→4→1→7	{ 0 , 1 , 4 , 5 }
1	1→6→0 , 1→7→2 , 1→7→3 , 1→6→0→4 , 1→6→0→5 , 1→6 , 1→7	{ 0 , 1 , 6 , 7 }
2	2→4→0 , 2→4→1 , 2→5→3 , 2→4 , 2→5 , 2→5→3→6 , 2→5→3→7	{ 2 , 3 , 4 , 5 }
3	3→6→0 , 3→6→1 , 3→7→2 , 3→7→2→4 , 3→7→2→5 , 3→6 , 3→7	{ 2 , 3 , 6 , 7 }
4	4→0 , 4→1 , 4→0→5→2 , 4→0→5→3 , 4→0→5 , 4→1→6 , 4→1→7	{ 0 , 1 , 4 , 5 }
5	5→2→4→0 , 5→2→4→1 , 5→2 , 5→3 , 5→2→4 , 5→3→6 , 5→3→7	{ 2 , 3 , 4 , 5 }
6	6→0 , 6→1 , 6→1→7→2 , 6→1→7→3 , 6→0→4 , 6→0→5	{ 0 , 1 , 6 , 7 }
7	7→3→6→0 , 7→3→6→1 , 7→2 , 7→3 , 7→2→4 , 7→2→5 , 7→3→6	{ 2 , 3 , 6 , 7 }

Figure 3.11 : Relay sets for ToR 0 to ToR 7 in 2, 2-Shufflecast.

waiting time caused by turn-taking. Shufflecast improves the multicast time, but the overall application runtime is higher than a full-bisection bandwidth network.

If we consider the same process to server mapping in Figure 3.8(a), assuming that there are no constraints, then DMM can start all the 8 multicasts simultaneously without turn-taking.

Recall that in a p, k-Shufflecast, P ToR's can simultaneously multicast to the rest of p, k-Shufflecast network with line-rate. It means that in a 2, 2-Shufflecast, we can only have 2 well-placed multicasts in parallel at line-rate. However, the DMM application starts a multicast from all the 8 ToRs simultaneously. The relays are shared (shown in Figure 3.11). The multicast rate will be 2.5 Gbps.

Another limitation is that the multicast tree rules installed on the switches are static. Thus, if the multicast group defined by the application should reach only a subset of

ToRs, It will still reach all ToRs in Shufflecast even if that ToR is not a relay for this subset. Figure 3.8(b) shows another process to server mapping for the DMM application. With this mapping there are two DMM row-wise multicast groups: { ToR 0, ToR 4, ToR 5, ToR 1} and { ToR 2, ToR 7, ToR 6, ToR 3}. However, Shufflecast will multicast to ToR 0 to ToR 7 for both groups; again, multiple simultaneous multicasts will share the relays. Therefore, multicasting with line-rate would not be possible.

3.5 Summary

Shufflecast architecture leverages inexpensive optical splitters and cleverly uses edge network bandwidth to segregate physically multicast traffic from unicast traffic, and the network becomes responsible for multicast. Shufflecast's data plane is scalable and supports line-rate throughput; its control plane is responsive and straightforward. Our experiments using a complete hardware and software prototype of Shufflecast show that Shufflecast can achieve high throughput, low latency reliable multicast for various real-world applications. Note that this solution cannot optimize unicast communications and perform more than p well-placed multicasts simultaneously with line-rate.

Chapter 4

Evaluate the benefits of a minimal data center network augmentation providing flexibility and reconfigurability

In Chapter 3, we evaluated the addition of a multicast dedicated network to optimize HPC and big data applications through minimizing group communication latency and maximizing group communication throughput. We observed that this solution could primarily benefit specific applications with certain characteristics, such as not having multiple simultaneous multicasts. For applications with multiple simultaneous multicasts, either the overhead added by turn-taking neutralize/surpass the improvement caused by minimizing multicast time, or it cannot multicast with line-rate.

We are interested in solutions that improve the application's runtime by minimizing the communication time for "**all**" HPC applications with zero to an insignificant overhead. At the same time, we want to retain as many of the existing benefits of the current data center networks as possible without significantly changing the network and increasing the hardware cost. How can we better use the DCN bandwidth?

Data center network (DCN) architectures are critical for achieving high throughput and low latency and maintaining low cost and complexity. To meet these goals, researchers have proposed a series of DCN architectures [66, 17, 16, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79] over the past decade. Although these proposals have competing designs for the network core, the designs for the network edge are similar: servers are organized in racks. The network core connects multiple racks, and each rack hosts tens of statically connected servers via a ToR switch. Standardized racks enable unified power supply and cooling, as well as significant space and cable savings. This rack-based topology and connectivity pattern is deeply ingrained in the design of existing DCN architectures. However, the drawback of this rack-based connectivity is also prominent. On the one hand, DCNs fragment the server pool into isolated racks; although this provides ease of management, the networks that connect racks are typically oversubscribed at the core to reduce equipment and operational cost [80, 81]. With typical oversubscription ratios somewhere between 4:1 to 20:1 [16, 80, 81], servers in the same rack enjoy line-rate throughput and low latency, but servers communicating across racks have much lower available bandwidths. On the other hand, traffic across racks is increasingly dominant in data center workloads [81, 82, 83, 84]. Firstly, more and more DCN traffic is escaping the rack boundary due to resource fragmentation [85], large scale jobs [46], special application constraints [86], and type-based server placement strategy [81]—e.g., one rack may host storage servers, and another rack may host cache servers. Secondly, there is also an increasing amount of traffic that leaves the pod. For instance, a web-frontend cluster may need to retrieve data from a database cluster or submit jobs to Hadoop clusters [81].

As a result, the core links in the oversubscribed layers are usually heavily utilized by flows across rack, and the edge links within racks are extremely under-utilized due to congestion higher up in the hierarchy; in some scenarios, more than 98% of the links observe less than 1% utilization [82].

Existing work mostly views the limitations of the rack design as a given and designs around them. Non-blocking network and its alternatives [16, 17, 87, 74, 88, 89, 79, 72, 73, 71, 66, 90] aim to enlarge the capacity of the network core to avoid congesting the over-subscribed layer, but the cost to build, operate, and upgrade those non-blocking networks will be much higher. Due to the scaling limit of CMOS-based electrical packet switches [91, 92, 92, 93, 94, 95, 96, 97, 80], building such a network while staying within the datacenter power budget is challenging. Rack-level reconfigurable networks [67, 69, 78, 77, 75] add additional bandwidth between the most intensively communicating racks with extra cables, lasers, or antennas to relieve the bottleneck at core links. But the performance improvement is constrained by the fact that the number of additional paths is usually limited. Besides better topology designs, smarter job placement and execution strategies [98, 84, 99, 100, 101, 3, 102, 103, 104, 105] can also reduce the cross-rack traffic by arranging the jobs based on their traffic pattern. However, these solutions generally cannot perform well if traffic patterns fluctuate at runtime [106, 107, 108].

Wu et al.[109] propose a rackless DCN architecture called RDC with the goal of relieving DCN congestion with topological reconfigurability at the edge. RDC logically removes the rack boundary of traditional data centers and the inefficiencies that come with it. As modern applications generate more and more cross-rack traffic, the traditional architecture suffers from contention at the core, imbalanced bandwidth utilization across racks, and longer network paths. RDC addresses these limitations by enabling servers to logically move across the rack boundary at runtime; moreover, it inherits desirable properties of the Clos topology, such as ease of deployment, maintenance, and expansion. RDC design achieves this by inserting circuit switches at the network edge between the ToR switches and the servers, and by reconfiguring the circuits to regroup servers across racks based on traffic patterns.

Adopting a reconfigurable network such as RDC should minimize both point-to-point and group communications across racks. We expect RDC to speed up an oversubscribed network to achieve near non-blocking network's performance which benefits the overall runtime.

We want to see how much benefit it brings to the overall runtime. A tutorial of RDC is given in Section 4.1. We have performed extensive evaluations of RDC in a hardware testbed (Section 4.2). RDC can speed up a 4:1 oversubscribed network to achieve nearly non-blocking network's performance. Note that to improve the communication performance of an HPC application executing on RDC, users need to understand how the application's communication patterns interact with the network, especially when those interactions result in congestion and instruct the application with the reconfiguration code.

4.1 RDC

4.1.1 Introduction

The rackless data center (RDC) architecture goal is to *remove* the fixed, topological rack boundaries while preserving the benefits of rack-based designs, e.g., ease of power supply, cooling, and space savings. In RDC, servers remain mounted on physical racks, but they are not bound statically to any ToR switch. Instead, servers can move logically from one ToR to another. This is achieved by using circuit switches, which can be dynamically reconfigured to form different connectivity patterns. Circuit changes can shift the servers to different topological locations while the servers remain immobile.

The power of RDC stems from the fact that servers can dynamically form logical groups that are optimized for the current traffic pattern. With this novel architecture that is not committed to any static configuration, servers that heavily communicate with each other can be grouped on-demand, and they can be regrouped as soon as the pattern changes again. This leads to performance benefits in many common, real-world scenarios: 1)Large jobs fragmented across racks; 2)Workloads with dynamic traffic patterns; 3)Applications with placement constraints; 4)Imbalanced out-of-pod traffic.

4.1.2 Potential benefits of rackless DCN design

The authors claim that RDC's reconfigurability leads to performance benefits in many common, real-world scenarios.

▶ **1: Mitigate the effect of resource fragmentation.** Large jobs cannot be placed on the same rack because they may not fit within the rack resources due to resource fragmentation; under RDC, servers that run the same applications can be regrouped for efficient communication. RDC can mitigate the effect of resource fragmentation reduce this inefficiency to a minimum by regrouping the servers smartly.

Since many cluster schedulers assign the resources greedily to each job without a global view[110, 111, 112], resource fragmentation is common and inevitable in the current data centers[113, 114]. Therefore, cross-rack servers may be involved in many jobs, resulting in cross-rack traffic that easily congests the oversubscribed core. In contrast, RDC can relocate servers logically for server groups that heavily communicate, reducing cross-rack traffic. As shown in Figure 4.1(a), RDC regroups the servers and localizes the cross-rack traffic from the two jobs.

▶ **2: Optimize for dynamic traffic patterns.** Workloads with changing traffic patterns can benefit from server regrouping at runtime to localize heavily communicating servers, enabling more network communication to enjoy full bandwidth and low latency. Even with flexible resource grouping, RDC may not localize all the traffic in some scenarios, like very large job scale [115], multi-tenant cloud [116], and infrastructure load-balancing [117], due to the limitation of the physical rack volume, power supply, and cooling requirement. For instance, applications like deep learning [118] and database [115] cannot be supported by a single rack to achieve both high throughput and low response time. Thus, the cross-rack traffic can grow very high, and out-of-pod traffic demand for different racks can also vary in the above scenarios. And the situation can be even worse if the traffic pattern keeps changing. However, even with such difficulties, RDC can still optimize the placement by rearranging the servers to balance the link load and localize the traffic as much as possible. When the traffic pattern changes, RDC can also reconfigure accordingly. As an example, Figure 4.1(b) depicts a large scale job and RDC finds an optimal way to localize most of the previous cross-rack traffic.

▶ **3: Accommodate application placement constraints.** Applications may intentionally spread their instances across racks for fault tolerance [86] or reduce synchronized power consumption spikes [119]. For example, to increase the system's resilience, some distributed storage systems, like HDFS, always require at least one data block replica to be placed on a different rack. RDC can localize traffic while accommodating application placement constraints. Figure 4.1(c) illustrates this mechanism. RDC can send the replicas to a different physical rack but within the same "logical" rack by regrouping the servers from different racks into one logical rack. This mechanism provides higher

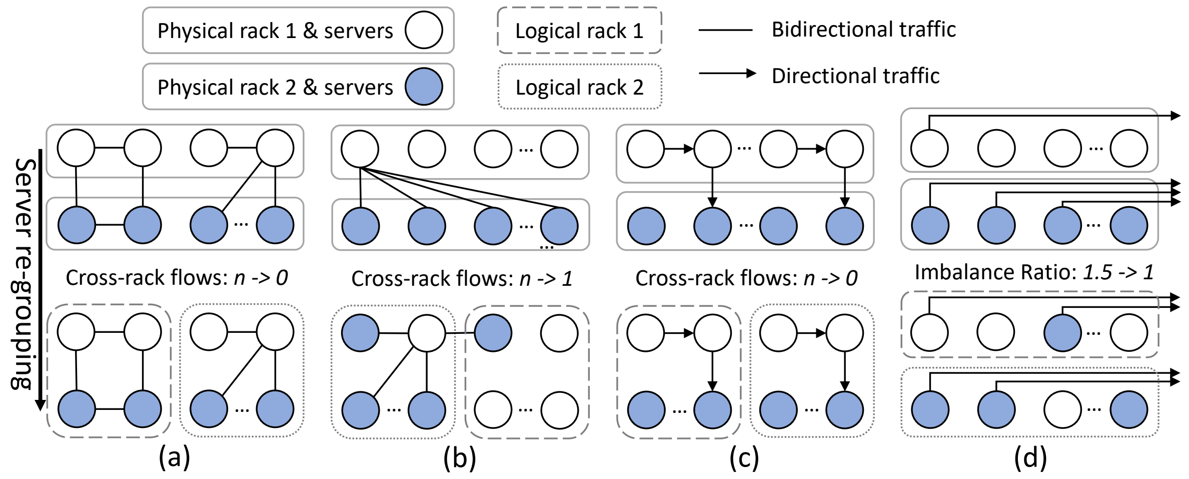


Figure 4.1 : Comparisons between before and after server regrouping for (a) placement optimization, (b) flexible server grouping for large jobs, (c) application constraints accommodation, and (d) out-of-pod load balancing.

bandwidth and also satisfies the replica placement policy of HDFS.

▶ **4: Balance out-of-pod traffic.** RDC can balance out-of-pod traffic. Traffic patterns across racks may be heavily skewed. RDC can redistribute servers across different racks; this allows rack uplinks to be load-balanced to mitigate congestion. RDC can regroup the servers according to their out-of-pod traffic demands and balance link utilization, hence relieving the previous bottleneck. In Figure 4.1(d), the imbalance ratio has been decreased to 1 from 1.5 after the grouping is changed according to the out-of-pod traffic demand.

4.1.3 The RDC Architecture

In this section, we first describe the key building block of RDC and then describe the RDC architecture. One circuit switch is inserted at the edge layer between servers and ToR switches per pod in this architecture. We describe the key features of the circuit switching technology below.

Circuit switching: Circuit switches can be dynamically reconfigured to form different connectivity patterns by establishing a circuit, from an entry to an exit servers/port.

The circuit guarantees the full bandwidth of the channel. The circuit functions as if the servers are physically connected. RDC uses circuit switching technology to achieve the design goal of providing pod-level reconfigurability. Supporting rackless architecture for networks with pods containing tens of racks and several hundreds of servers requires circuit switches with O(1000) ports. Circuit switches scale to tens of thousands of ports with switching delay on the order of microseconds (Non-MEMS) [68] to several milliseconds (3D MEMS-based optical circuit switches (OCS)) [120, 121]. Optical circuit switches do not encode, decode, or buffer packets, so they are protocol and data rate transparent, providing high bandwidth at very low power [121].

4.1.3.1 Connectivity structure

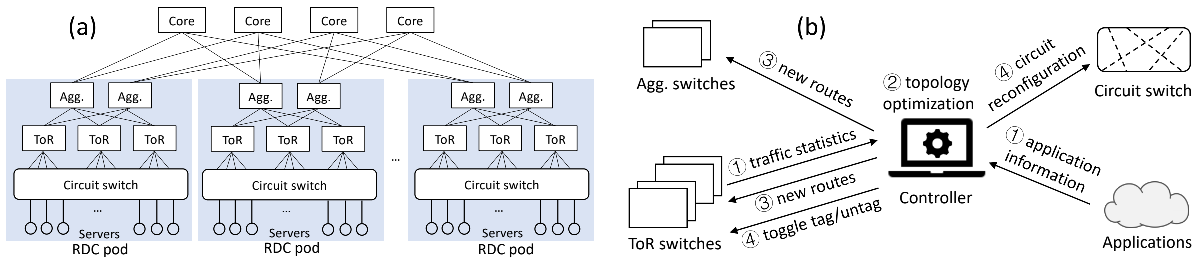


Figure 4.2 : RDC architecture and workflow overview. (a) is an example of the RDC network topology. Circuit switches are inserted at the edge between servers and ToR switches. Connectivities for aggregation switches (*agg.*) and core switches remain the same as in traditional Clos networks. (b) presents an overview of the workflow.

RDC uses circuit switches to achieve reconfigurable server-ToR connectivities. With circuit switches, software controllers can manage the circuit setup and reconfiguration, e.g., the TL1 interface. Circuits can be reconfigured independently of each other, so only inflight traffic traversing the reconfigured circuits would be disturbed; moreover, source hosts can detect the link-down event and buffer the unsent traffic for fast recovery. Since the link-down time is very short and only servers that need to change their rack membership have to experience this downtime, RDC will not cause significant traffic disruption. As for reconfiguration algorithms, we defer the discussion to Section 4.1.4.

Figure 4.2(a) shows an example of the RDC pods. RDC changes the traditional multi-layer Clos topology [16, 17] by inserting circuit switches at the edge layer between servers and ToR switches. The aggregation and core layers of the network remain the same. One circuit switch is used per pod. Each pod has m racks with n servers per rack, thus requiring $2mn$ ports on the circuit switch—half of the ports are connected to servers while the other half are connected to ToR switches. For example, a 16-rack pod with 32 servers per rack requires 1024 circuit switch ports. In traditional data centers, each server has a fixed connection to a single ToR; In contrast, RDC enables full flexibility to permute the server-ToR connectivities, allowing the most intensively communicating servers to be localized and enjoy the line-rate.

4.1.3.2 The pod controller

Today's data centers are constructed from modular pods [122, 123, 124, 125], where a pod typically hosts one type of service. RDC similarly views pods as basic units and uses a per-pod network controller that manages both packet switches and the circuit switch within the pod. The controller reconfigures the network at timescales of seconds or longer depending on the traffic pattern. The controller receives the traffic demand from the applications directly.

The workflow is illustrated in Figure 4.2(b). The controller 1) receives the information from the applications; 2) determines the optimized topology with certain optimization goals; 3) generates a set of new routes and pre-installs them on the packet switches; and 4) finally sends the circuit reconfiguration request to the circuit switch and simultaneously activates the new routing rules on packet switches. The first two steps serve as the RDC control plane, while the last two steps configure the data plane. The final step causes a small amount of disturbance due to the circuit reconfiguration delay.

4.1.4 RDC Control Algorithms

RDC allows applications to explicitly request reconfigurations and export their traffic demand and other essential information to the controller via RPC. RDC can localize traffic based on application demands. By moving the previous cross-rack traffic to be intra-rack, RDC improves the aggregated bandwidth and reduces the average latency. When applications have changing traffic patterns (e.g., distributed matrix multiplication (DMM) algorithms proceed in iterations with shifting traffic patterns), they can request reconfigurations before the next phase starts to ensure locality throughout the job. Section 4.2 presents a detailed evaluation of RDC.

4.2 Implementation and Evaluation

We conduct comprehensive evaluations using testbed experiments to demonstrate that RDC can improve application performance for applications. Our experiments focus on real-world applications of RDC to MPI-based distributed matrix multiplication (DMM) [106] as a use case.

Testbed. Our RDC prototype consists of 16 servers and 4 ToR switches in 4 logical racks, as well as one *agg.* switch and one circuit switch; Figure 4.3 illustrates our hardware testbed. The ToR switches are emulated on two 48-port Quanta T3048-LY2R switches. Each ToR switch has four 10 Gbps downlinks connected to the servers, and one 10 Gbps uplink to the *agg.* switch, forming an oversubscription ratio of 4:1. We can tune this ratio to emulate a non-blocking network by increasing the number of uplinks to 4. The *agg.* switch is a separate OpenFlow switch. The OCS is a 192-port Glimmerglass 3D-MEMS switch with a switching delay of several milliseconds. Each server has six 3.5 GHz dual-hyperthreaded CPU cores and 128 GB RAM, running TCP CUBIC on Linux 3.16.5.

4.2.1 Real-world application

We evaluate how RDC can improve the performance of real-world applications. To demonstrate if the communication speedup of RDC leads to significant application-level benefit, we use the same case study as Chapter 3 Section 3.3.1.1.1.

MPI DMM. We set up a 8-server OpenMPI cluster and a 16-server OpenMPI cluster across 4 racks and implemented a commonly used DMM algorithm [106] with 16 and 64

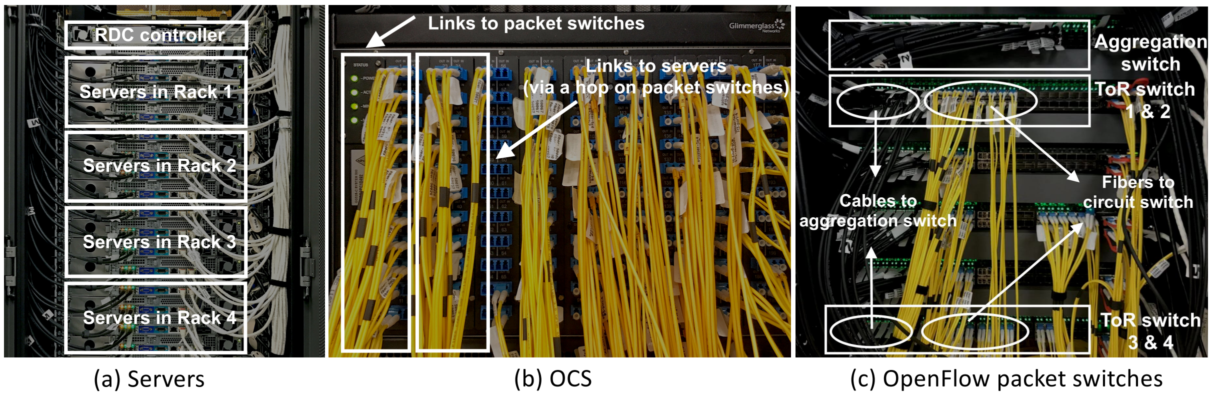


Figure 4.3 : RDC prototype with 4 racks and 16 servers.

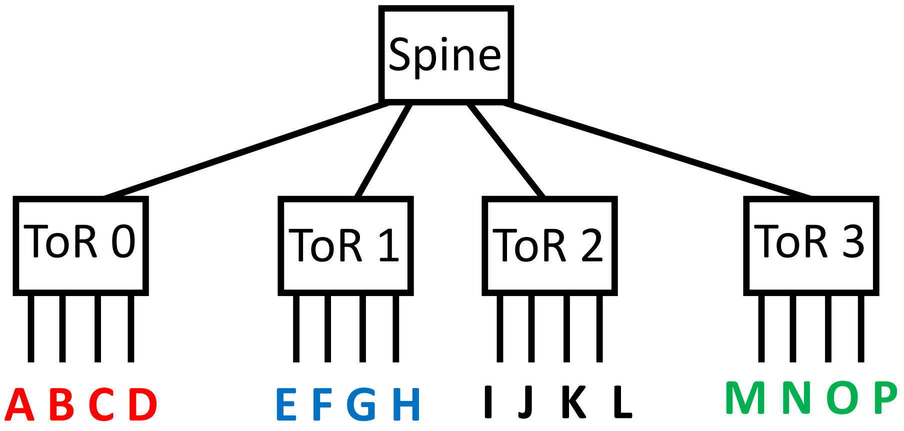


Figure 4.4 : The DMM testbed. A-P represent 16 servers, and A-D, E-H, I-L, M-P belong to four physical racks separately.

processes. Figure 4.4 shows the DMM testbed and the server to rack color coding. We benchmarked with different numbers of servers (8/16) and processes (16/64) on varying matrix sizes (10k×10k-100k×100k). The combination of 16 servers and 64 processes performs better than 8 servers and 64 processes and 8 servers and 16 processes. Therefore, we only show the results of the combination of 16 servers and 64 processes in this section.

Matrices are divided into 64 blocks (submatrices). Each server has 4 processes to form an 8×8 process layout. Reminder: DMM in each iteration performs a "multicast-multiply-roll" cycle where a process a) multicasts submatrix row-wise, b) multiplies submatrices, and c) shifts submatrices column-wise as shown in Figure 4.5. We consider six placements for the 64 processes over 16 servers: 1) Figure 4.6(a): places them row-wise zero cross-

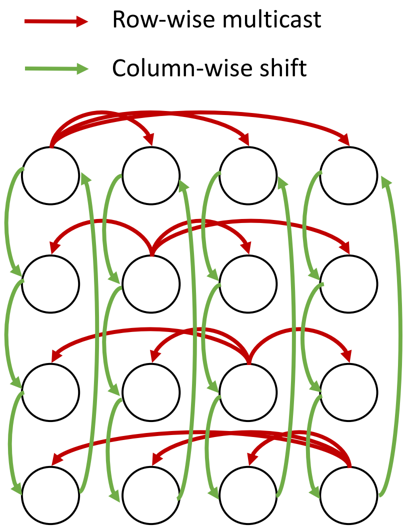
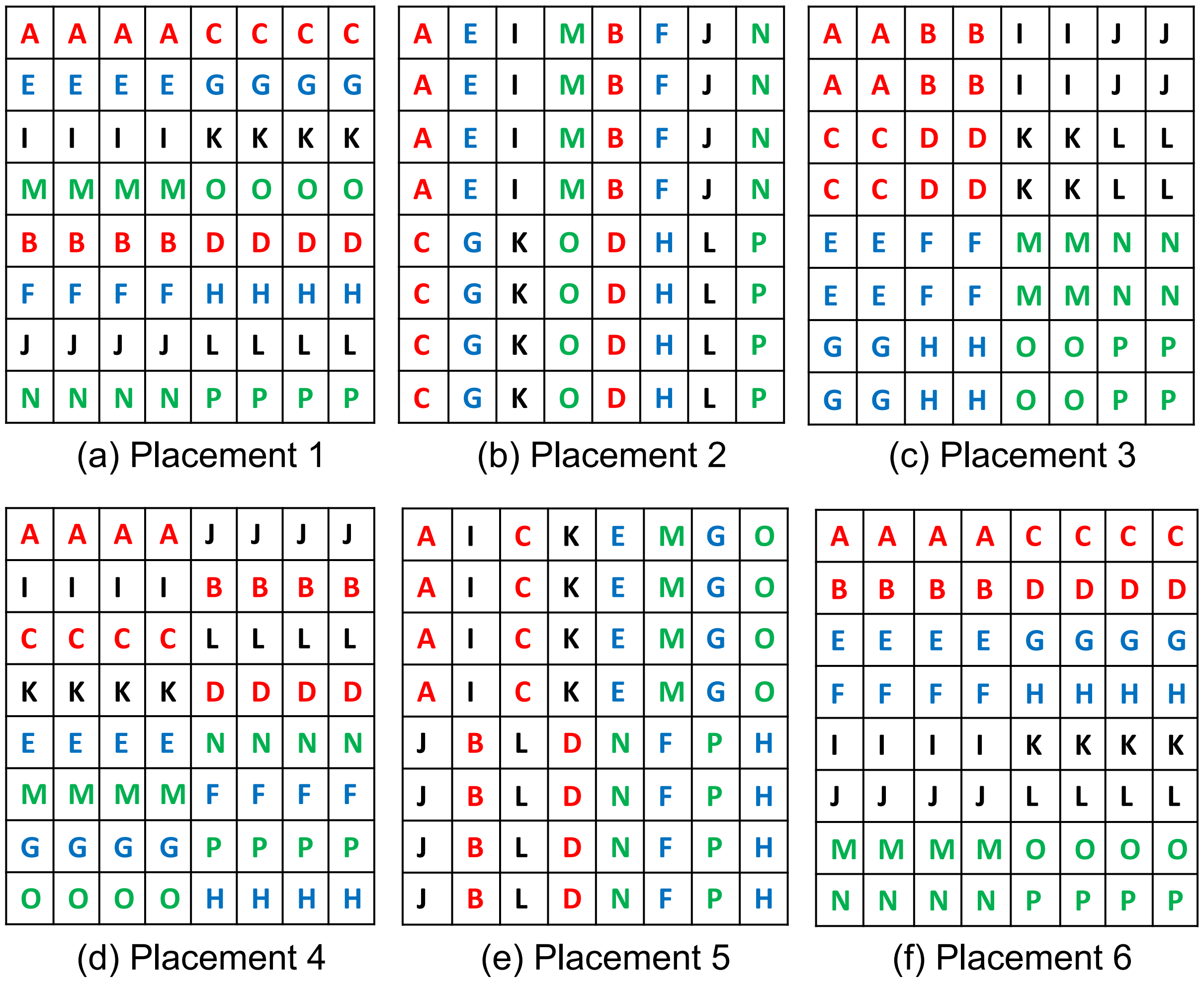


Figure 4.5 : The DMM traffic pattern.

rack traffic for multicast but all shift traffics are cross-rack, 2) Figure 4.6(b): places them column-wise zero cross-rack traffic for shift but all multicast traffic are cross-rack, 3) Figure 4.6(c): places the processes in a mixed manner, considering both multicast and shift traffic across racks, 4) Figure 4.6(d): places the processes in a mixed manner (some traffic are cross-rack but not all) for multicast but all shift traffics are cross-rack, 5) Figure 4.6(e): places the processes in a mixed manner (some traffic are cross-rack but not all) for shift but all multicast traffic are cross-rack, and 6) Figure 4.6(f): places the processes in a mixed manner considering shift but places them row-wise considering multicast (zero cross-rack traffic).

We ran each combination of matrix sizes, networks (RDC, static 4:1 oversubscribed network, NBLK network) 10 times and obtained their average running time. Figure 4.10 shows that RDC improves the shift time for placement 1 (Figure 4.6). For the matrix size $96k \times 96k$, RDC improves the overall communication time 3.9× compared to a static 4:1 oversubscribed network, achieving almost the same performance as NBLK network. Figure 4.11 shows that RDC improves the broadcast/ multicast time for placement 2 (Figure 4.6). For the matrix size $96k \times 96k$, RDC improves the overall communication time 2.3× compared to a static 4:1 oversubscribed network, achieving almost the same performance as NBLK network. Figure 4.12 shows that RDC improves the broadcast/



A	A	A	A	C	C	C	C
E	E	E	E	G	G	G	G
I	I	I	I	K	K	K	K
M	M	M	M	O	O	O	O
B	B	B	B	D	D	D	D
F	F	F	F	H	H	H	H
J	J	J	J	L	L	L	L
N	N	N	N	P	P	P	P

(a) Placement 1

A	E	I	M	B	F	J	N
A	E	I	M	B	F	J	N
A	E	I	M	B	F	J	N
A	E	I	M	B	F	J	N
C	G	K	O	D	H	L	P
C	G	K	O	D	H	L	P
C	G	K	O	D	H	L	P
C	G	K	O	D	H	L	P

(b) Placement 2

A	A	B	B	I	I	J	J
A	A	B	B	I	I	J	J
C	C	D	D	K	K	L	L
C	C	D	D	K	K	L	L
E	E	F	F	M	M	N	N
E	E	F	F	M	M	N	N
G	G	H	H	O	O	P	P
G	G	H	H	O	O	P	P

(c) Placement 3

A	A	A	A	J	J	J	J
I	I	I	I	B	B	B	B
C	C	C	C	L	L	L	L
K	K	K	K	D	D	D	D
E	E	E	E	N	N	N	N
M	M	M	M	F	F	F	F
G	G	G	G	P	P	P	P
O	O	O	O	H	H	H	H

(d) Placement 4

A	I	C	K	E	M	G	O
A	I	C	K	E	M	G	O
A	I	C	K	E	M	G	O
A	I	C	K	E	M	G	O
J	B	L	D	N	F	P	H
J	B	L	D	N	F	P	H
J	B	L	D	N	F	P	H
J	B	L	D	N	F	P	H

(e) Placement 5

A	A	A	A	C	C	C	C
B	B	B	B	D	D	D	D
E	E	E	E	G	G	G	G
F	F	F	F	H	H	H	H
I	I	I	I	K	K	K	K
J	J	J	J	L	L	L	L
M	M	M	M	O	O	O	O
N	N	N	N	P	P	P	P

(f) Placement 6

Figure 4.6 : Six different placements for DMM. A-P represent 16 servers and A-D, E-H, I-L, M-P belong to four physical racks separately.

multicast time for placement 3 (Figure 4.6). For the matrix size $96k \times 96k$, RDC improves the overall communication time 1.6× compared to a static 4:1 oversubscribed network, achieving almost the same performance as NBLK network.

Since the applications have changing traffic patterns, no static process placement is consistently optimal. Out of the three placements, placement 3 jointly minimizes the cross rack traffic for both communication patterns in DMM, outperforming the other two strategies. Even for optimal placement, RDC can further improve the performance by dynamically optimizing the topology at runtime.

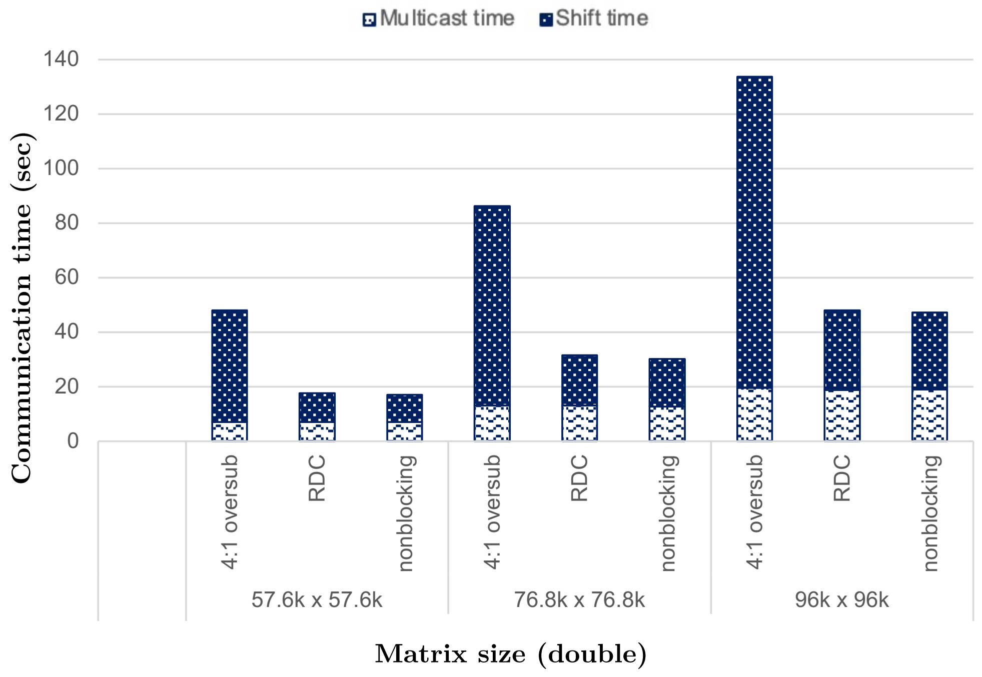


Figure 4.7 : DMM average shift time and multicast time for placement 1.

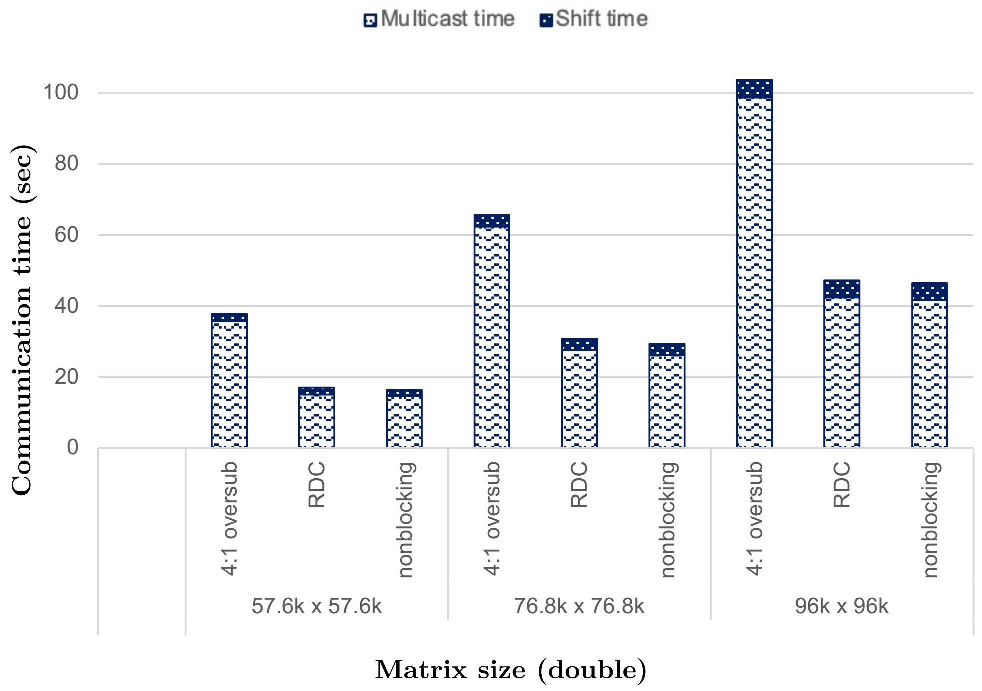


Figure 4.8 : DMM average shift time and multicast time for placement 2.

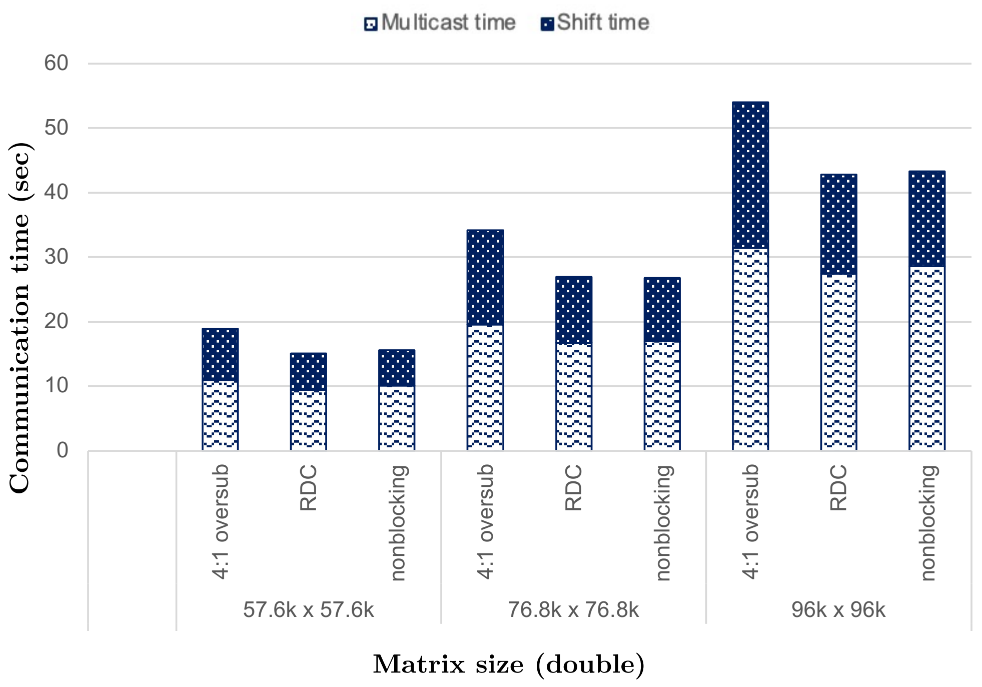


Figure 4.9 : DMM average shift time and multicast time for placement 3.

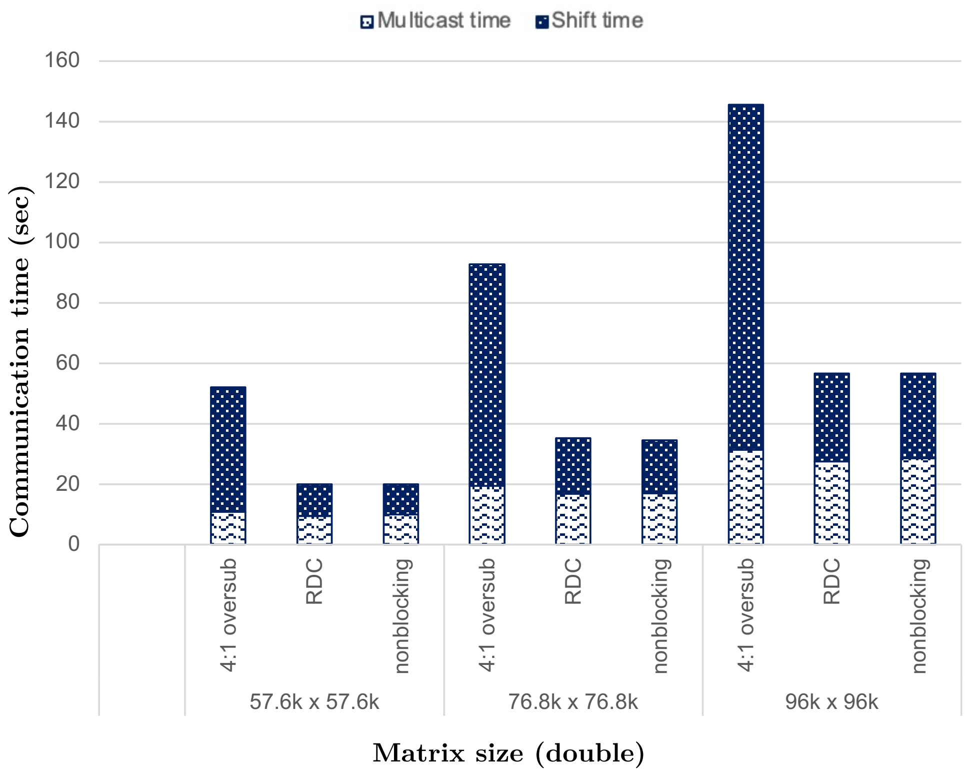


Figure 4.10 : DMM average shift time and multicast time for placement 4.

4.3 Limitations

Understanding communication patterns and how they interact with the network is complex for users. The application user must know the DCN architecture and topology

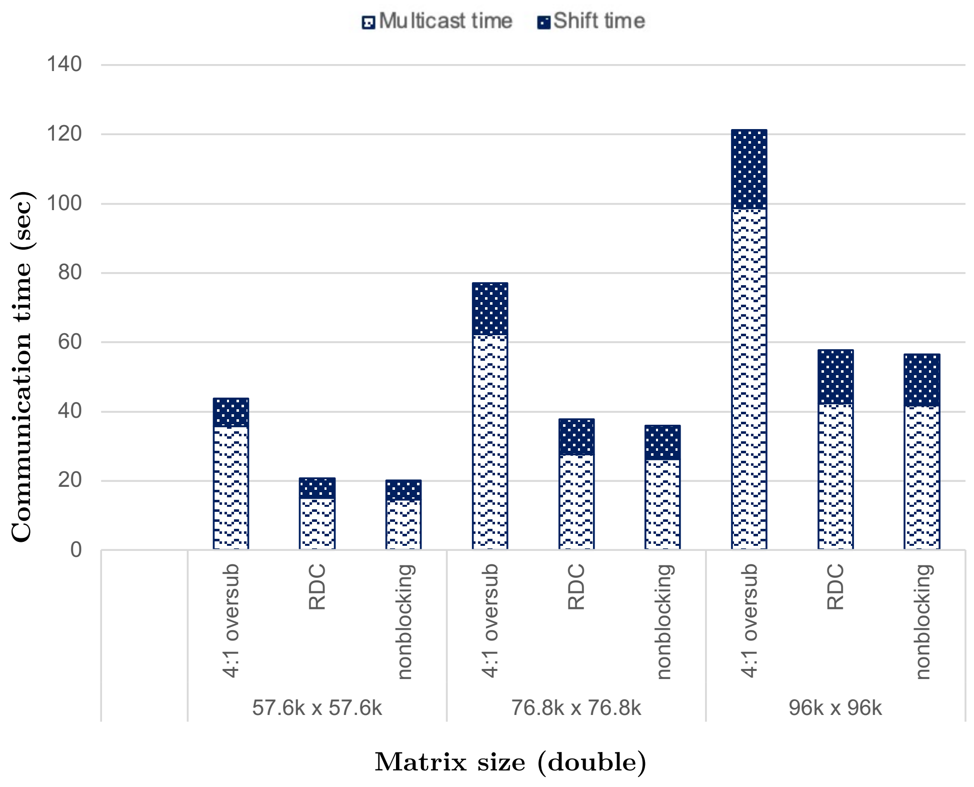


Figure 4.11 : DMM average shift time and multicast time for placement 5.

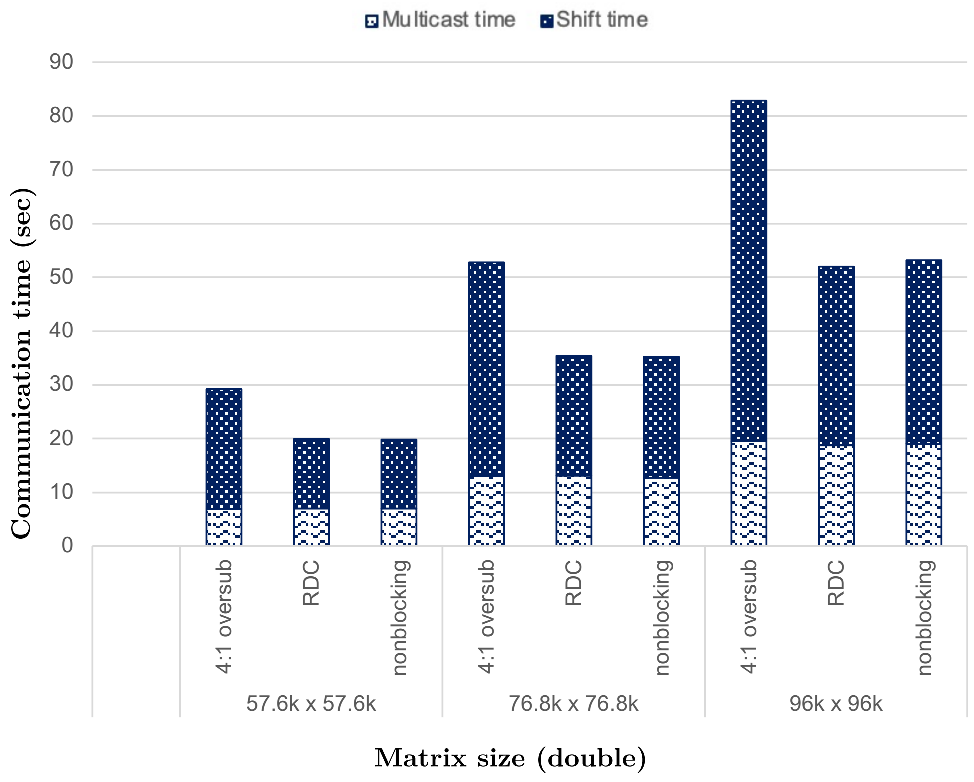


Figure 4.12 : DMM average shift time and multicast time for placement 6.

and know the application's different communication patterns. Based on this information and understanding, the user should augment the application source code with the required code for dynamically reconfiguring the network before each communication pattern changes. Pushing all this load to the application users is inconvenient and unreasonable. Therefore, we cannot expect the users to adopt this solution.

4.4 Summary

RDC, a "rackless" pod-centric DCN architecture, breaks the traditional rack boundaries in a pod. It creates the illusion that servers can move freely among edge switches in response to traffic pattern changes. Rather than optimizing the workloads based on the topology, RDC optimizes the topology to suit the changing workloads. RDC inserts circuit switches between the edge switches and the servers and reconfigures the circuits on demand to form different connectivity patterns. Our experimental results on intra-pod localization show that RDC can decrease communication time considerably for real-world applications. Although this solution can optimize both point-to-point and group communications, the high offline effort expected from the application users makes this solution not realistic.

Chapter 5

Automated Optimal Server Placement Detection

When application communication patterns are irregular or oscillating, communication-aware server placement in Data Center Networks (DCNs) can be critical for overall runtime. Understanding program communication patterns can be valuable and can let us leverage reconfigurable networks for improved server placement as seen in Chapter 4. However, it is limiting to require the user to have a deep and detailed understanding of their application's communication patterns and find a better server placement in the data center to take advantage of reconfigurable networks. Note that the term "communication pattern" here means the processes communicating in a program stage.

In this chapter, we propose the idea of using static program analysis techniques for automated communication pattern identification and automated optimal server placement identification in the DCN for each communication pattern in an application. Static analysis generally begins with control flow analysis—analyzing the code's intermediate representation (IR) form to understand the control flow between operations. The result of the control flow analysis is a control flow graph (CFG). Next, compilers analyze the details of how values flow through the code. Data flow analysis is the classic technique for compile time static program analysis. It allows the compiler to reason about the runtime flow of values in the program [126]. It represents facts about runtime behavior, describes the effect of executing each block on sets of facts, and propagates these facts around the CFG.

Our prototype uses this idea to augment the application source code with the code to reconfigure the network after each communication pattern change in the program. Our proposed solution bridges the gap between reconfigurable networks and users. Using our static analysis-based approach makes detecting oscillating communication patterns, finding the optimal server placement and annotating the program with reconfiguration codes easier and reduces the application user's burden.

In Section 5.1 we design techniques to find the optimal server placement for MPI collective routines (Section 5.1.3) and point-to-point communications (Section 5.1.4). In Section 5.2 we present some of the applications that can potentially benefit from our solution. In Section 5.3 we discuss the implementation details, describe our empirical study in Section 5.4 and conclude with a summary of strengths and weaknesses (Section 5.6).

5.1 Method

MPI communication operations are categorized into two groups: point-to-point communication and collective communication. The primary MPI communication mechanism is the point-to-point sending and receiving of messages by pairs of processes. Collective operations are used to exchange information among a group of processes. Although messages are sent between MPI processes (ranks), ultimately, packets are sent between physical servers in the network. Therefore, we need to know the mapping of processes to servers (logical to physical mapping). In subsection 5.1.1, we show how we obtain the processes to servers mapping.

Our communication pattern detection and optimal server placement identification methods depend on the MPI communication function category. We present our solution for each category in subsection 5.1.3 and subsection 5.1.4. The basic compile time analysis techniques that both solutions rely on are presented in subsection 5.1.2.

After detecting the communication pattern and finding the optimal server placement, we need to reconfigure the network.

5.1.1 Process to Server Mapping

We create a mapping from the processes to the servers by parsing the rankfile. A rankfile is a text file passed to *mpirun* at runtime. It specifies how individual processes should be mapped to servers and to which core they should be bound. Each line of a rankfile

```
rank <N>=<hostname> slot=<slot list>
```

Figure 5.1 : Rankfile format.

specifies the location of one process (Figure 5.1). Each process's rank N refers to its rank in MPI_COMM_WORLD. The *hostname* is the IP address of the server to which the process with that rank is assigned. The slot refers to the core to which the process will be bound. Figure 5.2 shows an example of a rankfile text file and its graphical representation for 4 nodes/servers and 16 processes. In this setting, 4 consecutive processes/ranks will be bounded to 4 cores of a server. Rank 0 to 3 are placed on core 1 to 4 of node1, rank 4 to 7 are placed on core 1 to 4 of node2, and so on. Figure 5.3 shows a second example of a rankfile text file and its graphic representation for 4 nodes and 16 processes. Each process/rank are assigned to nodes in a cyclic fashion. Rank 0 is placed on slot 1 of node1, rank 1 is placed on slot 1 of node2, rank 2 is placed on slot 1 of node3, rank 3 is placed on slot 1 of node4, rank 4 is placed on slot 2 of node1, so on. Rankfile gives the user the ability to statically hand tune the placement once before runtime.

5.1.2 Compile-time Analysis and Transformation Techniques

The first compile time analysis that our solutions rely on is the usage-definition (use-def) analysis which provides a data structure that consists of the uses of a variable X, and all the definitions of variable X that can reach that use without any other intervening definitions (Definition 5.1.2). This analysis is built based on Reaching Definition (RD) analysis (Definition 5.1.1). RD is one of the classic data flow analysis problems. Figure 5.4 shows RD with a simple example. The reaching definitions to the first node are empty. However, after that statement is executed, we have d_1, which reaches the second node. If the if condition holds then both d_1 and d_2 can reach the third node because both d_1 and d_2 reach the second node.

Definition 5.1.1 [Reaching Definitions] A ***definition*** *of a variable x is a statement that*

```
rank 0=node1 slot=1
rank 1=node1 slot=2
rank 2=node1 slot=3
rank 3=node1 slot=4
rank 4=node2 slot=1
rank 5=node2 slot=2
rank 6=node2 slot=3
rank 7=node2 slot=4
rank 8=node3 slot=1
rank 9=node3 slot=2
rank 10=node3 slot=3
rank 11=node3 slot=4
rank 12=node4 slot=1
rank 13=node4 slot=2
rank 14=node4 slot=3
rank 15=node4 slot=4
```

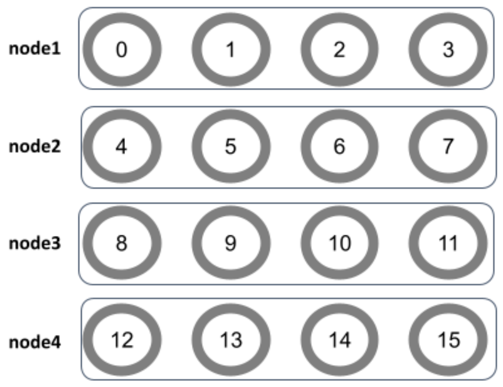


Figure 5.2 : Example #1: Consecutive process rank to node/server assignment.

```
rank 0=node1 slot=1

rank 1=node2 slot=1

rank 2=node3 slot=1

rank 3=node4 slot=1

rank 4=node1 slot=2

rank 5=node2 slot=2

rank 6=node3 slot=2

rank 7=node4 slot=2

rank 8=node1 slot=3

rank 9=node2 slot=3

rank 10=node3 slot=3

rank 11=node4 slot=3

rank 12=node1 slot=4

rank 13=node2 slot=4

rank 14=node3 slot=4

rank 15=node4 slot=4
```

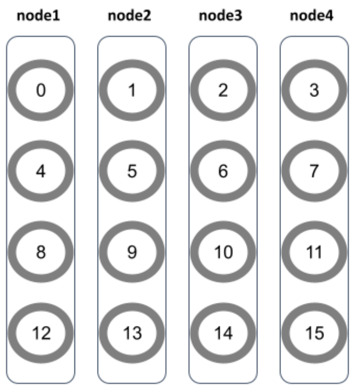


Figure 5.3 : Example #2: Iterative process rank to node/server assignment.

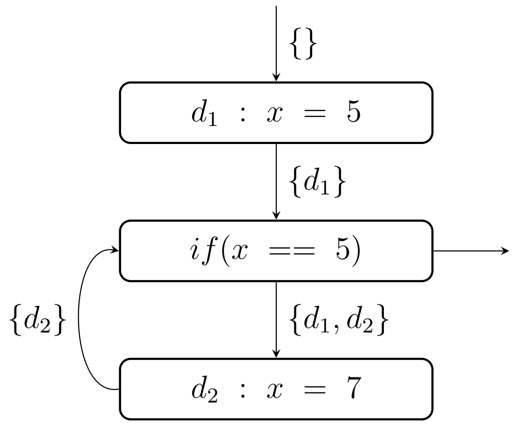


Figure 5.4 : An example of reaching definition.

may modify the value of variable x. A ***definition*** *of a variable x at node k* ***reaches*** *node n if there is a path from k to n along which x is not defined. Note that there is no need to have a use or definition of variable x in node n in order for the definition to reach n.*

Definition 5.1.2 [Use-def chain] It is a chain that links each use, U, of variable x to the definitions, D that can reach that use without any other intervening definitions.

Definition 5.1.3 (Def-use chain) It is a chain that links each definition, D, of variable x to those uses, U that the definition can reach.

The second compile-time analysis and transformation that our solutions rely on is program slicing. Program slicing is a technique for simplifying programs by focusing on selected aspects of program semantics. It computes the program statements (a program slice) that affect the values at a program point, which we are interested in. We find the program skeleton, which is “an abstracted program that is derived from a more extensive program where the source code/statements that are determined to be irrelevant are removed for the purposes of the skeleton” [127, 128]. We use the extractMPISkeleton program analysis module of the ROSE compiler, which follows an iterative process and performs static slicing of the code with the aid of annotations.

The skeleton generator module receives as input the AST generated by the ROSE compiler and the constraints/slicing criterion provided by the user. Constraints are not

```
(API_FUNCTION_NAME ARGUMENT_COUNT
      (dep-type argA ..)
      (dep-type argB ..)
       ...
)
```

Figure 5.5 : API function specification format.

hard-coded in the tool because they depend on the performance dimension that the user wants the skeleton to probe. The criteria is expressed in the form of an API specification file that contains information about the API functions that should be preserved. The module skeletonizes programs relative to the API specifications. The functions that are part of the API are preserved in the skeleton, and further code is preserved based on their dependencies.

We specify each function that we would like to be preserved by the function name, argument count, and a list of dependency types/roles for each argument by position. The format to specify a function in the API specification file is as shown in Figure 5.5. $API\_FUNCTION\_NAME$ represents the function name, the number of arguments is represented by $ARGUMENT\_COUNT$, followed by a list of dependency types *dep-type* for each argument arg. The example in Figure 5.6 shows a small portion of the API specification file for MPI API.

The *api-spec* tag labels each API by name. In the example in Figure 5.6 the API name is MPI. The *dep-types* tag contains a list of dependency types. This list allows analyzing the roles of arguments to API functions and categorizing the code based on their impact on the API functions. The names of dependency types are user-defined. For example, MPI calls usually have arguments related to the payload and some other arguments related to the message passing topology. MPI_Send has six arguments; the first three are related to "payload", and the fourth is "topology". Since the fifth and sixth arguments are not specified, they take the default dependency type "other". A simple

```
(api-spec MPI
  ( dep-types payload topology tag other )
  ( default_deptype other )
  (
    (MPI_Init 2 )
    (MPI_Finalize 0 )
    (MPI_Abort 2 )

    (MPI_Comm_rank 2 (topology 1) )
    (MPI_Comm_size 2 (topology 1) )
    (MPI_Comm_split 4 )

    (MPI_Send 6 (payload 0 1 2)
                (topology 3)
                (tag 4)
    )
  )
)
```

Figure 5.6 : MPI API specification example.

```
(api-spec-collection
    (include-api "mpi_api.spec" (omit-deps payload))
    (include-api "stdio_input.spec" (omit-deps buffer))
)
```

Figure 5.7 : An example of API specifications collection.

example is to preserve all code that influences the message passing topology and replace code that initializes buffers with a constant value. The API specification collection *api-spec-collection* is specified by a set of API of specifications (Figure 5.7), and the skeleton generator uses it to know what set of APIs to skeletonize relative to and how to do so. A specification of the dependency type to be eliminated is provided for each API. The API specification collection in Figure 5.7 instructs the tool to include API specifications for MPI and a subset of C STDIO functions and to eliminate the code that relates to the computation of payload data for MPI calls and buffer management code for the STDIO API. The user can have relatively fine control over what is removed at the API level.

Given the API specifications and the API specification collection, the skeleton generator uses dependency analysis provided by ROSE to label statements within the program based on their role concerning the API. The labels are used to prune the program AST (Abstract Syntax Tree) before generating source code representing the reduced skeleton.

The AST is one of the most commonly used Internal Representations (IRs) by compilers to transmit the program between compilation phases [129]. It mimics the form of the program at different points in translation. The AST includes the important syntactic structure of the program, but as opposed to the parse tree, it omits any nonterminals that are not needed to understand that structure. It is very close to the source-language syntax, therefore, retains concise representations for most of the abstractions in the source language. This IR form is mostly used for analyses and transformations that are tied to source code structure. Figure 5.8 shows the AST for $a = 2 * b + c$.

The skeleton generator adds Static Single Assignment (SSA) property to the AST.

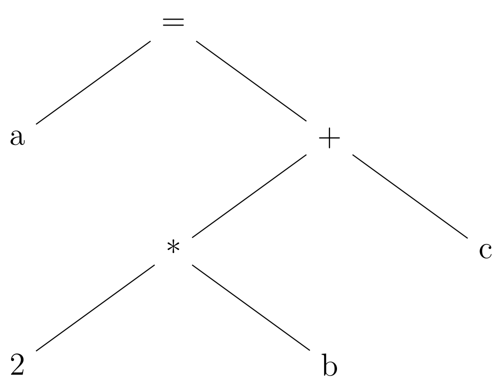
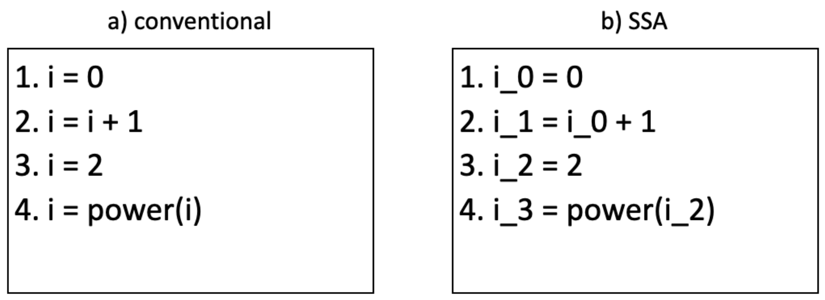


Figure 5.8 : AST for $a = 2 * b + c$.



a) conventional

```
1. i = 0
2. i = i + 1
3. i = 2
4. i = power(i)
```

b) SSA

```
1. i_0 = 0
2. i_1 = i_0 + 1
3. i_2 = 2
4. i_3 = power(i_2)
```

Figure 5.9 : The SSA form of the program section in (a) is shown in (b).

SSA is a property that requires each variable to be assigned exactly once. In other words, every assignment to a variable creates a new version of the variable. Figure 5.9 shows a simple example of a conventional program section and its SSA form. The SSA form simplifies the process of def-use analysis because determining all uses of a variable from its definition is done without any other intervening definitions.

A set of variables corresponding to the contents of expressions that form the function call arguments are obtained for each API function selected to be included in the skeleton. Each argument is labeled with the dependency type defined with the API specification. For example, the MPI API dependency types defined in the example were “topology” or “payload”.

The def-use graph for these expressions is traversed in the skeletonizer. Each program element leading to the API call is labeled with the dependency type from the API specification.

The code transformation is performed after the AST nodes are annotated/labeled with their role in the skeleton. If a statement is not in the dependency chain of the API functions, then the statement is removed. If a statement is in the dependency chain and the complete set of roles associated with it do not appear in the *omit-deps* parameter in the API specification parameter file, the statement is preserved. After the AST transformation, the final code is generated in the language of the original program (in our case, c++).

5.1.3 Collective Communication and Computation

Collective operations provide a higher-level approach to organizing a parallel program. Each process involved in the collective group executes the same communication function. Communication and computation are coordinated among a group of processes in a communicator. A communicator represents a logical group of MPI processes.

In this subsection, we are interested in automatically identifying collective communication patterns to find the optimal server placement in our reconfigurable DCN for each communication function call and augment the application source code with the reconfiguration code to reconfigure the network after each pattern change. The first step toward our goal is to identify processes in a communicator (Section 5.1.3.1). Our first attempt to automatically identify collective communicators using static analysis is presented in Section 5.1.3.1.1 and our second attempt/final solution is presented in Section 5.1.3.1.2. In Section 5.1.3.2, we present our algorithm that tailors the server placement identified in Section 5.1.3.1.2 to specific communication algorithms to find the optimal server placement with the least number of cross-rack communications and least number of circuit reconfiguration.

5.1.3.1 Communication Pattern Detection for Collectives

MPI has five basic types of collective data movement functions: broadcast, scatter, gather, all-gather, and all-to-all. Note, MPI has a set of collective computation functions that combine communication with computation: reduce, scan, exscan, and reduce-scatter. Figure 5.10 and Figure 5.11 show these collective functions with their parameters. These functions all have a parameter of type MPI_Comm in common that is important for identifying communication patterns. MPI_Comm is the primary communicator object type used by MPI to determine which processes are involved in a communication. The default communicator provided by MPI is MPI_COMM_WORLD. The MPI_COMM_WORLD contains all processes in program execution. If the parameter *comm* is MPI_COMM_WORLD, then we know the processes involved in the communication at that point of the program, and we only need to find the optimal server placement for the specific communication algorithm used (Section 5.1.3.2). However, it is not always this simple; complex applications use the MPI_Comm_split function to create new communicators. MPI_Comm_split is a powerful mechanism for dividing a single communicating group of processes into an arbitrary number of subgroups (Figure 5.12) [130]. This function splits the original communicator *comm* to a group of sub-communicators *newcomm* based on the input values *color* and *key*. A new communicator is created on each process, but it does not mean that the process leaves the original communicator.

Let us look at a program with 8 processes to understand how the split function works. All the 8 processes are by default grouped inside MPI_COMM_WORLD as shown in Figure 5.13. Now, we want to make two new communicator groups. One communicator will group processes 0, 1, and 2, and the other communicator will group processes 5, 6, and 7. Processes 3 and 4 will only be part of MPI_COMM_WORLD. Note that after a split, all the processes are still part of MPI_COMM_WORLD. As pointed before, the split function distributes all of the processes of an existing communicator to new communicators based on their *color*. *color* is the value assigned to each process that determines which communicator a process will end in after splitting. To split processes 0, 1, and 2 to one communicator and processes 5, 6, and 7 to another communicator, we assign processes 0, 1, and 2 the *color* 0 and assign processes 5, 6, and 7 the *color* 1

```
1. MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

2. MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
        int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

3. MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
        int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

4. MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
        int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

5. MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
        int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

Figure 5.10 : MPI collective data movement functions.

(Figure 5.14). Since we do not assign a color to processes 3 and 4, MPI would not place them in any new communicators. Figure 5.15 shows the two new communicators created by the split.

To detect the communication group for every collective function, we want to analyze the *MPI_Comm_split* function that generates the communication group/communicator used by the collective function in the program.

First, we do use-def analysis to find which *MPI_Comm_split* defines the communication group used by the collective function. In Figure 5.16 we show a simple example to demonstrate the use-def annotated CFG of a program containing two split function calls and a broadcast function call. The *MPI_Bcast* function in node 6 depends on the variable *newcomm*. The split function in node 4 reassigns *newcomm* after node 2. The definition of *newcomm* that reaches node 6 is coming from node 4.

To simplify the program and ease the analysis, we find the skeleton of the program that the *MPI_Comm_split* and the MPI collectives parameters of interest depend on by using the extract MPI skeleton program analysis module. The API function that we want to preserve for our purpose is the selected *MPI_Comm_split* function. Figure 5.12, shows that *MPI_Comm_split* has four parameters. We create a new API specification

```
1. MPI_Reduce( void* send_data, void* recv_data, int count, MPI_Datatype datatype,
        MPI_Op op, int root, MPI_Comm comm)

2. MPI_Scan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
        MPI_Op op, MPI_Comm comm)

3. MPI_Exscan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
        MPI_Op op, MPI_Comm comm)

4. MPI_Reduce_scatter(const void *sendbuf, void *recvbuf, const int recvcounts[],
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Figure 5.11 : MPI collective computation functions.

```
MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

Figure 5.12 : Splitting a communicator into a group of sub-communicators.

for MPI. There we label the dependency type of the second and third parameters (*color* and *key*) as “topology" shown in Figure 5.17. We pass our MPI API specifications and the program source code to the skeleton generator. We want the skeleton generator to skeletonize relative to the API function specifications that we pass and preserve all code that influences the message passing topology.

An example of a small portion of an MPI program is provided in Figure 5.18. The def-use/use-def chain for relating uses of variables to their defining statements is shown in Figure 5.19. Our focus is on the MPI_Comm_split. The blue arrows point from the uses to the definitions (Use-def chain), and the orange arrows point from the definitions to the uses (Def-use chain). For the MPI_Comm_split function on line 15, the variables corresponding to the contents of expressions that form the arguments are obtained. By traversing the def-use chain, the program elements relating to the expressions *row* and $myPE$ are labeled as a dependency with the dependency type “topology” which is inherited from the MPI API function specification (Figure 5.17).

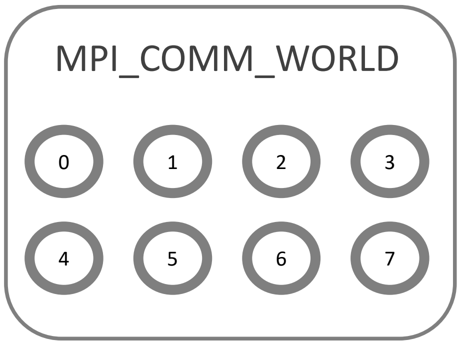


Figure 5.13 : MPI_COMM_WORLD for a program that has 8 processes.

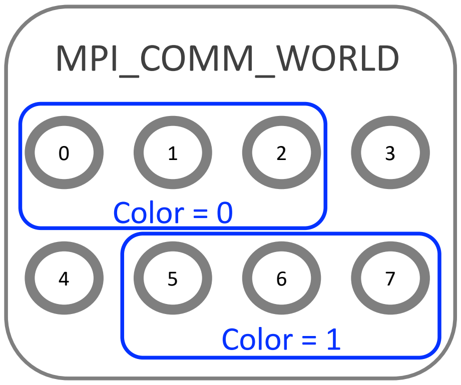


Figure 5.14 : The color assigned to processes to split the original communicator.

The arguments row and $myPE$ are defined on lines 12 and 3 respectively, so these lines of code are preserved. The expression on line 12 depends on $myPE$ and $maxIndex$ therefore, lines 3 and 11 are preserved (following the blue arrow). The variable $maxIndex$ defined on line 11 has a blue arrow to line 2 ($size$ is defined on line 2) therefore the expression on line 2 is preserved as well. All expressions defined from line 5 to 11 and line 13 to 15 are removed. The skeleton we obtain contains the expressions that form the MPI function call arguments. Figure 5.20 shows the skeleton generated for the input program in Figure 5.18.

To statically analyze MPI_Comm_split in the generated skeleton without analyzing the MPI implementation of the split function, we propose to emulate the MPI_Comm_split to find the communication group at compile time. We also comment out MPI_Comm_size and MPI_Comm_rank functions. These two functions return the size of the group associated with MPI_COMM_WORLD and the rank of the process executing the code. We define new variables holding this information based on the rankfile. Before attempting

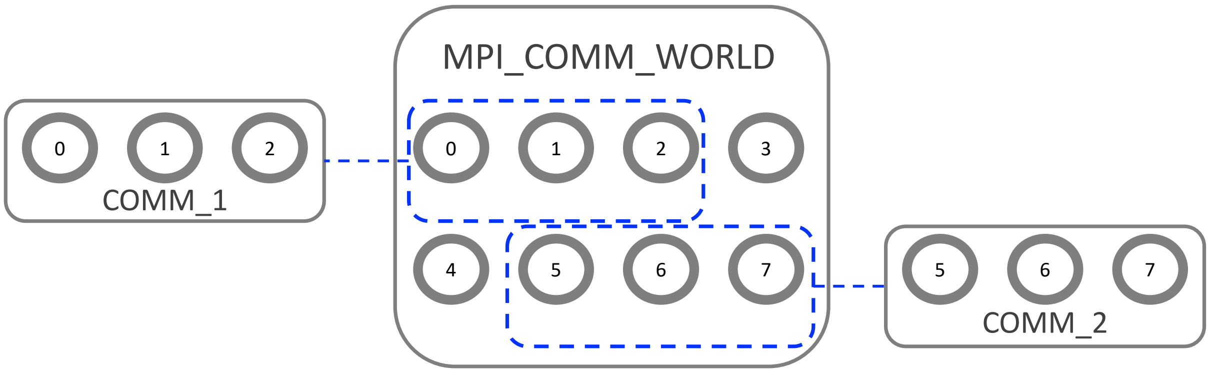


Figure 5.15 : The two new communicator generated based on the colors assigned to processes.

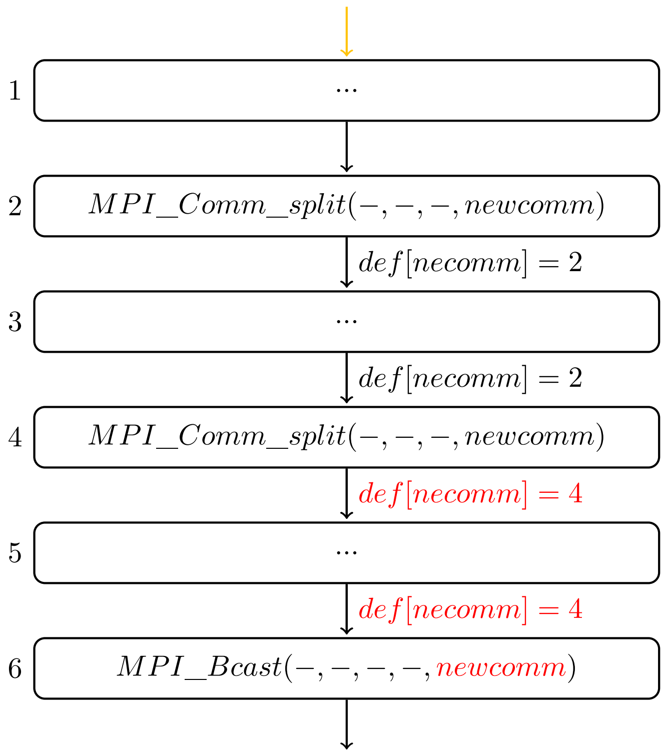

1
...
2
$MPI\_Comm\_split(-,-,-,newcomm)$
$def[necomm] = 2$
3
...
$def[necomm] = 2$
4
$MPI\_Comm\_split(-,-,-,newcomm)$
$def[necomm] = 4$
5
...
$def[necomm] = 4$
6
$MPI\_Bcast(-,-,-,-,newcomm)$

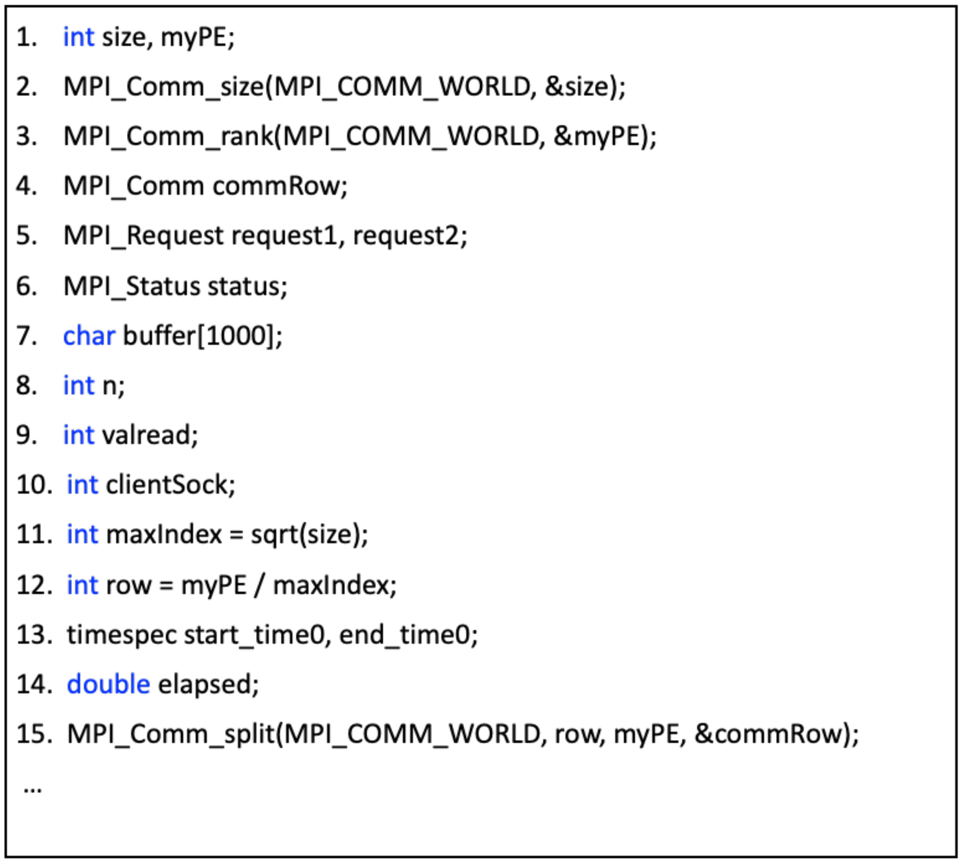
Figure 5.16 : Use-def Control Flow Graph.

to emulate split, we need to understand how it functions.

The MPI_Comm_split function has four parameters (Figure 5.12). The first parameter, $comm$, is the communicator used as the basis for the subcommunicators. In the example in Figure 5.20, $comm$ is the MPI_COMM_WORLD. The second parameter,

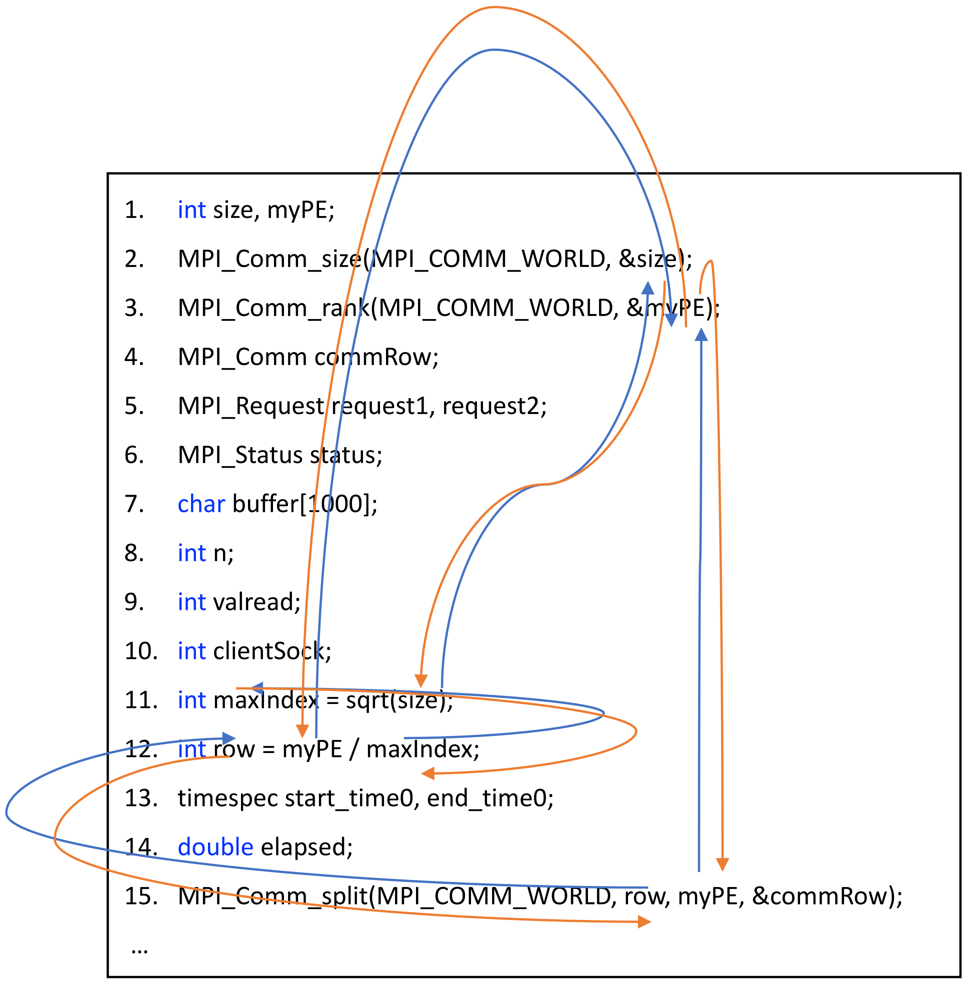
```
(MPI_Comm_split 4 (topology 1 2) )
```

Figure 5.17 : *MPI_Comm_split* API specification.



```
int size, myPE;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &myPE);
MPI_Comm commRow;
MPI_Request request1, request2;
MPI_Status status;
char buffer[1000];
int n;
int valread;
int clientSock;
int maxIndex = sqrt(size);
int row = myPE / maxIndex;
timespec start_time0, end_time0;
double elapsed;
MPI_Comm_split(MPI_COMM_WORLD, row, myPE, &commRow);
...
```

Figure 5.18 : Skeleton generator input program.

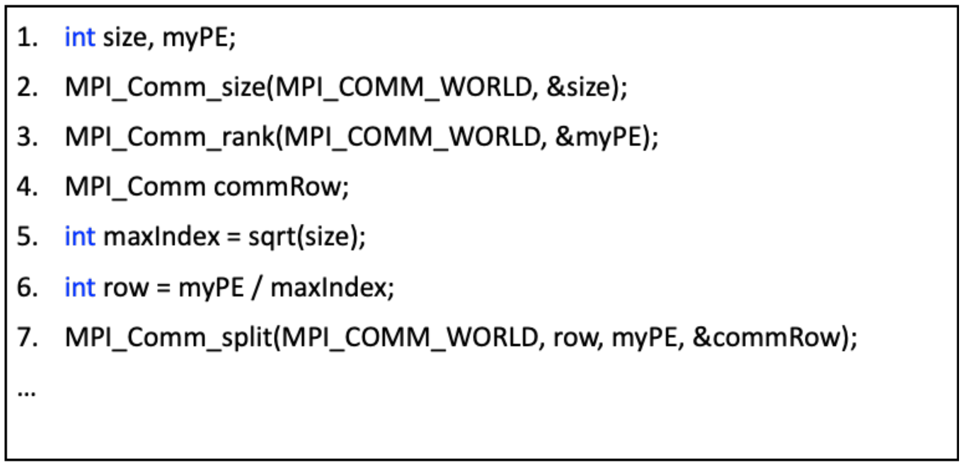


```
int size, myPE;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &myPE);
MPI_Comm commRow;
MPI_Request request1, request2;
MPI_Status status;
char buffer[1000];
int n;
int valread;
int clientSock;
int maxIndex = sqrt(size);
int row = myPE / maxIndex;
timespec start_time0, end_time0;
double elapsed;
MPI_Comm_split(MPI_COMM_WORLD, row, myPE, &commRow);
…
```

Figure 5.19 : Def-use chain.

color, determines which communicator each process will belong to (each process gets a color). All processes with the same color value are assigned to the same communicator. In this example, the *row* value determines the new communicator allocation. The third parameter, *key*, determines each process's ordering or rank in the new communicator. The process with the lowest *key* value will get rank 0, and so on. In our example, $myPE$, the process rank in the original communicator, is passed as *key*. Therefore, the processes will retain the same order in the new communicator. The final parameter, *newcomm* is the new communicator returned by the function. The *newcomm* in our example is a pointer to the new communicator, which is *commRow*.

The *color* and *key* could be of the following type: 1) a constant value, 2) a macro, 3) a function, 4) a mathematical expression or 5) a variable. All macros are preprocessed and



```
int size, myPE;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &myPE);
MPI_Comm commRow;
int maxIndex = sqrt(size);
int row = myPE / maxIndex;
MPI_Comm_split(MPI_COMM_WORLD, row, myPE, &commRow);
...
```

Figure 5.20 : Skeleton generator output program.

replaced with their corresponding values. If *color* and *key* are of type 3 or 4, their definition depends on the process rank. If they are of the type variable, then a constant value, a macro, a function, a mathematical expression, or another variable can be assigned. If *color* and *key* are not type constant, the definition depends on the process rank. We run use-def analysis and query the use-def analysis graph to find the reaching definition of *color* and *key* to this specific MPI_Comm_split in the program.

Algorithm 1 shows the Split emulation pseudocode. We want to calculate the *color* and *key* for every process in the program and find the processes in subcommunicators. We replace the rank variable in the definition of *color* and *key* with the variable i. We compute the *color* for all processes included in *comm* by looping on all process ranks in *comm* $(\forall i : 0 \leq i < size(comm))$ (*line* 1 to 4). Then we sort the processes by their *color* so that all processes with the same *color* are next to each other and create the new communicator groups $newcommPrime$ (*line* 5). The $newcommPrime$ is a list of all processes ordered in groups based on their color. To find the order of the processes in their new communicator group, we compute *key* for all the processes included in *comm* by looping on all process ranks in *comm* $(\forall i : 0 \leq i < size(comm))$ and create a mapping from process to key (*line* 6 to 9). We sort the new communicator groups $newcommPrime$ based on the process *key* (*line* 10).

Algorithm 1 Emulate Split

```
for $i \leftarrow 0$ to $size$ do
    $color := colorDef;$
    $processColors[i] := color;$
end for
$newcommPrime :=$ sortByColor($processColors$);
for $i \leftarrow 0$ to $size$ do
    $key := keyDef;$
    $processToKeyMapping[i] := key;$
end for
sortByKey($newcommPrime$, $processToKeyMapping$);
```

We explore two approaches to identify collective communicators using static analysis automatically. Our first approach in analyzing split function emulation is presented in Section 5.1.3.1.1 and our second approach/final solution is presented in Section 5.1.3.1.2.

5.1.3.1.1 First approach to identifying communicators at compile time

To find the communication group at compile-time, we propose to use something similar to constant propagation analysis that we call array constant propagation.

Definition 5.1.4 (Constant propagation) An optimization that discovers, at compile time, expressions that must have known constant values, evaluates them and replaces their run-time evaluation with the appropriate value [129].

Constant propagation determines at every program point the variables that have a constant value. Then it propagates these calculated or determined values to the next reachable point of the program. Constant propagation has been used to turn variable reads and computations into constants.

For example, if we have *int* $x = 2$ in the program. We can replace all occurrences of x with 2 in the program and eliminate *int* $x = 2$. However, for accuracy, all branches of

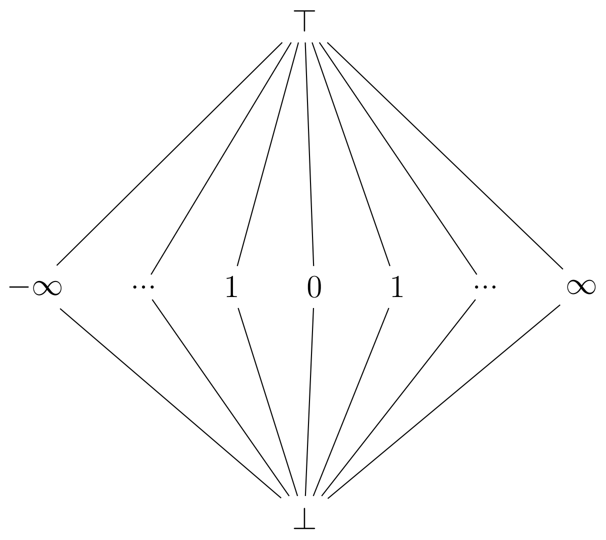


Figure 5.21 : Constant Propagation lattice.

the program and reassignments of a variable must be tracked.

Constant propagation tries to find a mapping between variables and values of $N \cup \{\top, \bot\}$ for every program point/statement. If a variable is mapped to a constant number, that constant number is the variable's value in that point of the program on every execution. If a variable's value in that point of the program is undetermined (It could be constant or not later in the program), it is mapped to $\top$. In other words, no definition of the variable has been seen along any path reaching this point of the program. If the variable's value is not initialized, or the statement is unreachable (constant value cannot be guaranteed), the variable is mapped to $\bot$. In other words, $\bot$ means that the variable can have different values at this point of the program along different paths reaching this point. The lattice shown in Figure 5.21 map each variable V in the program to its constant status ($V \rightarrow N \cup \{\top, \bot\}$). At each point in the program, the transfer function related to that statement is executed on the mapping of the variable from the input lattice to generate the output lattice. For example, if the input mapping of a statement is $[x \rightarrow 0,\ y \rightarrow 1]$, and the statement itself is $x = 2$, then after moving through this statement, the mapping should be $[x \rightarrow 2,\ y \rightarrow 1]$.

Definition 5.1.5 (Lattice) A lattice is a partially ordered set $L(\leq)$ such that for all a, b *in L, a and b have a least upper bound $a \sqcup b$ and a greatest lower bound $a \sqcap b$. This lattice is denoted by $L(\leq, \sqcup, \sqcap)$ [131, 132].*

Definition 5.1.6 (Partially ordered set) A partially ordered set is a set S equipped with a binary relation $\leq$, which is reflexive, transitive, and antisymmetric [131, 132].

- *$c \in S$ is an upper bound of $X \subseteq S$ if and only if $\forall c\prime \in X$, $c\prime \leq c$.*
- *$c \in S$ is a lower bound of $X \subseteq S$ if and only if $\forall c\prime \in X, c \leq c\prime$.*
- *$c \in S$ is the least upper bound of $X \subseteq S$ if and only if $\forall c\prime \in X$, $c\prime \leq c$, and $\forall c\prime\prime \in S$ such that $\forall c\prime \in X$, $c\prime \leq c\prime\prime$, we have $c \leq c\prime\prime$.*

The ordering is not necessarily total; that is, it may not have $a \leq b$ or $b \leq a$. Two elements can be incomparable.

The communicator defined/initialized by the split function remains static if another split function does not reassign it at some point in the program. Our MPI_Comm_split emulation substitutes the MPI communicator with an array. If the elements of this array are constant and can be determined at compile time similar to constant propagation, we can use it to reconfigure the network before each communication pattern.

Our emulation of the MPI_Comm_split function for the example in Figure 5.20 is shown in Listing 5.1. We substitute the line 7 of the skeleton with this code. Only the process with rank 0 will run it. We create an array of size *size* called MPI_COMM_WORLD. The index of the array relates to the rank of the process in the world communicator. We calculate the color for every process (line 4 to 7). The color is assigned to the array element for that process (line 6). We create an array of size *size* called *commRow*. In this array, we order the processes based on their *color*. For example, processes with *color* 0 will be added first to the *commRow* and in the ascending order of their *myPE*. We have an emulation of the split in our code that we can run static analysis on it.

Our array constant propagation algorithm finds a mapping between arrays and values of the lattice for every program point/statement. The Hasse diagram of this lattice

Listing 5.1: *MPI_Comm_split* substitution

```
int row;
int *MPI_COMM_WORLD= create1DArray(size);
int *commRow= create1DArray(size);
for (int i = 0; i < size; i++){
   row = i / maxIndex;
   MPI_COMM_WORLD[i] =  row;
}
//find the number of distinct elements
int res = 1;
for (int i = 1; i < size; i++) {
    int j = 0;
    for (j = 0; j < i; j++)
        if (MPI_COMM_WORLD[i] == MPI_COMM_WORLD[j])
           break;
    if (i == j)
      res++;
}
//sort processes based on color
int index = 0;
for (int i = 0; i < res; i++){
    for (int j = 0; j < size; j++){
        if(i==MPI_COMM_WORLD[j]){
            commRow[index]=j;
            index++;
        }
    }
}
```

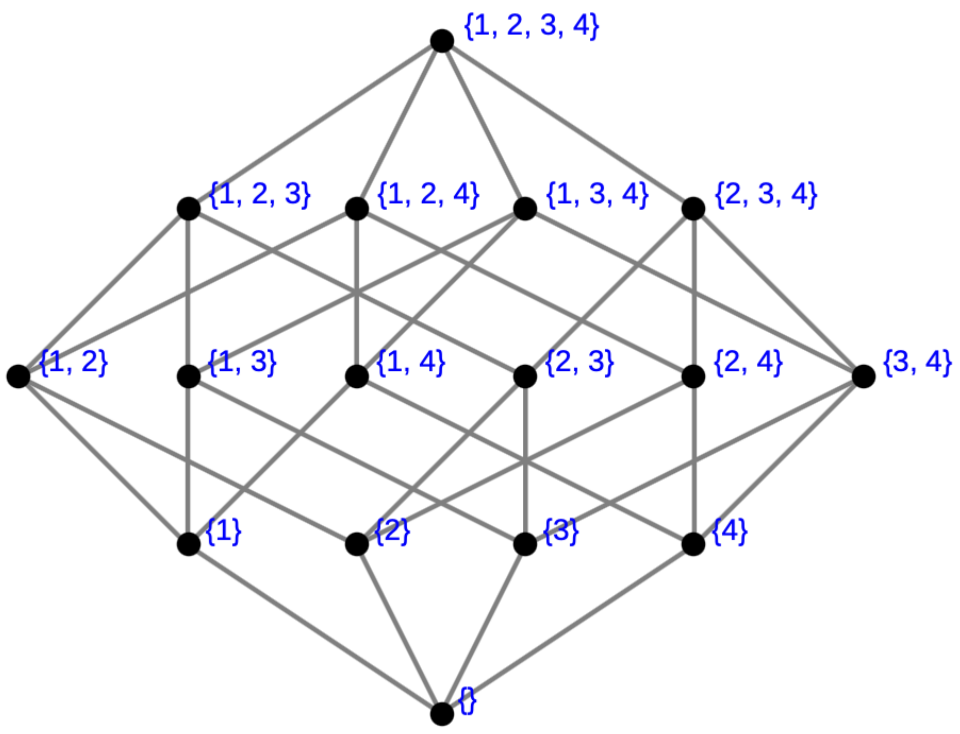


Figure 5.22 : Hasse diagram for power set of 4 processes.

for 4 processes is shown in Figure 5.22. A Hasse diagram represents a finite partially ordered set, in the form of a drawing of its transitive reduction [133]. A concrete lattice element here is a set of processes. The level 1 has the combination of all possible groups containing one process out of *size*. For example, if the World communicator includes 4 processes then at level 1, there are four possibilities $\{0\}$, $\{1\}$, $\{2\}$, and $\{3\}$. The level 2 has the combination of all possible groups containing two processes out of *size*. If the World communicator includes 4 processes then at level 1, there are six possibilities $\{0,1\}$, $\{0,2\}$, $\{0,3\}$, $\{1,2\}$, $\{1,3\}$, $\{2,3\}$. The last level has one concrete element, which is a set of all processes (it equals the World $\{0,1,2,3\}$). Each level of the Hasse diagram lattice includes $\dfrac{size!}{(size - level)! \times level!}$ elements.

However, as the number of processes gets larger the lattice gets much bigger. We can avoid this problem by using bounded regular section analysis [134, 135]. Regular sections describe which elements of an array has been used and which have been defined as a side effect of a function call. A bounded regular section is a regular section with bounds. It

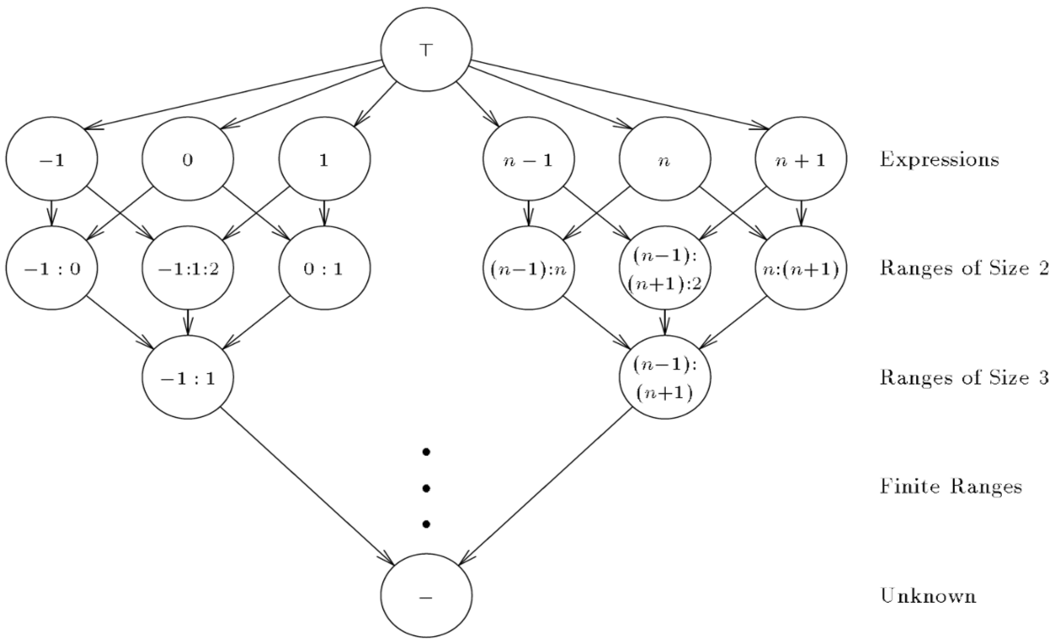


Figure 5.23 : Lattice for regular section subscripts.

```
for (int i=0; i<=6;i++)
        a[i] = b[i*2+1];
```

Figure 5.24 : Example of a loop to demonstrate the bounded regular section.

can be written with triplet notation $[\,l : u : s\,]$. l represents the lower bound, u represents the upper bound and s the stride. It can represent sparse region such as strips and dense region such as rows.

Figure 5.23 shows the subscript lattice. The descriptors for bounded regular sections are vectors of elements from this lattice. The lattice elements include constants, ranges showed with the triplet notation $[l : u : s]$ and $\perp$. Constants represent the value of variables on the entry of the function. Ranges give the constant for the lower bound, upper bound, and stride of a variant subscript. $\perp$ indicates no knowledge of the value. Ranges can be constructed through a sequence of meet operations or directly from the bound of loop induction variable used in the array subscript.

For example, in Figure 5.24 the value of i is represented as $[\,0 : 6 : 1\,]$ and the value of

$i*2+1$ (indices of used elements of array b) is represented as $[2*l+1 : 2*u+1 : 2*s] = [1 : 13 : 2]$.

The bounded regular section analysis algorithm merges the ranges by finding the lowest lower bound and the highest upper bound, then correcting the stride. The result of merging 1, 3 and 5 is the same as merging $[1 : 5 : 4]$ and 3 which is $[1 : 5 : 2]$.

Most MPI programs split communicators with a regular pattern. Based on a formula, the processes are regrouped. We mean that, for example, processes are split into two groups of odd and evens with the formula $x \ mod \ 2$, which the stride is 2. The groups generated from 16 processes are demonstrated as $[0 : 14 : 2]$ and $[1 : 15 : 2]$. The first group consists of all even processes $\{0, 2, 4, 6, 8, 10, 12, 14\}$ and the second group consists of all odd processes $\{1, 3, 5, 7, 9, 11, 13, 15\}$. A more general case would be that all processes with the same value for modular division over a constant number are grouped. Therefore, this solution can accurately find the processes involved in communication groups following a pattern.

However, in an unlikely scenario if the split does not follow a regular pattern then this solution would not be suitable. If a communication group includes process 0, 1, 3, 6, and 9 then the bounded regular section representation of this group is $[0 : 9 : 1]$. Bounded regular section $[0 : 9 : 3]$ does not include process 1 and $[0 : 9 : 1]$ includes extra processes (2, 4, 5, 7, and 8).

Note, after the analysis is over, we bring back/uncomment the *MPI_Comm_split* function to the code and store the result of our analysis in variable that will be used in Section 5.1.3.2. All these actions take place in the middle end of the compiler.

5.1.3.1.2 Second approach to identifying communicators at compile time

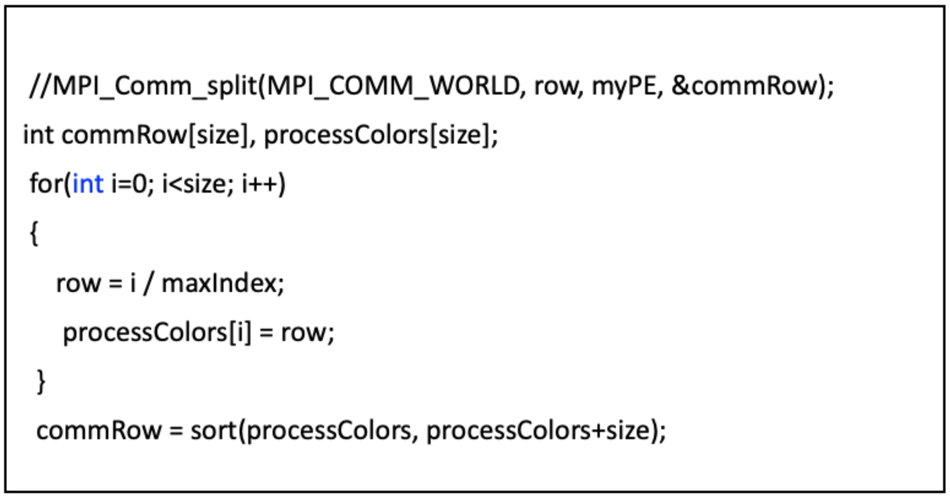
We explore another approach to identifying communicators in this section which is based on commonly used static analysis techniques. After learning which *MPI_Comm_split* defines the collective communication group using def-use analysis, we substitute the code with an emulation of *MPI_Comm_split* to find the communication group. Algorithm 1 shows the new communicator group identification pseudocode (Split emulation) for our second approach. In this solution, instead of analyzing the emulation to find all the elements in new subgroups in the middle-end of the compiler, we first augment the program source code with the emulation and unparse the program. Afterward, we run the output program. The process 0 runs the emulation and obtains the new groups for that setting (a fixed number of processes and rankfile). We only run the emulation once for each runtime setting. We do not need to re-run it for the same setting every time running the MPI program because the subgroups remain the same for each communication pattern.

Figure 5.25 shows the MPI_Comm_split substitution for the skeleton in Figure 5.20. Since the *key* is $myPE$ which is the process rank. We do not need to compute *key* for all processes.

The result of the emulation are stored and are used in Section 5.1.3.2 to find the optimal server placement for communication pattern.

The Algorithm 2 shows the pseudocode of the overall steps we take to identify the processes in new groups/communicators created by a split function. We create a mapping from the processes to the servers by parsing the rankfile (*line* 1 in Algorithm 2).

If *color* and *key* are not of type constant, the definition depends on the process rank. We run use-def analysis and query the use-def analysis graph to find the reaching definition of *color* and *key* to this specific MPI_Comm_split in the program (*line* 3 in Algorithm 2). We want to find the *color* for every process in the program. We replace the rank variable in the definition of *color* and *key* with the variable i (*line* 3 and 4 in Algorithm 2). The *lines* 5 to 14 are part of the split emulation pseudocode shown in Algorithm 1 . We compute the *color* for all processes included in *comm* by looping on all process ranks in *comm* ($\forall i : 0 \leq i < size(comm)$) (*line* 5 to 8). Then we sort the processes by their *color* so that all processes with the same *color* are next to each other and create the new communicator groups *newcommPrime* (*line* 9). The *newcommPrime* is a list of all processes ordered in groups based on their color. To find the order of the processes in the new communicator group, we compute *key* for all the processes included



```
//MPI_Comm_split(MPI_COMM_WORLD, row, myPE, &commRow);
int commRow[size], processColors[size];
for(int i=0; i<size; i++)
{
   row = i / maxIndex;
    processColors[i] = row;
 }
 commRow = sort(processColors, processColors+size);
```

Figure 5.25 : *MPI_Comm_split* substitution for the skeleton in Figure 5.20.

in *comm* by looping on all process ranks in *comm* ($\forall i : 0 \leq i < size(comm)$) and create a mapping from process to key (*line* 10 to 13). We sort the new communicator groups *newcommPrime* based on the process *key* (*line* 14). The last step is to use the mapping from process to server represented by *mapping* to replace the processes with the server node name that they are placed at (*line* 14) and remove redundancies. For example: if the process 0 and 1 are on node1 and process 2 and 3 are on node2, and they all are part of a new communicator group, then without removing the repetitions, the reconfiguration string looks like "node1, node1, node2, node2". Therefore after removing the redundancies, we get the proper server placement order "node1, node2". The code on line 1 and lines 5 to 16 are inserted in the program source code before *MPI_Comm_split*, and the values for the optimal server placement are computed at runtime and passed to the controller to reconfigure the network before the collective communication starts. Note that only rank 0 emulates split.

Note that the resulting server placement is optimal for the default collective communication algorithm (chain algorithm). However, it is not optimal for other algorithms. In section 5.1.3.2 we present our approach to tuning the server placement to different communication algorithms and obtain the optimal server placement for that specific algorithm.

Algorithm 2 Steps to identify the new groups/communicators

```
1: $mapping$ := mapProcessToServer(rankfile);
2: queryUseDef($program$, $key$, $color$);
3: $keyDef$ := replace($keyDef$, $rank$, $i$);
4: $colorDef$ := replace($colorDef$, $rank$, $i$);
5: for $i \leftarrow 0$ to $size$ do
6:     $color$ := $colorDef$;
7:     $processColors[i]$ := $color$;
8: end for
9: $newcommPrime$ := sortByColor($processColors$);
10: for $i \leftarrow 0$ to $size$ do
11:     $key$ := $keyDef$;
12:     $processToKeyMapping[i]$ := $key$;
13: end for
14: sortByKey($newcommPrime$, $processToKeyMapping$);
15: $newcomm$ := replaceProcessByServer($newcommPrime$, $mapping$);
16: $newCommMapping$ := createNewCommMapping($mapping$,
    $processToKeyMapping$);
```

5.1.3.2 Optimal Server Placement Identification

MPI has a wide range of algorithms for collective communication. There is no "one" optimal implementation. Optimality depends on multiple factors such as the system's physical topology, the number of processes involved, and message sizes. For example, there are nine algorithms for broadcast: basic linear, chain, pipeline, split binary tree, binary tree, binomial tree, knomial tree, *scatter_allgather*, and *scatter_allgather_ring*. The linear algorithm employs a single-level tree topology where the root node has $P-1$ children (P is the total number of processes/nodes). Figure 5.26 shows an example of the basic linear tree, and Figure 5.27 shows an example of the chain for a group of 8

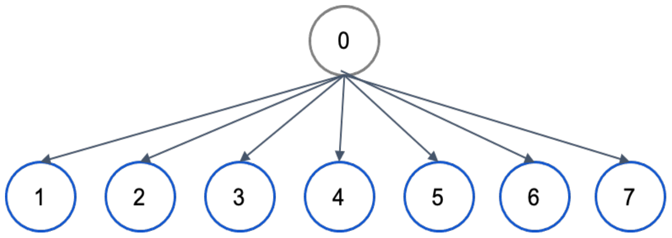


Figure 5.26 : Linear (Flat tree).

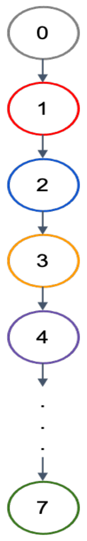


Figure 5.27 : Chain tree.

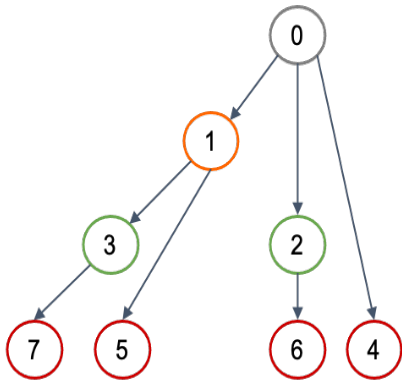


Figure 5.28 : Binomial tree.

processes ($P = 8$).

The binomial tree algorithm [136] is a well-known algorithm, and it is used to broadcast small messages. Figure 5.28 shows the structure of a binomial tree for a group of 8 processes. In the first step, the root sends the whole message to process/node 1. Next, the root and node 1 send the message to node 2 and node 3, respectively. Then the algorithm continues recursively. Here we have marked nodes with colors based on the step that they receive the message. The maximum nodal degree of the binomial tree decreases from the root down to the leaves as follows: $\lceil log_2 P \rceil$, $\lceil log_2 P \rceil - 1$, $\lceil log_2 P \rceil - 2$, ... While this algorithm is preferred for small messages, it is not suitable for large messages because some nodes send the whole message several times.

Every MPI implementation has a default algorithm. The user can change the algorithm before running their program. The algorithm that they pick will remain static

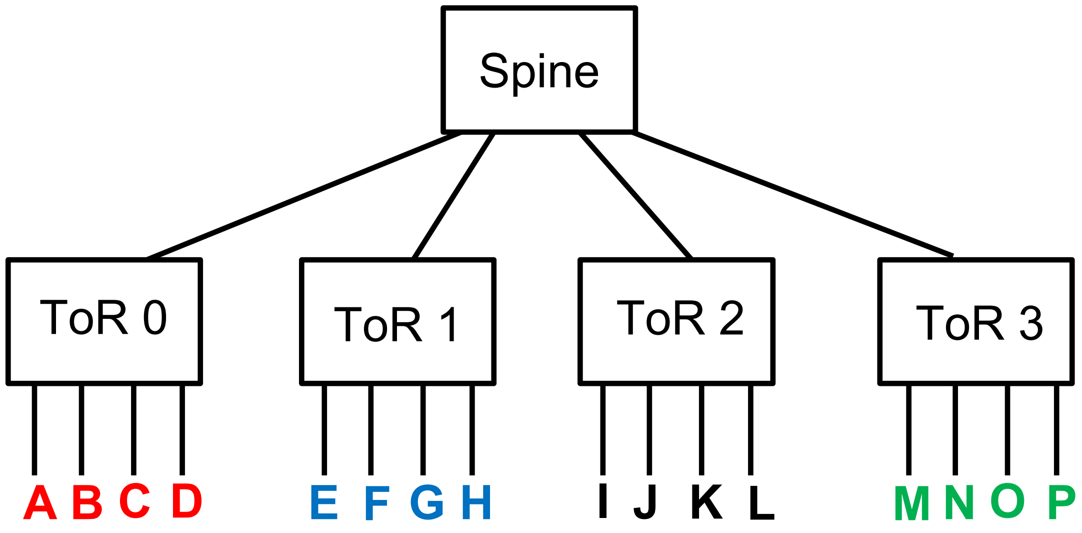
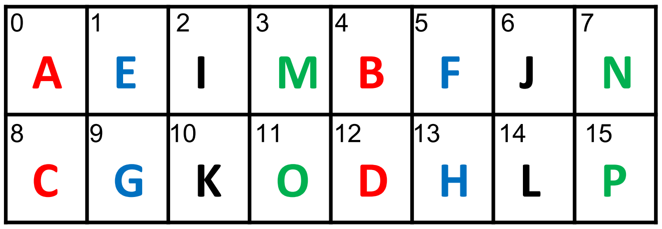


Figure 5.29 : Original server placement. A-P represent 16 servers and A-D, E-H, I-L, M-P belong to four physical racks separately.

during the runtime. This algorithm might not be the optimal algorithm for every instance of a collective communication function.

Let us see with an example: we have a network containing 16 servers and an application that runs on 16 processes spread over this network. Figure 5.29 shows the original placement of 16 servers in the network topology. *A-P* represent the 16 servers and *A-D*, *E-H*, *I-L*, *M-P* belong to four physical racks (ToRs) color-coded separately. Figure 5.30 shows the application's process to server mapping (one process per server). Processes' rank in the *WORLD* group is shown on the top left of each square box. The server the process is mapped to is shown with its letter naming in the center of each box. We have two broadcast groups. Processes with rank smaller or equal to 7 are placed in one broadcast communicator group, and the processes with rank bigger than 7 are grouped in another broadcast communicator. In the first group, the process 0 initiates a broadcast, and in the second group, process 8 in the world communicator initiates a broadcast which in the new group, this process has the rank 0.

If the broadcast algorithm in our example is set to chain, then the chains for broadcast communicators are shown in Figure 5.31. In the first chain (Figure 5.31(a)), all



0 A	1 E	2 I	3 M	4 B	5 F	6 J	7 N
8 C	9 G	10 K	11 O	12 D	13 H	14 L	15 P

Figure 5.30 : Example of a mapping of 16 processes to 16 servers.

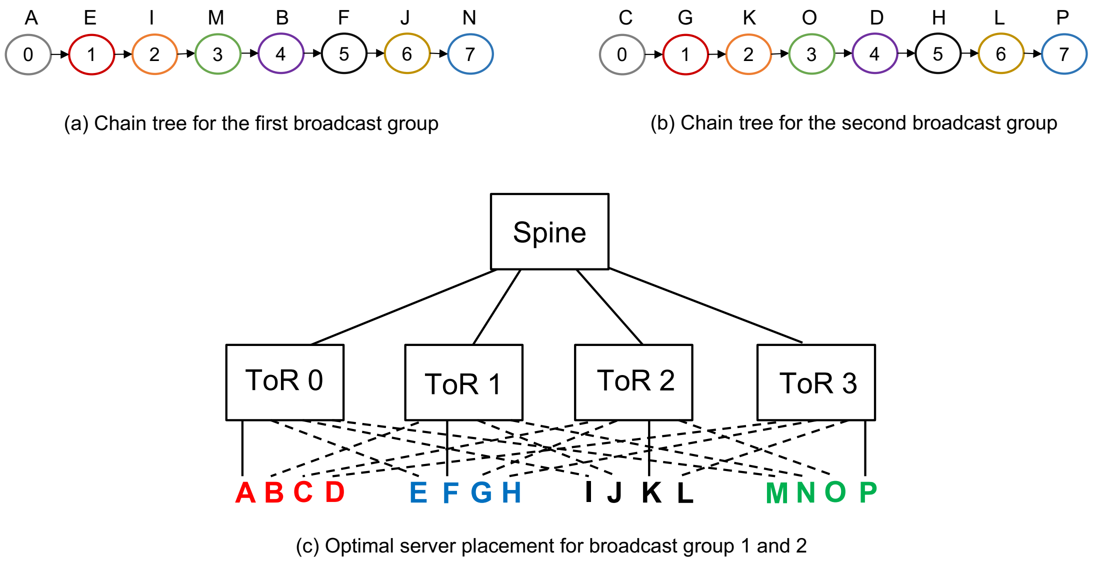

A E I M B F J N
0 1 2 3 4 5 6 7
(a) Chain tree for the first broadcast group
C G K O D H L P
0 1 2 3 4 5 6 7
(b) Chain tree for the second broadcast group
Spine
ToR 0
ToR 1
ToR 2
ToR 3
A B C D
E F G H
I J K L
M N O P
(c) Optimal server placement for broadcast group 1 and 2

Figure 5.31 : Example: Broadcast with chain algorithm. (a) shows the chain for the first broadcast group. (b) shows the chain for the second broadcast group. (c) presents the optimal server placement for both (a) and (b) in the network.

communications between nodes are cross rack: A to E, E to I, I to M, M to B, B to F, F to J, and J to N. Similarly, in the second chain (Figure 5.31(b)), all communications between nodes are cross rack: C to G, G to K, K to O, O to D, D to H, H to L, and L to P. The optimal server placement in the topology would be A, E, I, M under ToR 0, B, F, J, N under ToR 1, C, G, K, O under ToR 2, and D, H, L, P under ToR 3. With this placement, the number of cross-rack communication is minimized to the least possible in our testbed (reduced from 7 to 1 for each broadcast). Figure 5.31(c) shows our testbed after we reconfigure it with the optimal server placement sequence generated by Algorithm 2.

We assume the default algorithm for broadcast is the chain algorithm. Therefore, the sequence of servers generated by Algorithm 2 is optimal for the default collective communication algorithm (chain algorithm). However, we can tune it to other algorithms.

For example, If the broadcast algorithm is set to binomial tree, then we can tune the sequence of servers generated by Algorithm 2 and compute the optimal placement for the binomial tree algorithm. Figure 5.32(a),(b) show the binomial tree for the broadcast communicators of our example.

Figure 5.32(a) shows that in the first step server A sends to server E, which A and E are under two different racks. In the second step, E sends the data to M, and A sends the data to I, which both communications are cross rack. In the final step, M sends to N, E sends to F, I sends to J, and A sends to B. This translates to four concurrent flows which are all local exceptionally. Therefore, only step 2 oversubscribe the network. Considering both broadcast groups at step 2, network is over-subscribed (4 : 1 over-subscription).

In the first tree, Figure 5.32(a), 3 out of 7 communications are cross rack: A to E, A to I, and E to M. Similarly, in the second tree (Figure 5.32(b)), 3 out of 7 communications between nodes are cross rack: C to G, C to K, and G to O. In this example, luckily all the concurrent communications happening in the last level of the binomial trees are within racks. However, it is not always the case. An improved server placement in the topology would be as follow: Placing M, N, E, F under ToR 0, I, J, A, B under ToR 1, O, P, G, H under ToR 2, and C, D, K, L under ToR 3 (Figure 5.32(c)). With this placement, the number of cross-rack communication is minimized. It is reduced from 3 to 1 for each broadcast. Only A to E in group one and C to G in group two are cross-rack. Both cross-rack communications are in step 0. If the broadcasts happen simultaneously, then overall, there are two cross-rack communications. If we consider both broadcast groups at step 0, the network is 2 : 1 over-subscribed, and at other steps with more concurrent transmissions, there are no cross-rack flows. The number of circuits reconfigured in this placement is the highest for this setting. All 16 servers have logically moved to a different ToR; therefore, 16 circuits have been reconfigured.

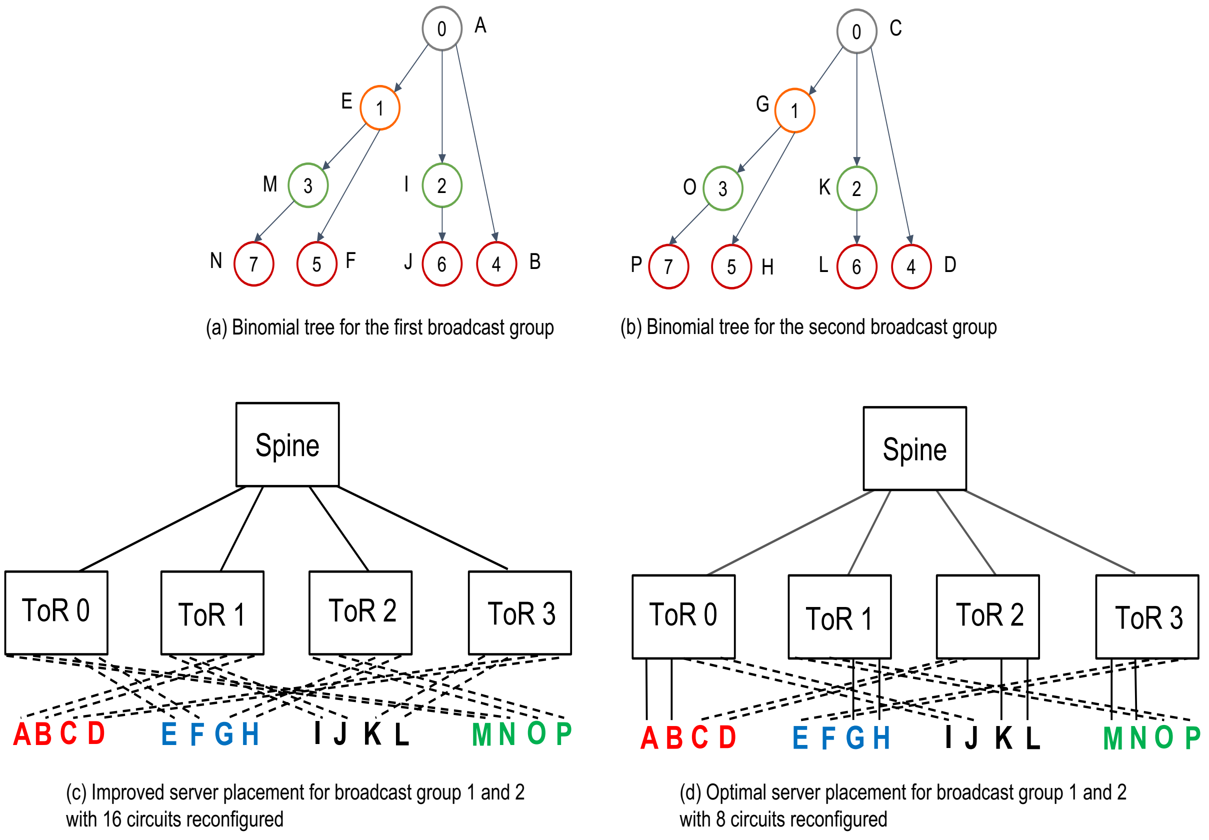

0 A
E 1
M 3
I 2
N 7
5 F
J 6
4 B
(a) Binomial tree for the first broadcast group
0 C
G 1
O 3
K 2
P 7
5 H
L 6
4 D
(b) Binomial tree for the second broadcast group
Spine
ToR 0
ToR 1
ToR 2
ToR 3
A B C D
E F G H
I J K L
M N O P
(c) Improved server placement for broadcast group 1 and 2 with 16 circuits reconfigured
Spine
ToR 0
ToR 1
ToR 2
ToR 3
A B C D
E F G H
I J K L
M N O P
(d) Optimal server placement for broadcast group 1 and 2 with 8 circuits reconfigured

Figure 5.32 : Example: Broadcast with binomial algorithm. (a) shows the binomial tree for the first broadcast group. (b) shows the binomial tree for the second broadcast group. (c) presents an improved server placement for both (a) and (b) in the network. (d) presents the optimal server placement for both (a) and (b) in the network with minimal circuit reconfigured.

In Figure 5.32(d), we present a better placement with the same performance as the placement in Figure 5.32(c) however, with minimal circuit reconfiguration required. Only 8 servers are logically moved to another ToR. In this server placement, A, B, I, J are placed under ToR 0, O, P, G, H are placed under ToR 1, C, D, K, L under ToR 2, and M, N, E, F under ToR 3. This placement is the optimal server placement because 1) it reduces the number of cross-rack flows to the lowest number possible, and 2) it logically moves the least number of servers.

As pointed out before, the sequence of servers generated by Algorithm 2 is the optimal for the default collective communication algorithm (chain algorithm). But, if the communication algorithm is set to any other algorithm, we tune the sequence of servers and find the optimal server placement for it.

If the broadcast algorithm is set to the binomial tree, the regular RDC will reconfigure the network to the order shown in the broadcast group (chain). However, our automated RDC will take into account the broadcast algorithm simultaneously as the broadcast group communicator and ranks. It calculates the optimal server placement for the example above with the lowest number of cross-rack contending flows (Figure 5.32(c)). This placement matches with the optimal server placement we discussed. It reduces the number of cross-rack communications from 3 to 1 for each broadcast: A to E in group one and C to G in group two. Automated RDC will also consider the new ranking in the broadcast group, but the original RDC does not.

Algorithm 3 shows the pseudocode for identifying the optimal server placement with the least amount of concurrent cross-rack communication for the binomial tree MPI_Bcast algorithm. This algorithm is a greedy algorithm walking bottom-up in the tree. As the number of simultaneous communication is higher as we walk down the tree, minimizing cross-rack communication is critical for the bottom of the tree.

Lines 3 to 16, goes through all leaf nodes in the binomial tree ($biTree$). Finds the leaf node's parent (line 5) and then find the server that this leaf and parent process are mapped to, respectively (line 6 to 12). Afterward, we add the pairs of server source (parent) and destinations (leaf) to the config string and the source to destination mapping. Lines 18 to 36, for every pair of source and destination "$< p, n >$", find the parent of the source node and look up in the source and destination mapping $mappingSrcDest$ to find which leaf node p' is its parent. Then add or insert the non redundant pair or node at the right location of the config string.

Algorithm 3 Tune Node/Server Placement to Binomial Tree Algorithm part 1

```
 1: string $config$ := NULL;
 2: map $mappingsSrcDst$;
 3: for each $node \in biTree$ do
 4:     if isLeaf(node) then
 5:         $parrent$ := findParent($node$);
 6:         if worldComm then
 7:             $parrent$ := findNode($parrent$,mapping);
 8:             $child$ := findNode($node$,mapping);
 9:         else
10:             $parrent$ := findNode($parrent$,$newCommMapping$);
11:             $child$ := findNode($node$,$newCommMapping$);
12:         end if
13:         insertToMappingsSrcDst($mappingsSrcDst$,$parrent$,$child$);
14:         addToConfig($config$,$parrent$,$child$);
15:     end if
16: end for
17: if $rackNodeCount/2 > 1$ then
18:     map tempMapping;
19:     for $i \leftarrow 0$ to $rackNodeCount/2 - 1$ do
20:         if $i$ !=0 then
21:             $mappingsSrcDst$ := $tempMapping$;
22:             $tempMapping$ := NULL;
23:         end if
24:                                                   ▷ Next page
```

Algorithm 3 (Continued) Tune Node/Server Placement to Binomial Tree Algorithm part 2

```
25:         for each $<p,n> \in mappingsSrcDst$ do
26:             $p' :=$ findParent($p$);
27:             $n' := mappingsSrcDst[p']$;
28:             $tempMapping[p'] := n'$;
29:             if $i == 0 \wedge$ !find($config$,"$p$,$n$") then
30:                 addToConfig($config$,$p$,$n$);
31:             else
32:                 insertToConfig($config$,"$p$,$n$", $p'$,$n'$);
33:             end if
34:         end for
35:     end for
36: end if
```

5.1.4 Point-to-Point Communication

MPI has two basic types of point-to-point data movement functions: send and receive. Sending and receiving are the two foundational concepts of MPI. The sender process calls the *MPI_Send* function to send data to the receiver process, and the receiver process calls the *MPI_Recv* function to receive the data from the sender. These functions have an argument of type *int* that specifies which process their sending to "*destination*" or receiving from "*source*" (marked in red in Figure 5.33). Similar to *color* and *key* in section 5.1.3.1.2, *destination* and *source* could be of the following type: 1) a constant value, 2) a macro, 3) a function, 4) a mathematical expression or 5) a variable.

We are interested in automatically identifying the optimal server placement and reconfiguring the network before every communication pattern changes in the application. Finding the communication pattern of point-to-point communications boils down to detecting the sender and receiver (*destination* and *source*) pairs and creating all the possible disjoint undirected graphs from these pairs.

```
1. MPI_Send(void* data, int count, MPI_Datatype datatype, int destination,

        int tag, MPI_Comm communicator)

2. MPI_Recv(void* data, int count, MPI_Datatype datatype, int source,

        int tag, MPI_Comm communicator, MPI_Status* status)
```

Figure 5.33 : MPI point-to-point data movement functions.

If *source* and *destination* are not of type constant, the definition depends on the rank of the calling process.

In the example code Listing 5.2, we have a MPI_Send and MPI_Recv pair that their *destination* and *source* are defined earlier in the source code with a mathematical equation and depend on process rank. Every process runs this code in parallel and computes a different *destination* and *source* pair. The only variable that has a different value for each process in these equations is *rank*.

Algorithm 4 shows the pseudocode for identifying the optimal node placement for pairs of MPI_Send and MPI_Recv. We query the use-def analysis graph to find the reaching definition of *destination* to a specific MPI_Send and the reaching definition of *source* to the specific MPI_Recv in the program (*line* 3 and *line* 5 in Algorithm 4). We want to find the *source* and *destination* for every process in the program. We replace the variable referring to the process rank in the definition of *source* and *destination* with the variable i (*line* 4 and 6). We compute the *source* and *destination* process rank for all processes included in MPI_COMM_WORLD ($\forall i : 0 \leq i < size(\text{MPI\_COMM\_WORLD})$) (*line* 8 and 9). We find server bounded to each *source* and *destination* process rank then add the source and destination server pair to a source to destination mapping *mappingsSrcDst* (*line* 10 to 12). We find all the disjoint undirected graphs made from the source and destination node mappings (*line* 14 to 41). The string variable $config$ contains the server placement order to be passed to the controller. Note that only the process rank 0 of the MPI_COMM_WORLD runs this algorithm.

Algorithm 4 Send and Receive node/server placement part 1

```
 1: string $config$;
 2: map $mappingsSrcDst$;
 3: $destinationDef$ := queryUseDef($program$, $destination$);
 4: $destinationDef$ := replace($destination$, $rank$, $i$);
 5: $srcDef$ := queryUseDef($program$, $source$);
 6: $srcDef$ := replace($source$, $rank$, $i$);
//**Insert all the LOC below into the source code before the specific** $MPI\_send$
 7: **for** $i \leftarrow 0$ to $size$ **do**
 8:     $destinationPrime$ := $destinationDef$;
 9:     $srcPrime$ := $srcDef$;
10:     $src$ := findNode($i$,mapping);
11:     $dst$ := findNode($destinationPrime$,mapping);
12:     insertToMappingsSrcDst($src$,$dst$); //do not add if redundant
13: **end for**
14: **for each** <s,d> $\in$ $mappingsSrcDst$ **do**
15:     **if** s = d **then**
16:         **if** s not in t **then**
17:             addToConfig($config$,s);
18:             **if** findCount($mappingsSrcDst$,d) > 1 **then**
19:                 $mappingDestinationSender$ := findAllMapping(d);
20:                 **for each** <d,d'> $\in$ $mappingDestinationSender$ **do**
21:                     **if** d != d' **then**
22:                         addToConfig($config$,d');
23:                     **end if**
24:                 **end for**
25:             **end if**
26:         **end if**
27:     **else**
28:                                                         ▷ Next page
```

Algorithm 4 (Continued) Send and Receive node/server placement part 2

```
29:         if <s,d> not in $config$ then
30:             if <d,s> not in $config$ then
31:                 addToConfig($config$,s,d);
32:                 $mappingSameSender$ := findAllMapping(s);
33:                 for each <s,d'> in $mappingSameSender$ do
34:                     if d != d' then
35:                         addToConfig($config$,d');
36:                     end if
37:                 end for
38:             end if
39:         end if
40:     end if
41: end for
```

We insert lines 7 to 41 in the program source code before the specific MPI_Send, and the values for the optimal server placement are computed at runtime and passed to the controller to reconfigure the network before this point-to-point communication starts.

The Listing 5.3 shows the *destination* and *source* calculation code inserted in the source code from Listing 5.2 and Table 5.2 shows the pairs for our example with the number of processes in MPI_COMM_WORLD set to 16. Figure 5.34 shows the step-by-step Send and Receive disjoint undirected graphs creation. For this example, with the process to server mapping shown in Figure 5.35 the optimal server placement is "A E I M B F J N C G K O D H L P" which is presented in Figure 5.36. Servers are logically placed in this order: A, E, I, and M under ToR 0, B, F, J, and N under ToR 1, C, G, K, and O under ToR 2, and D, H, L, and P under ToR 3. This placement reduces concurrent cross-rack flows from 16 to 0; therefore, all communications happen at line-rate.

Listing 5.2: Send and Receive example

```
int size, rank;
//Determines the size of the group associated with MPI_COMM_WORLD
MPI_Comm_size(MPI_COMM_WORLD, &size);
//Determines the rank of the calling process in the MPI_COMM_WORLD
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int maxIndex = sqrt(size);
int destination = (rank + (maxIndex - 1) * maxIndex) % (maxIndex * maxIndex);
int source = (rank + maxIndex) % (maxIndex * maxIndex);
MPI_Send(s, size, MPI_INT, destination, 0, MPI_COMM_WORLD);
MPI_Recv(s, size, MPI_INT, source, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Listing 5.3: Source and destination pair calculation example

```
int size, rank;
//Determines the size of the group associated with MPI_COMM_WORLD
MPI_Comm_size(MPI_COMM_WORLD, &size);
//Determines the rank of the calling process in the MPI_COMM_WORLD
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int maxIndex = sqrt(size);
//Calculates the destination rank
int destination = (rank + (maxIndex - 1) * maxIndex) % (maxIndex * maxIndex);
//Calculates the source rank
int source = (rank + maxIndex) % (maxIndex * maxIndex);
//Calculates all pairs of source and destination
int dest;
if(rank == 0){
  cout << "Source Destination" << endl;
  for (int i = 0; i < size; i++){
    dest = (i + (maxIndex - 1) * maxIndex) % (maxIndex * maxIndex);
    cout << i <<" " << dest << endl;
  }
}
MPI_Send(s, size, MPI_INT, destination, 0, MPI_COMM_WORLD);
MPI_Recv(s, size, MPI_INT, source, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Source	Destination	Source	Destination
0	12	8	4
1	13	9	5
2	14	10	6
3	15	11	7
4	0	12	8
5	1	13	9
6	2	14	10
7	3	15	11

Table 5.2 : Source and Destination pairs for world communicator of size 16.

5.2 Potential Applications

Here we look at a few out of many applications that will potentially benefit from our solution.

5.2.1 Hierarchical Tucker Tensor Decomposition

Massive amounts of multidimensional data with multiple aspects and high dimensionality are generated in big data processing/analysis such as computational neuroscience, neuro-informatics, pattern/image recognition, imaging data analysis, signal processing and machine learning. Analyzing large-scale, multidimensional data sets is challenging. Tensors are a good suit to represent such massive multidimensional data in a compact way [137]. Diverse branches of science use tensors; such as data analysis, signal and image processing [138, 139, 140, 141], quantum physics, and quantum chemistry [142, 143, 144].

A tensor is d-dimensional array with a uniform type. d is called the order of the tensor. It is impossible to store a higher-order tensor explicitly because it grows exponentially with d. To address this issue various data-sparse formats have been presented.

The hierarchical Tucker decomposition format is a storage-efficient scheme to approx-

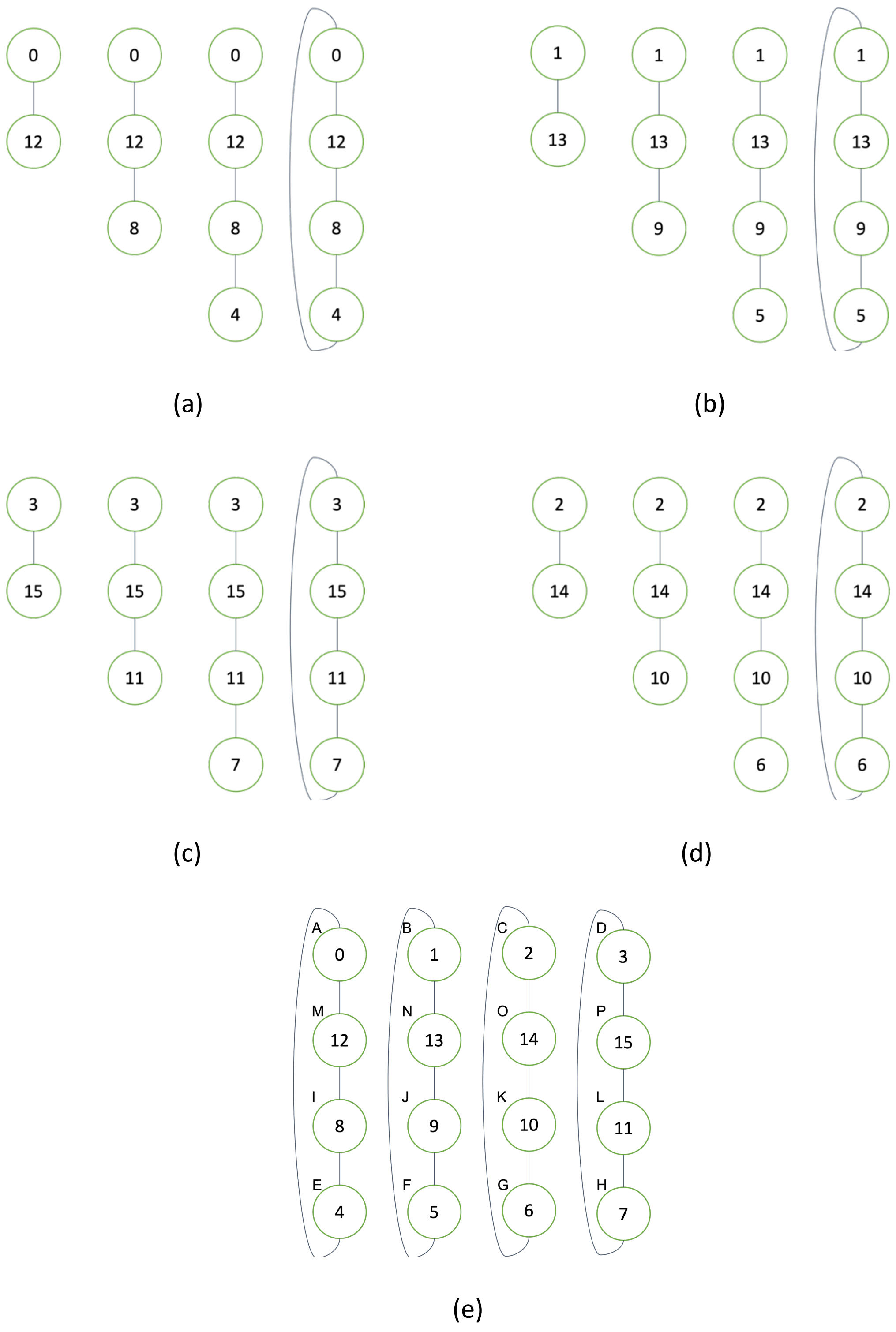


Figure 5.34 : Steps in the Send and Receive disjoint undirected graphs creation. (a) Steps to build the first send and receive graph from left to right. (b) Steps to build the second send and receive graph from left to right. (c) Steps to build the third send and receive graph from left to right. (d) Steps to build the last send and receive graph from left to right. (e) All the disjoint graphs from send and receives of the 16 processes.

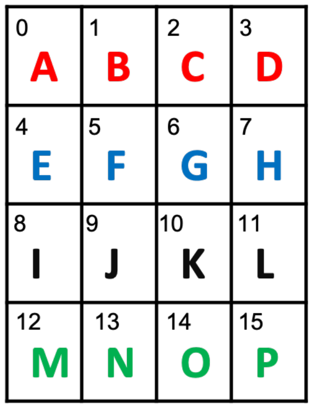


Figure 5.35 : Example of a mapping of 16 processes to 16 servers.

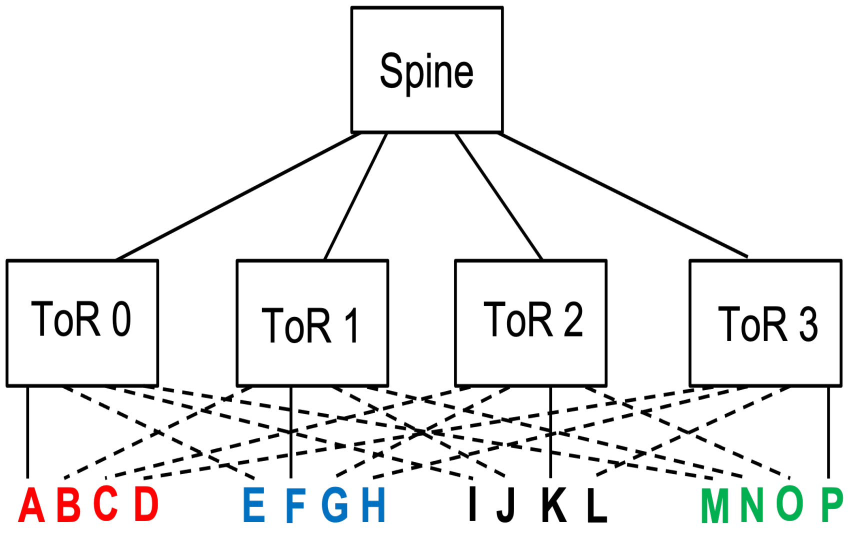
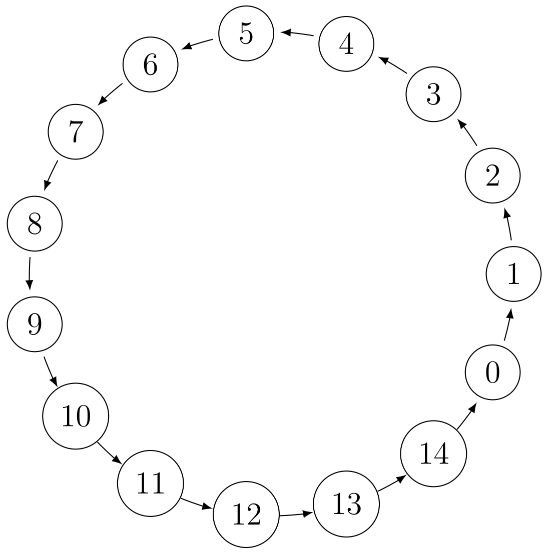
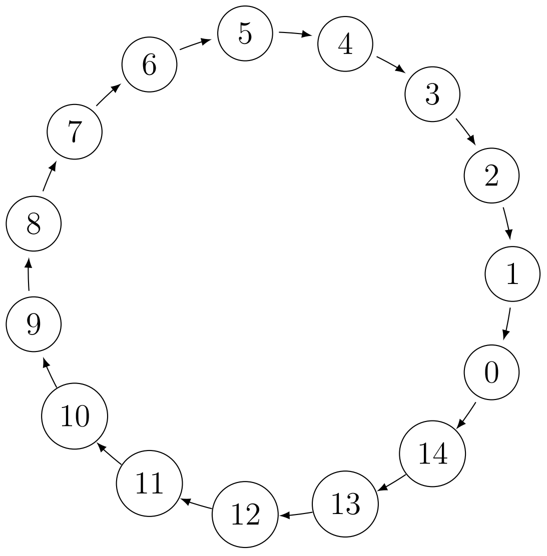


Figure 5.36 : Example: The optimal server placement for send and receive.



(a) First communication pattern.



(b) Second communication pattern.

Figure 5.37 : 2D-Heat diffusion equation communication patterns.

imate and represent tensors which is well-suited for high-order tensors [145, 146, 147]. It avoids exponential growth of storage requirements as opposed to the Tucker decomposition scheme.

The hierarchical Tucker format is a tree network (has no cycles), composed of tensors of order at most three. Each tree node is associated with a different compute node. There are toolboxes for the construction and manipulation of tensors in the hierarchical Tucker format [147].

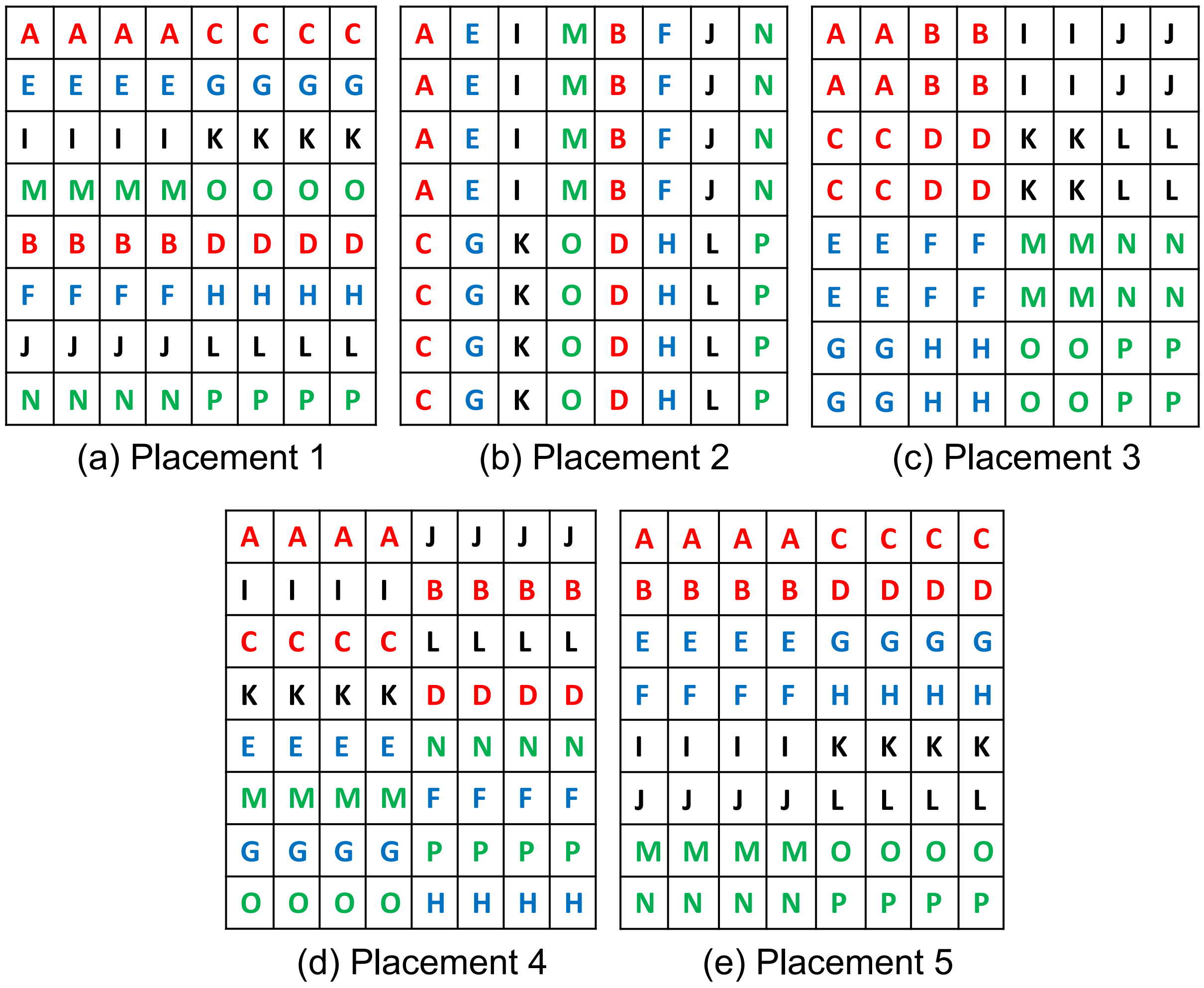
A high performance computing oriented parallel implementation of the Hierarchical Tucker tensor decomposition toolbox [148] can benefit from our solution [149] —always the constructor uses MPI in the building of the tree. Message size is small. Also, each node communicates with the other nodes on the tree through the Open MPI message passing library. But the size of the data depends on the type of operation performed. Some operations that have no communication or the data size is insignificant there is no need to reconfigure the network. However, some operations have larger data exchanged and depending on the placement of the processes they can benefit from the automated network reconfiguration.

5.2.2 2D-Heat Diffusion Equation

Heat equation was first developed for modeling how a quantity such as heat diffuses through a given region [150]. It is one the most widely studied topics in pure mathematics, and its analysis is regarded as fundamental to the broader field of partial differential equations. It is also important in many fields of science and applied mathematics such as probability theory, quantum mechanics, image analysis, hydrodynamical shocks, and numerical analysis.

The 2D-heat diffusion equation can be solved by centered finite differences in space and the forward Euler method in time. The finite difference method provides a good numerical approximations to the solution of the 2D-heat equation. Finite difference algorithms are well suited for parallel programming. As the number of evaluation points increases, the effect of parallelism becomes more significant because each processor has to carry bigger computational effort in every iteration.

The first point to point communication pattern is as follows each process $p : 0 \leq p \leq np$ (np is the number of processes in the group) communicates with the next process $p+1$ counter clockwise communication. Only the last process $p = np - 1$ communicates with process 0. Figure 5.37a shows the first communication pattern for $np = 16$. The second communication pattern is the reverse of the first pattern. Every process $p : 0 \leq p \leq np$ communicate with the process $p - 1$ clockwise communication. Only the last process 0 communicates with process $np - 1$. Figure 5.37b shows the second communication pattern for $np = 16$. We want to see if our automated solution can benefit this testcase. Figure 5.29 shows the RDC testbed and our server to rack color coding. Figure 5.39 shows the first communication pattern for the 64 processes. We consider five placements for the $np = 64$ processes over 16 servers: 1) Figure 5.38(a): places them row-wise- one cross-rack traffic in every 8, 2) Figure 5.38(b): places them column-wise- all traffic are cross-rack, 3) Figure 5.38(c): one cross-rack traffic in every 4, 4) Figure 5.38(d): places the processes in a mixed manner, one cross-rack traffic in every 8 or one cross-rack traffic in every 4, 5) Figure 5.38(e): places the processes in a manner (some traffic are cross-rack



A	A	A	A	C	C	C	C
E	E	E	E	G	G	G	G
I	I	I	I	K	K	K	K
M	M	M	M	O	O	O	O
B	B	B	B	D	D	D	D
F	F	F	F	H	H	H	H
J	J	J	J	L	L	L	L
N	N	N	N	P	P	P	P

(a) Placement 1

A	E	I	M	B	F	J	N
A	E	I	M	B	F	J	N
A	E	I	M	B	F	J	N
A	E	I	M	B	F	J	N
C	G	K	O	D	H	L	P
C	G	K	O	D	H	L	P
C	G	K	O	D	H	L	P
C	G	K	O	D	H	L	P

(b) Placement 2

A	A	B	B	I	I	J	J
A	A	B	B	I	I	J	J
C	C	D	D	K	K	L	L
C	C	D	D	K	K	L	L
E	E	F	F	M	M	N	N
E	E	F	F	M	M	N	N
G	G	H	H	O	O	P	P
G	G	H	H	O	O	P	P

(c) Placement 3

A	A	A	A	J	J	J	J
I	I	I	I	B	B	B	B
C	C	C	C	L	L	L	L
K	K	K	K	D	D	D	D
E	E	E	E	N	N	N	N
M	M	M	M	F	F	F	F
G	G	G	G	P	P	P	P
O	O	O	O	H	H	H	H

(d) Placement 4

A	A	A	A	C	C	C	C
B	B	B	B	D	D	D	D
E	E	E	E	G	G	G	G
F	F	F	F	H	H	H	H
I	I	I	I	K	K	K	K
J	J	J	J	L	L	L	L
M	M	M	M	O	O	O	O
N	N	N	N	P	P	P	P

(e) Placement 5

Figure 5.38 : Five different process to server placements. A-P represent 16 servers and A-D, E-H, I-L, M-P belong to four physical racks separately.

but not all) that there is one cross-rack traffic in every 16.

Based on Figure 5.39, placement 5 is already optimal so our solution would not be able to reduce the communication time further but we can automatically reconfigure the network so that this testcase perform as well as placement 5 for the other four process to server placements.

5.2.3 Distributed Matrix Multiplication

Distributed Matrix Multiplication (DMM) is another application that can benefit from our solution. Matrices are decomposed into *np* blocks. In each iteration of a commonly

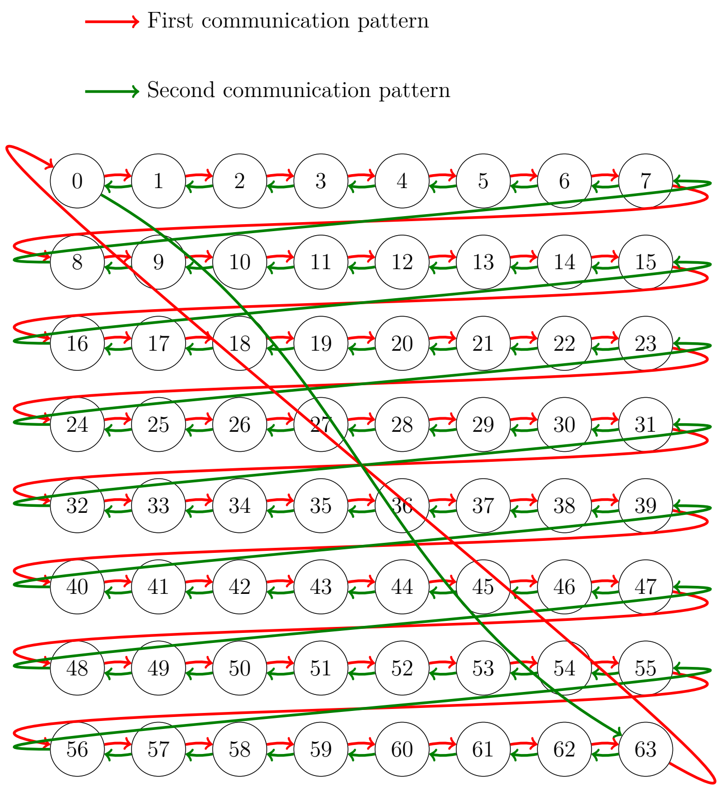


Figure 5.39 : 2D-Heat diffusion equation first communication pattern for 64 processes.

used DMM algorithm [47, 63] , the algorithm performs a "broadcast-shift-multiply" cycle where a process a) broadcasts its submatrix block row-wise, b) shifts its submatrix block column-wise, and c) multiplies submatrices. Figure 5.40 shows the traffic pattern for $np = 64$. Every row and column has $\sqrt{np} = 8$ processes that form a 8×8 2D process layout. The red arrows show the first communication pattern (row-wise broadcast) and the green arrows show the second communication pattern (column-wise shift).

We want to see if our automated solution can benefit this testcase. Our server to rack assignment in our testbed and color coding is shown in Figure 5.29. Considering the five placements in Figure 5.38 for the $np = 64$ processes over 16 servers. Each server has 4 processes.

1) Figure 5.38(a): places them row-wise zero cross-rack traffic for broadcast but all

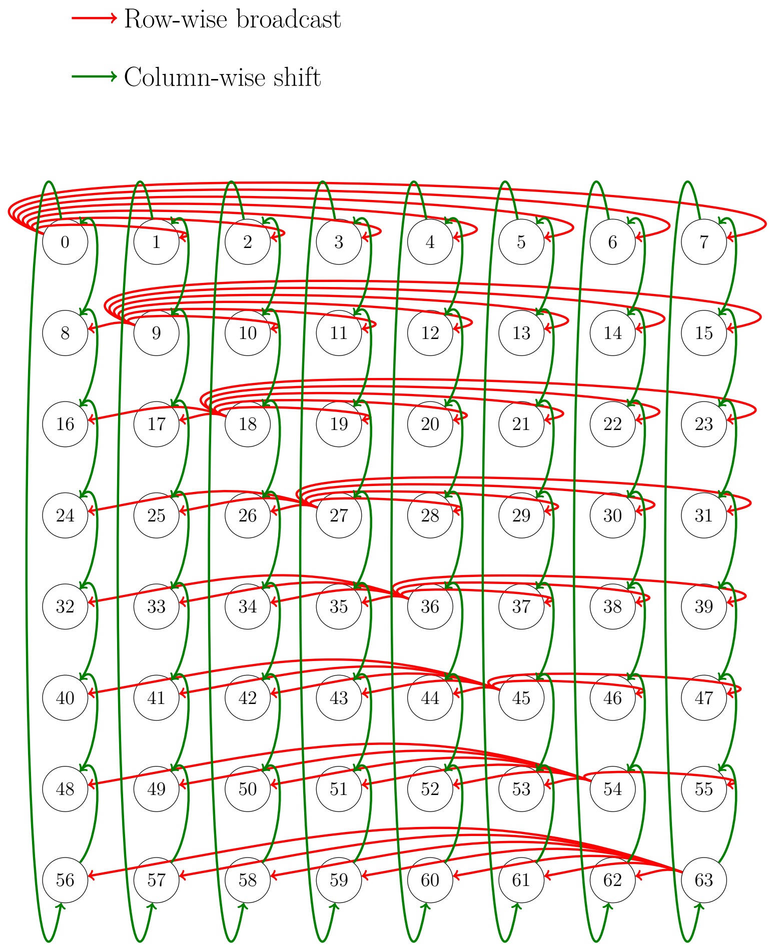


Figure 5.40 : Distributed matrix multiplication traffic patterns for 64 processes.

shift traffics are cross-rack, 2) Figure 5.38(b): places them column-wise zero cross-rack traffic for shift but all broadcast traffic are cross-rack, 3) Figure 5.38(c): places the processes in a mixed manner, considering both broadcast and shift traffic across racks, 4) Figure 5.38(d): places the processes in a mixed manner (some traffic are cross-rack but not all) for broadcast but all shift traffics are cross-rack, 5) Figure 5.38(e): places the processes in a mixed manner considering shift but places them row-wise considering broadcast (zero cross-rack traffic).

Based on Figure 5.40, none of the placements is optimal for both of the communication patterns in the DMM algorithm. Using our solution (Algorithm 1, Algorithm 3, and Algorithm 4) we can automatically find the optimal placement for each communication pattern and reconfigure the network before the communication pattern changes in every iteration. This will reduce the communication time for both broadcast and shift by logically moving the servers and minimizing the number of cross-rack communications. For example, for the first communication pattern with placement 1 and broadcast algorithm set as chain, there are no cross-rack flows but the second communication pattern has 8 cross-rack flows per column therefore, we need to reconfigure the network to the optimal placement for the second communication pattern to minimize the number of cross-rack flows. Our solution will place A, E, I, and M under ToR 0, B, F, J, and N under ToR 1, C, G, K, and O under ToR 2, and D, H, L, and P under ToR 3. The number of cross-rack flows drops from 8 to 2 per column. Only, B to M and A to N remain cross-rack. In the next iteration, the network is not suited for the first communication pattern so we also need to reconfigure the network for the first communication pattern. Our solution will place A, C, E, and G under ToR 0, I, K, M, and O under ToR 1, B, D, F, and H under ToR 2, and J, L, N, and P under ToR 3.

5.3 Implementation

The input to our prototype consists of the source file of a C++/MPI program, API specification file and a rankfile. The analyzer—our module for communication pattern identification and code transformation—is based on the ROSE framework [151]. ROSE is a compiler infrastructure supplying methods for source-to-source code analysis and transformation. Our program inserts the reconfiguration code in the source code, passes the new placement order of servers to the controller before the corresponding communication function. We first create a general socket connection to the controller. For each reconfiguration, we use the same socket. The controller expects a string containing the servers/nodes. Based on the order that the servers' name/IP have in the $config$ string, the circuit switch is reconfigured, and the servers will be placed in that order. Note that we only need the process with rank 0 of the MPI_COMM_WORLD to send the node order to the controller.

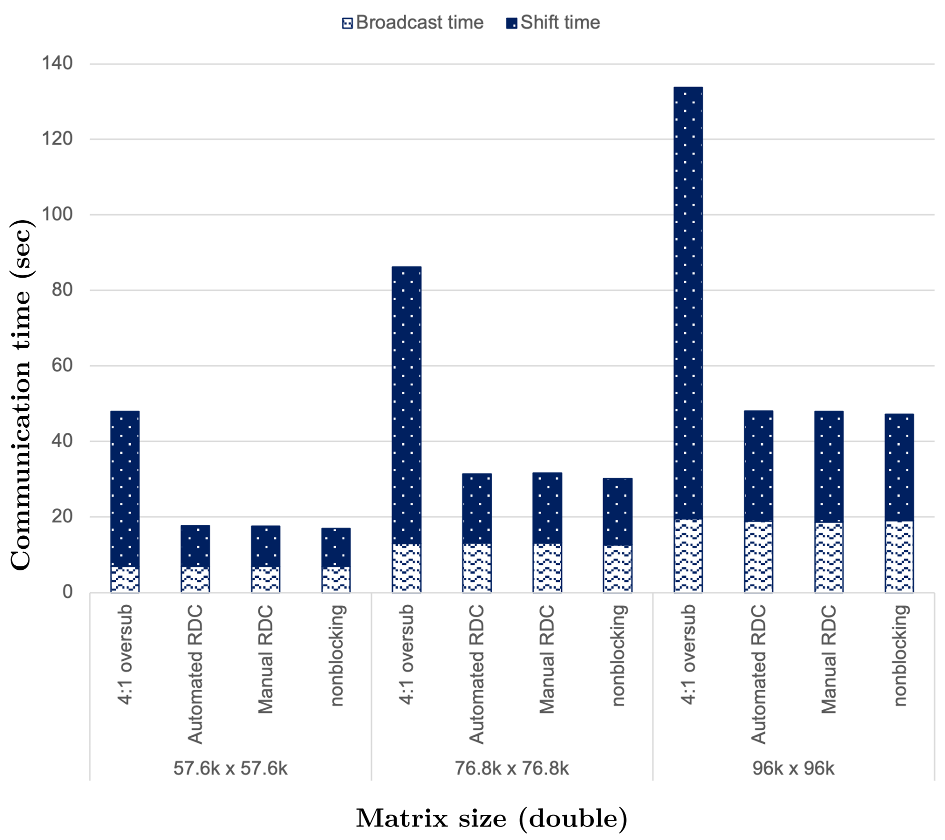


Figure 5.41 : Placement 1: DMM average shift time and broadcast time.

5.4 Experimental Results

We set up a 16-node Open MPI cluster across 4 racks and implement a commonly used Distributed Matrix Multiplication (DMM) algorithm [47, 63] with 64 processes. Matrices are decomposed into 64 blocks. Each server has 4 processes to form a 8×8 2D process layout. We presented the different communication patterns in each iteration in Section 5.2.3 (Figure 5.40). We run experiments with the first three placements from Figure 5.38 for the 64 processes over 16 servers.

We ran each of the combinations of matrix sizes, networks (Automated RDC, Manual RDC, static 4 : 1 oversubscribed network, NBLK network) 10 times and obtained their average running time. The analysis and program extension take around 27.5 seconds. The communication group identification takes approximately 9 milliseconds for a communication world of size 64.

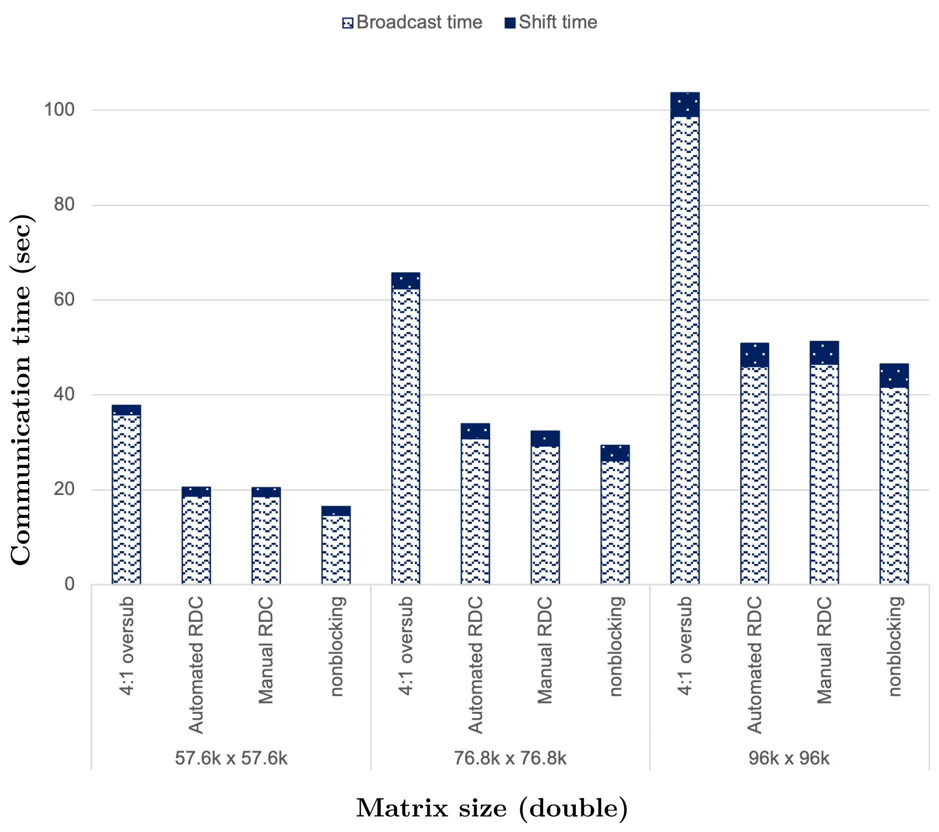


Figure 5.42 : Placement 2: DMM average shift time and broadcast time.

Figure 5.41 shows that for the process to server placement 1, our automated solution improves the overall communication time by 2.78× and the shift time by 3.93× compared to a static 4 : 1 oversubscribed network. The manual solution improves the overall communication time by 2.79× and the shift time by 3.91× compared to a static 4 : 1 oversubscribed network. The amount of improvement due to automated RDC remains the same as the manual improvement. Our experiment results for placement 2 and placement 3 shown in Figure 5.42 and Figure 5.43 makes the same point as above. The automated RDC performs as well as manual RDC and close to the nonblocking network.

5.5 Related Work

Researchers have studied communication pattern identification with different goals in mind including debugging and performance analysis, optimization, job scheduling, target system selection, and system design.

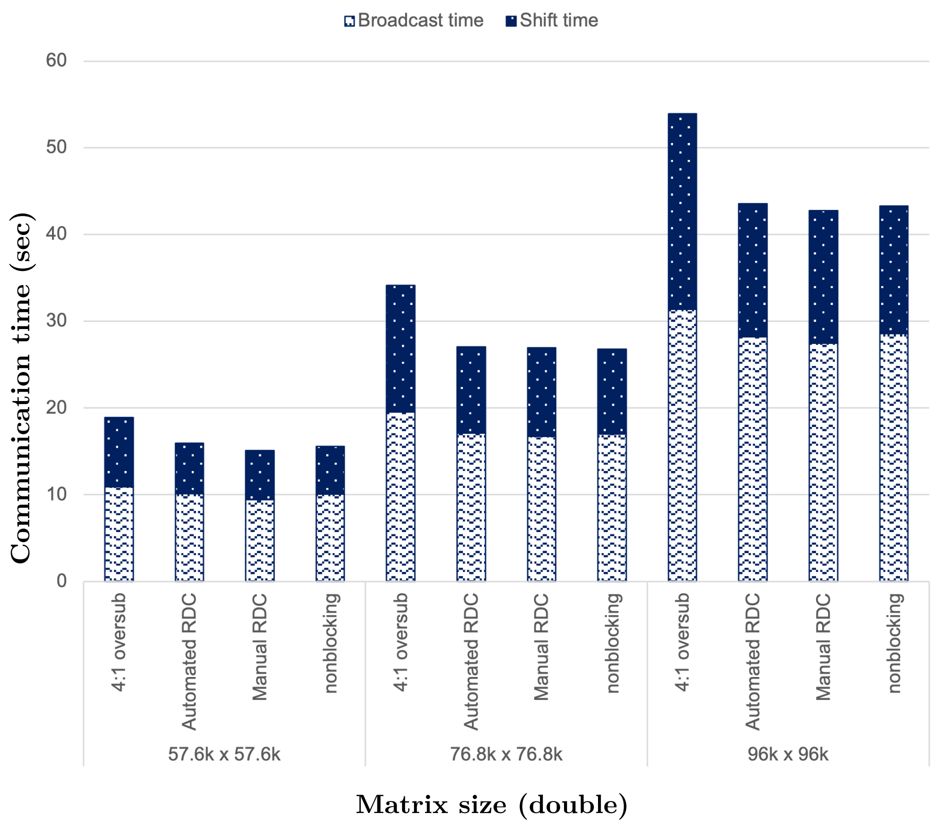


Figure 5.43 : Placement 3: DMM average shift time and broadcast time.

Oxbow toolkit [152] is a collection of tools that empirically characterize application behaviors independent of performance. It characterizes on several axes: computation, communication, memory capacity and access patterns. Oxbow uses mpiP library[153] which is a lightweight profiling tool used to generate statistics and communication topology data. It stores the volume of data transferred between ranks in an adjacency matrix. The communication topology is the result of visualizing the adjacency matrix for an application run.

It captures communication topology, message size histograms for point-to-point operations and captures message size histograms for collective operations. Oxbow, uses this data to compare communication patterns and data transfer volumes across different applications and their input sets. It can predict what kinds of system attributes better matches an application by testing a given system with a given application.

Roth et al. [154] propose AChax (Automated Communication Pattern Characterization) [155] which captures communication pattern recognition expertise in an automated tool. The mpiP profiling tool is used to collect information about an application's communication topology and message volume. Given data describing application communication behavior, a search-based analysis is used to compare the application's observed communication pattern against a library of common communication patterns.

Later they propose a hybrid approach [156] that combines deep learning classification, and regression with the existing AChax. It is more effective to detect and parameterize the communication patterns and pattern combinations used by some real-world applications but it is very slow.

Communication performance has substantial variation on parallel computing clusters depending on if the communication is on-node, intra-rack, or cross-rack locations. Communication aware/optimal process placement can be critical for the overall runtime when the application communication patterns are irregular. A high level understanding of communication patterns as well as their relation to source code structures is required to optimize the performance of High Performance Computing (HPC) applications.

Cornea et al. [157] propose a static program analysis approach (using AST and SDG) which identifies application communication signature topologies such as star, ring, mesh, or torus. They have formally defined a few communication topologies such as star, ring, mesh, or torus that they check the application communication signature against. Their goal is to understand program communication patterns in order to minimize the impact of capacity variation in communication parameters and use this knowledge to choose the best execution platform. As opposed to the previous lines of work, this work does not rely on profiling or using execution traces and it statically identifies the communication topology.

Preissl et al. [158] propose an algorithm to detect communication patterns in parallel traces in order to identify the Send and Receive events that are part of inefficient/repetitive communication patterns (e.g., poorly implemented broadcast operations). These inefficient communication patterns are good targets for source code optimizations by replacing them with more efficient equivalent operations. They achieve their goal by taking two steps. The first step is to detect the inefficient communication patterns which relies on dynamic program analysis. They annotate the target MPI application to generate an MPI trace of the program executed under a given set of parameters. They use pattern matching to find recurring inefficient communication structures. The second step is to do the program transformation which relies on static program analysis. They generate an abstract syntax tree (AST) of the application and perform a static analyses to extract control and data flow. They do mapping between the detected communication patterns and the static analyses information and guide potential source-to-source transformations.

5.6 Summary

Many applications have oscillating traffic patterns therefore no static placement is suitable for every communication pattern. Our automated static analysis-based solution can complement RDC and, without user involvement, instruct the program with the appropriate reconfiguration code to improve the performance and dynamically optimize the topology at runtime. Our solution can even benefit applications with a single traffic pattern. It can find the optimal placement for that specific traffic pattern. For these applications, we need to reconfigure the network only once at the starting point of the application as opposed to applications with changing communication pattern which we need to reconfigure the network multiple time (before every oscillation). Our experimental results on automated intra-pod localization show that automated RDC can perform as well as manual RDC and decrease communication time considerably for real-world applications.

Chapter 6

Conclusions and Future Work

The main goal of this thesis is to investigate whether challenges faced by HPC applications in the communication phase can be addressed by augmenting/tweaking the existing data center network architectures with low-cost optical technologies.

We study data center networks and provide an analysis of the literature covering various research areas, including data center network interconnection architectures, network protocols for data center networks, and state-of-the-art communication frameworks.

We propose a broad scale of solutions to improve HPC application performance by optimizing and improving the communication performance for HPC applications. We keep in mind to retain as many of the existing benefits of current data center networks as possible without significantly changing the network and increasing the hardware cost.

6.1 Concluding remarks

Overall, the results from the above chapters 3, 4, and 5 suggest a positive answer to our main question " Whether augmenting/tweaking the existing data center network architectures with low-cost optical technologies can solve issues that HPC applications are facing in current data center networks?". Specifically, the highlights of our results are:

- We show that augmenting the existing data center network with a dedicated multicast network such as Shufflecast can relieve the bandwidth requirement pressure on the existing network and improve multicast communication performance of real-world applications with minor modifications, which leads to overall application runtime performance improvement.

- We show that a dynamic network such as RDC that can adjust its topology for different traffic patterns can improve the overall network performance of real-world applications with minor modifications regardless of the communication type (multicast or unicast).

- We design a static analysis-based communication pattern detection and optimal server placement identification solution for HPC applications. We show that using our solution can complement and make a dynamic/ reconfigurable network such as RDC more accessible and user-friendly while maintaining the same performance. Our experimental evaluations show that the optimal server placement automatically identified by our solution can speedup a 4 : 1 oversubscribed network to achieve nearly non-blocking network's performance similar to the original manual RDC.

6.2 Future work

This section points out directions for future works that leverage automated rackless data center networks. Although this thesis has made progress towards understanding how to improve network performance for HPC applications through software and hardware augmentation to today's DCN architecture, some exciting problems remain unresolved: The solutions we proposed in Chapter 4 and Chapter 5 optimize the network topology for the different communication patterns of a single application. However, it is likely that multiple applications run on the cluster. We want to answer this question: "Can we optimize the network topology to the needs of multiple concurrent applications?".

Handling Concurrent Applications. First, let us look at some simple examples. Imagine we have 32 servers belonging to eight physical racks in our testbed as shown in Figure 6.1, and two DMM applications run in this cluster. In this setting, each application with 64 processes run on 16 different servers. The mappings of processes to servers are shown in Figure 6.2. For both mappings, the row-wise broadcasts are intra-rack; however, there are 8 cross-rack flows for every column-wise shift. Because these two applications do not share any servers, our current solution can handle it and find the optimal placement that minimizes the number of cross-rack communications for both applications. Each application can separately reconfigure its section of the network without hurting the other concurrent running application. Figure 6.3 shows the topology that reduces the number of cross-rack flows for each column-wise shift from 8 to 2.

What happens if the servers are shared? Figure 6.4 shows an example in which we have 24 servers belonging to six physical racks. Two DMM applications (DMM1 and DMM2) with the process to server mapping shown in Figure 6.2 run on this testbed concurrently. The servers under ToR 0 (A, B, C, D), ToR 1 (E, F, G, H), ToR 2 (I, J, K, L), and ToR 3 (M, N, O, P) are shared by the DMM applications. The first DMM application uses four cores of the sixteen first servers and the second DMM application uses two cores of the sixteen first servers plus four cores of the eight last servers belonging to ToR 4 (Q, R, S, T), and ToR 5 (U, V, W, X).

If we assume that both identical DMM applications start at the same time and work on the same size of matrices, then the points in the program that the communication pattern changes are the same, and since they run on the same size of matrices, both applications will reach to those points approximately at the same time. With some stretching, our algorithms in Chapter 5 can minimize the number of cross-rack flows for both applications. To find the optimal placement for the joint applications: 1) it has to prioritize between the applications per zone/ToR and 2) only one process from one of the applications should be in charge to reconfigure the network and not both, or a controller can take charge.

A technique to find the application with a higher priority in one zone/ToR is to find the application with the highest number of cross-rack flows in that zone/ToR. In our example, the number of cross-rack flows originating from the first four racks by the first DMM application is higher than the second DMM app. Therefore, DMM1 has a higher priority for the first four racks. We place A, E, I, M under ToR 0, Q, U, Y, AC under ToR 1, C, G, K, O under ToR 2, and S, W, AA, AE under ToR 3. This placement minimizes the number of cross-rack flows for each column-wise shift from 8 to 2 for

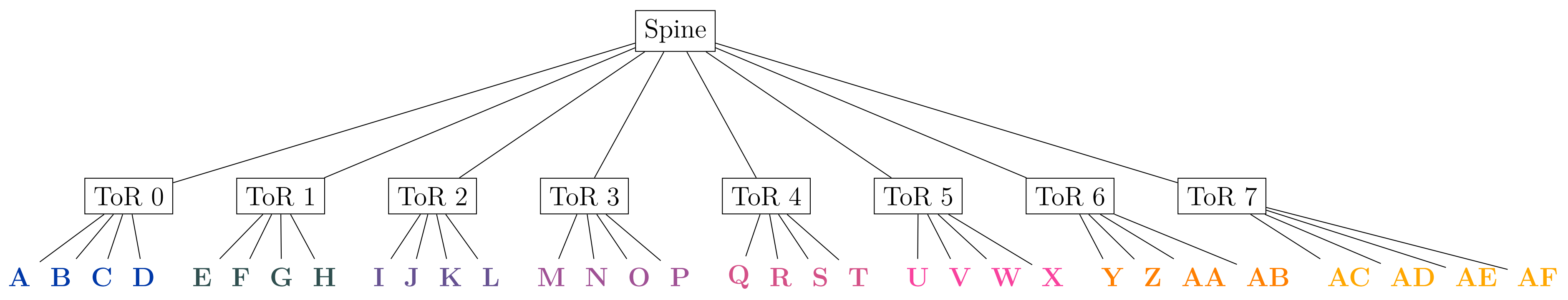
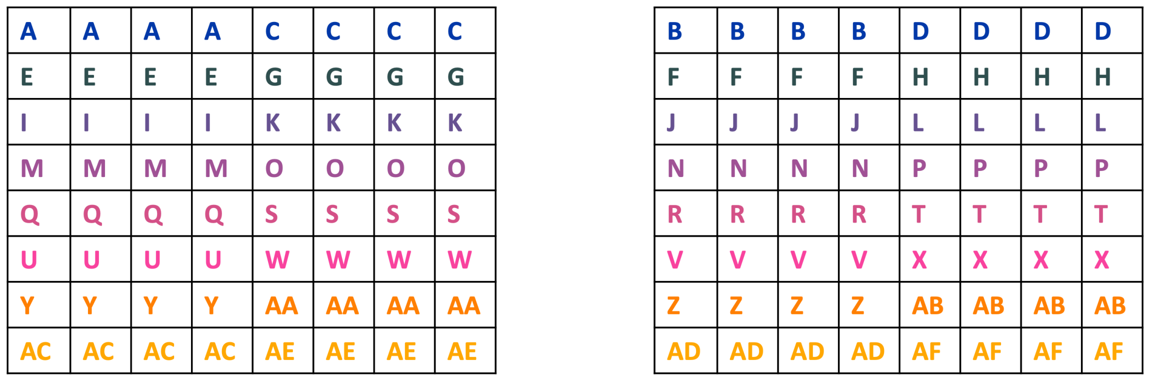


Figure 6.1 : The original server placement in the testbed. A-X represent 32 servers and A-D, E-H, I-L, M-P, Q-T, U-X, Y-AB, AC-AF belong to eight physical racks separately.



A	A	A	A	C	C	C	C
E	E	E	E	G	G	G	G
I	I	I	I	K	K	K	K
M	M	M	M	O	O	O	O
Q	Q	Q	Q	S	S	S	S
U	U	U	U	W	W	W	W
Y	Y	Y	Y	AA	AA	AA	AA
AC	AC	AC	AC	AE	AE	AE	AE

(a) First DMM application server placement

B	B	B	B	D	D	D	D
F	F	F	F	H	H	H	H
J	J	J	J	L	L	L	L
N	N	N	N	P	P	P	P
R	R	R	R	T	T	T	T
V	V	V	V	X	X	X	X
Z	Z	Z	Z	AB	AB	AB	AB
AD	AD	AD	AD	AF	AF	AF	AF

(b) Second DMM application server placement

Figure 6.2 : Mappings of 64 processes to 16 servers for two DMM applications.

the DMM1. Now, we find the optimal placement for the remainder of the servers as it suits DMM2. We place Q, R, U, V under ToR 4, and U, V, W, X under ToR 5. The resulting placement is shown in Figure 6.6. The cross-rack flows for each column-wise shift are minimized from 8 to 3 for the DMM2. We can reconfigure the network before the shift phase of both DMM applications starts. Since both applications are launched simultaneously, and the matrices are the same size, the processes involved will all finish the broadcast phase and get to the shift phase approximately at the same time. After the shift phase and before the broadcast phase, we reconfigure the network back to the original placement shown in Figure 6.4 which has the optimal placement for the broadcast phase. In this placement, all row-wise broadcasts are intra-rack. Throughout the runtime of both DMMs the network will be reconfigured back and forth between the placement in Figure 6.6 and the placement in Figure 6.6.

Note that the assumption we made for this example does not hold in many cases. The applications can start at different timestamps and overlap at different stages of the algorithm, or they can operate on different matrix sizes.

If two instances of the same application start at a different time, then one of the instances is ahead of the other. For example, in the case of the DMM application, one can be in the broadcast phase and the other in the shift phase or one in the shift phase and the other in the computation phase, etc. For the former case, the network

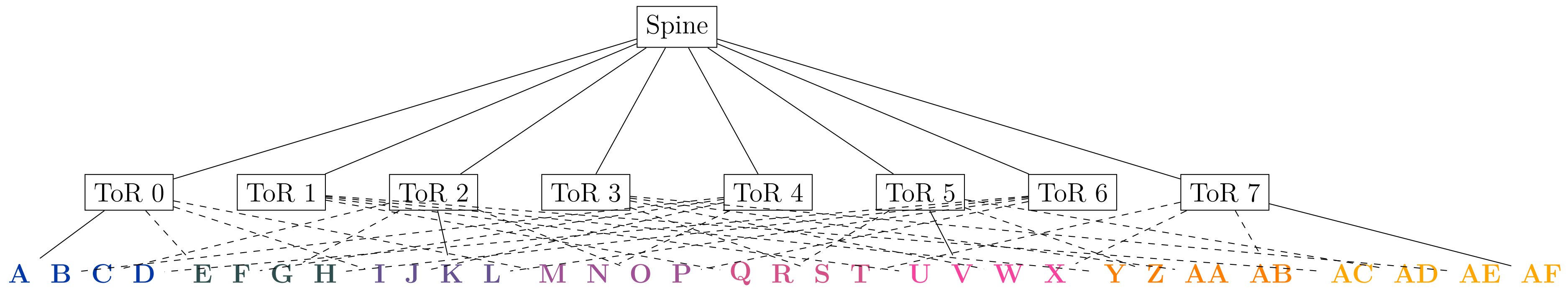

Spine
ToR 0
ToR 1
ToR 2
ToR 3
ToR 4
ToR 5
ToR 6
ToR 7
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z AA AB AC AD AE AF

Figure 6.3 : Best server placement for shift with minimal cross-rack communication.

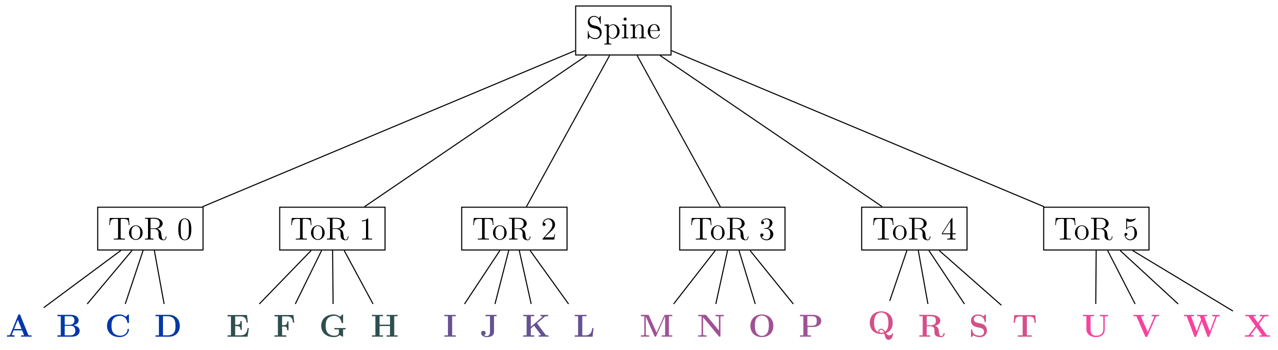
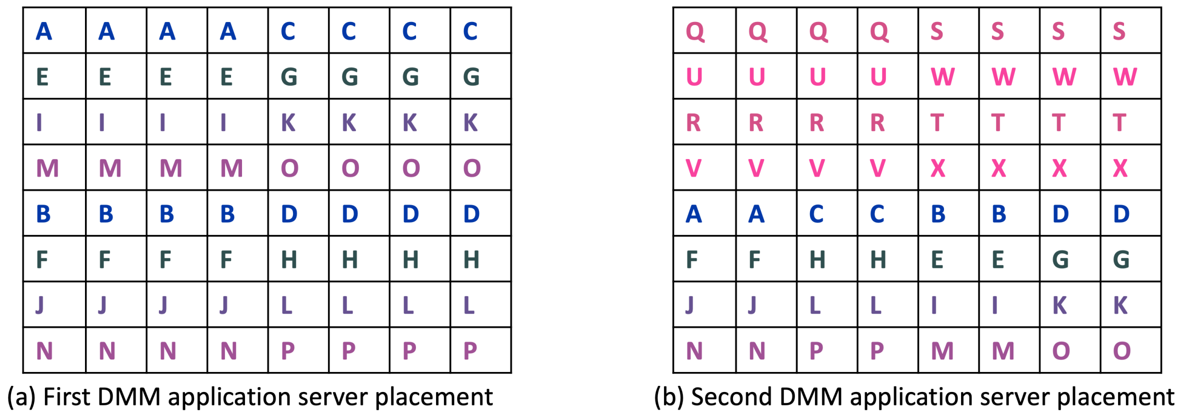


Figure 6.4 : The original server placement in the testbed. A-X represent 24 servers and A-D, E-H, I-L, M-P, Q-T, U-X belong to six physical racks separately.



A	A	A	A	C	C	C	C
E	E	E	E	G	G	G	G
I	I	I	I	K	K	K	K
M	M	M	M	O	O	O	O
B	B	B	B	D	D	D	D
F	F	F	F	H	H	H	H
J	J	J	J	L	L	L	L
N	N	N	N	P	P	P	P

(a) First DMM application server placement

Q	Q	Q	Q	S	S	S	S
U	U	U	U	W	W	W	W
R	R	R	R	T	T	T	T
V	V	V	V	X	X	X	X
A	A	C	C	B	B	D	D
F	F	H	H	E	E	G	G
J	J	L	L	I	I	K	K
N	N	P	P	M	M	O	O

(b) Second DMM application server placement

Figure 6.5 : Mappings of 64 processes to 16 servers for two DMM applications.

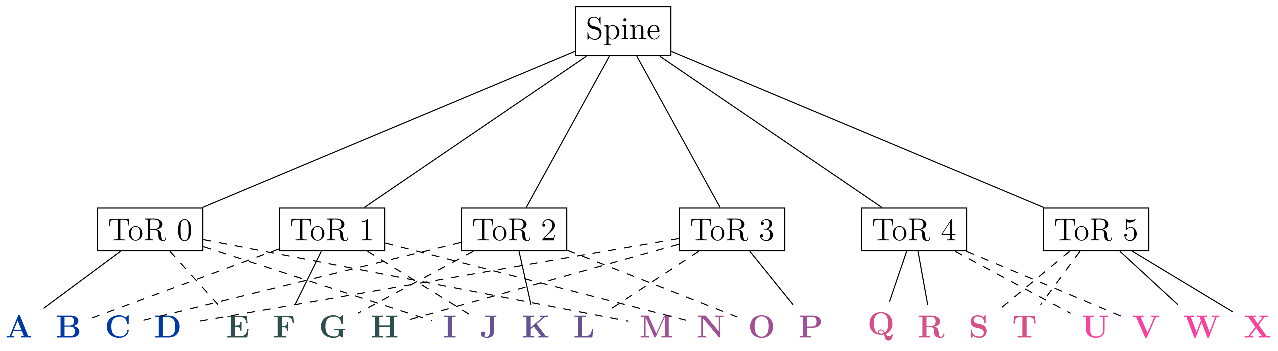


Figure 6.6 : Optimal server placement for shift with minimal cross-rack communication.

should be reconfigured in a way that minimizes the number of row-wise broadcast's cross-rack communication for DMM1 and column-wise shift's cross-rack communication for DMM2, and for the latter, the network should be reconfigured to a placement that has the minimum amount of column-wise shift's cross-rack communication for DMM1.

Finding the joint communication patterns of multiple concurrent applications with overlapping/shared servers can be challenging. By running our static analysis-based solution, we can find the different communication patterns of each application separately, but because the servers are shared between the applications, the applications can not individually reconfigure the network. If they do, it can create: 1) packet loss and 2) create higher cross-rack traffic for the other applications sharing the servers. We need to know which phase each application is at and how long each phase approximately takes. Based on these approximations and each application's starting time, we can have an image of the joint traffic pattern at different times in the data center network. A controller can use these pieces of information to find the optimal placement for all the concurrent applications at different points and reconfigure the network before a new communication pattern starts.

Challenges. One challenge here is how to have an overall image of our network and the communication patterns of the concurrent applications running on it. The question to be solved is "how can we estimate and approximate when certain combined traffic patterns will be seen on our network?". If we could have an abstraction of the program that demonstrates the notion of time with an approximation of how long each stage will take, we could have an automated controller that reconfigures the network based on this information.

One direction that can be explored is temporal logic, which specifies properties/ events occurring over time. Linear Temporal Logic (LTL) expresses properties over a single computation path or run. In LTL, each moment in time has a well-defined successor moment. We can encode the behavior the behavior of applications with LTL. LTL shows the orders of events and do not show the time between events. The extensions of LTL is called Timed Propositional Temporal Logic (TPTL), in which variables are introduced to measure times between two events. Using these types of logic, we can have an abstract image of the network showing different possibilities of concurrent events (phases of the applications). One crucial missing piece is the estimation of the time between events or how long every application phase takes. These problems need to be addressed in future studies.

Bibliography

[1] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song, "Design and evaluation of a real-time url spam filtering service," in *2011 IEEE symposium on security and privacy*, pp. 447–462, IEEE, 2011.

[2] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the netflix prize," in *International conference on algorithmic applications in management*, pp. 337–348, Springer, 2008.

[3] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 98–109, 2011.

[4] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[5] T. Chiba and T. Onodera, "Workload characterization and optimization of tpc-h queries on apache spark," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 112–121, IEEE, 2016.

[6] "Tpc benchmark h," 2001. `http://www.tpc.org/tpch/`.

[7] A. Shieh, S. Kandula, A. G. Greenberg, C. Kim, and B. Saha, "Sharing the data center network.," in *NSDI*, vol. 11, pp. 23–23, 2011.

[8] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[9] P. G. Raponi, F. Petrini, R. Walkup, and F. Checconi, “Characterization of the communication patterns of scientific applications on blue gene/p,” in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pp. 1017–1024, IEEE, 2011.

[10] N. R. Adiga, G. Almási, G. S. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, *et al.*, “An overview of the bluegene/l supercomputer,” in *SC’02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pp. 60–60, IEEE, 2002.

[11] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, “Characterization of mpi fsage on a production supercomputer,” in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 386–400, IEEE, 2018.

[12] R. Thakur, R. Rabenseifner, and W. Gropp, “Optimization of collective communication operations in mpich,” *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[13] S. Sistare, R. Vandevaart, and E. Loh, “Optimization of mpi collectives on clusters of large-scale smp’s,” in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, pp. 23–es, 1999.

[14] K. Hasanov and A. Lastovetsky, “Hierarchical redesign of classic mpi reduction algorithms,” *The Journal of Supercomputing*, vol. 73, no. 2, pp. 713–725, 2017.

[15] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, “Performance analysis of mpi collective operations,” *Cluster Computing*, vol. 10, no. 2, pp. 127–143, 2007.

[16] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, “Vl2: A scalable and flexible data center network,”

in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pp. 51–62, 2009.

[17] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 63–74, ACM, 2008.

[18] M. McBride and O. Komolafe, "Multicast in the data center overview," internet-draft, RFC Editor, February 2019.

[19] B. D. Wozniak, F. D. Witherden, F. P. Russell, P. E. Vincent, and P. H. Kelly, "Gimmik—generating bespoke matrix multiplication kernels for accelerators: Application to high-order computational fluid dynamics," *Computer Physics Communications*, vol. 202, pp. 12–22, 2016.

[20] S. Itoh, P. Ordejón, and R. M. Martin, "Order-n tight-binding molecular dynamics on parallel computers," *Computer physics communications*, vol. 88, no. 2-3, pp. 173–185, 1995.

[21] R. Furrer, M. G. Genton, and D. Nychka, "Covariance tapering for interpolation of large spatial datasets," *Journal of Computational and Graphical Statistics*, vol. 15, no. 3, pp. 502–523, 2006.

[22] "Mpi broadcast and collective communication," 2019. `http://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/`.

[23] D. Li, Y. Li, J. Wu, S. Su, and J. Yu, "Esm: Efficient and scalable data center multicast routing," *IEEE/ACM Transactions on Networking (TON)*, vol. 20, no. 3, pp. 944–955, 2012.

[24] X. Li and M. J. Freedman, "Scaling ip multicast on datacenter topologies," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pp. 61–72, ACM, 2013.

[25] M. Shahbaz, L. Suresh, J. Rexford, N. Feamster, O. Rottenstreich, and M. Hira, "Elmo: Source routed multicast for public clouds," in *Proceedings of the ACM Special Interest Group on Data Communication*, pp. 458–471, 2019.

[26] Y. Vigfusson, H. Abu-Libdeh, M. Balakrishnan, K. Birman, R. Burgess, G. Chockler, H. Li, and Y. Tock, "Dr. multicast: Rx for data center communication scalability," in *Proceedings of the 5th European conference on Computer systems*, pp. 349–362, ACM, 2010.

[27] "Nack-oriented reliable multicast (norm) transport protocol," 2009. `https://tools.ietf.org/html/rfc5740`.

[28] J. Widmer and M. Handley, "Extending equation-based congestion control to multicast applications," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 275–285, 2001.

[29] J. Widmer and M. Handley, "Tcp-friendly multicast congestion control (tfmcc): Protocol specification," tech. rep., RFC 4654, August, 2006.

[30] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable application layer multicast," in *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 205–217, 2002.

[31] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "Splitstream: high-bandwidth multicast in cooperative environments," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 298–313, 2003.

[32] A. Das, I. Gupta, and A. Motivala, "Swim: Scalable weakly-consistent infection-style process group membership protocol," in *Proceedings International Conference on Dependable Systems and Networks*, pp. 303–312, IEEE, 2002.

[33] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, *et al.*, "Overcast: reliable multicasting with on overlay network," in *Proceedings of the 4th Conference*

on Symposium on Operating System Design & Implementation-Volume 4, p. 14, USENIX Association, 2000.

[34] J. Cao, C. Guo, G. Lu, Y. Xiong, Y. Zheng, Y. Zhang, Y. Zhu, C. Chen, and Y. Tian, "Datacast: A scalable and efficient reliable group data delivery service for data centers," *IEEE Journal on Selected Areas in Communications*, vol. 31, no. 12, pp. 2632–2645, 2013.

[35] "Murder: Fast datacenter code deploys using BitTorrent," 2010. https://github.com/lg/murder.

[36] D. Basin, K. Birman, I. Keidar, and Y. Vigfusson, "Sources of instability in data center multicast," in *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*, pp. 32–37, ACM, 2010.

[37] "Power, Pollution and the Internet." http://www.nytimes.com/2012/09/23/technology/data-centers-waste-vast-amounts-of-energy-belying-industry-image.html.

[38] E. Weingärtner, R. Glebke, M. Lang, and K. Wehrle, "Building a modular bittorrent model for ns-3," in *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, pp. 337–344, 2012.

[39] Z. Cai, Z. J. Gao, S. Luo, L. L. Perez, Z. Vagena, and C. Jermaine, "A comparison of platforms for implementing and running very large scale machine learning algorithms," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 1371–1382, ACM, 2014.

[40] T. Mitchell, "20 newsgroups," 1999. `http://kdd.ics.uci.edu/databases/20newsgroups/20newsgroups.html`.

[41] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, pp. 3111–3119, 2013.

[42] S. Dutta, M. Fahim, F. Haddadpour, H. Jeong, V. Cadambe, and P. Grover, "On the optimal recovery threshold of coded matrix multiplication," *arXiv preprint arXiv:1801.10292*, 2018.

[43] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, "Compressed linear algebra for large-scale machine learning," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 27, no. 5, pp. 719–744, 2018.

[44] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "Systemml: Declarative machine learning on mapreduce," in *2011 IEEE 27th International Conference on Data Engineering*, pp. 231–242, IEEE, 2011.

[45] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman, "The architecture of scidb," in *International Conference on Scientific and Statistical Database Management*, pp. 1–16, Springer, 2011.

[46] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

[47] G. C. Fox, S. W. Otto, and A. J. Hey, "Matrix algorithms on a hypercube i: Matrix multiplication," *Parallel computing*, vol. 4, no. 1, pp. 17–31, 1987.

[48] M. Lopes and C. Iseli, "Multiplication matriciel." `https://github.com/Matoran/multiplicationMatriciel`, 2018.

[49] O. M. Team, "Open MPI a high performance message passing library," 2018. `https://www.open-mpi.org`.

[50] J. Dugan, E. Seth, B. A. Mah, J. Poskanzer, and K. Prabhu, "iperf - the ultimate speed test tool for tcp, udp and sctp," 2003. `https://iperf.fr`.

[51] L. Lamport, “The part-time parliament,” *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[52] L. Lamport *et al.*, “Paxos made simple,” *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[53] L. Lamport, “Fast paxos,” *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[54] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone, “Ring paxos: A high-throughput atomic broadcast protocol,” in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pp. 527–536, IEEE, 2010.

[55] “Libpaxos is a collection of open source implementations of the paxos algorithm,” 2013. `http://libpaxos.sourceforge.net`.

[56] Y. Amir and J. Kirsch, “Paxos for system builders: An overview,” in *Workshop on Large-Scale Distributed Systems and Middleware (LADIS 2008), Yorktown, NY*, 2008.

[57] S. Das, A. Rahbar, X. C. Wu, Z. Wang, W. Wang, A. Chen, and T. Ng, “Shufflecast: An optical, data-rate agnostic and low-power multicast architecture for next-generation compute clusters,” *arXiv preprint arXiv:2104.09680*, 2021.

[58] “Fs (fiberstore) - leading communication systems integrator and optical solutions provider for data centers,” 2019. `www.fs.com`.

[59] “Finisar,” 2019. http://finisar.com.

[60] H. S. Stone, “Parallel processing with the perfect shuffle,” *IEEE transactions on computers*, vol. 100, no. 2, pp. 153–161, 1971.

[61] “Welcome to ryu the network operating system(nos).” `https://ryu.readthedocs.io/en/latest/`.

[62] X. S. Sun, Y. Xia, S. Dzinamarira, X. S. Huang, D. Wu, and T. E. Ng, “Republic: Data multicast meets hybrid rack-level interconnections in data center,” in

2018 IEEE 26th International Conference on Network Protocols (ICNP), pp. 77–87, IEEE, 2018.

[63] G. rey Fox *et al.*, "Solving problems on concurrent processors," 1988.

[64] "Libfastpaxos." `https://sourceforge.net/projects/libpaxos/files/LibFastPaxos/src-rev-17/`.

[65] "Libpaxos3." `https://sourceforge.net/projects/libpaxos/files/LibPaxos3/`.

[66] W. M. Mellette, R. Das, Y. Guo, R. McGuinness, A. C. Snoeren, and G. Porter, "Expanding across time to deliver bandwidth efficiency and low latency," *arXiv e-prints*, p. arXiv:1903.12307, Mar 2019.

[67] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. Ng, M. Kozuch, and M. Ryan, "c-through: Part-time optics in data centers," in *ACM SIGCOMM Computer Communication Review*, vol. 40, pp. 327–338, ACM, 2010.

[68] M. Ghobadi, R. Mahajan, A. Phanishayee, N. Devanur, J. Kulkarni, G. Ranade, P.-A. Blanche, H. Rastegarfar, M. Glick, and D. Kilper, "Projector: Agile reconfigurable data center interconnect," in *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 216–229, ACM, 2016.

[69] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, "Helios: a hybrid electrical/optical switch architecture for modular data centers," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 339–350, 2010.

[70] N. Hamedazimi, Z. Qazi, H. Gupta, V. Sekar, S. R. Das, J. P. Longtin, H. Shah, and A. Tanwer, "Firefly: A reconfigurable wireless data center fabric using free-space optics," in *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 319–330, ACM, 2014.

[71] W. M. Mellette, R. McGuinness, A. Roy, A. Forencich, G. Papen, A. C. Snoeren, and G. Porter, "Rotornet: A scalable, low-complexity, optical datacenter network," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 267–280, ACM, 2017.

[72] G. Porter, R. Strong, N. Farrington, A. Forencich, P. Chen-Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat, "Integrating microsecond circuit switching into the data center," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, (New York, NY, USA), pp. 447–458, ACM, 2013.

[73] Y. J. Liu, P. X. Gao, B. Wong, and S. Keshav, "Quartz: a new design element for low-latency dcns," in *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 283–294, ACM, 2014.

[74] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson, "F10: A fault-tolerant engineered network," in *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pp. 399–412, 2013.

[75] H. Liu, F. Lu, A. Forencich, R. Kapoor, M. Tewari, G. M. Voelker, G. Papen, A. C. Snoeren, and G. Porter, "Circuit switching under the radar with reactor," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, (Seattle, WA), pp. 1–15, USENIX Association, 2014.

[76] A. Chatzieleftheriou *et al.*, "Larry: Practical network reconfigurability in the data center," in *NSDI*, USENIX, 2018.

[77] X. Zhou, Z. Zhang, Y. Zhu, Y. Li, S. Kumar, A. Vahdat, B. Y. Zhao, and H. Zheng, "Mirror mirror on the ceiling: Flexible wireless links for data centers," *ACM SIGCOMM CCR*, vol. 42, no. 4, pp. 443–454, 2012.

[78] S. Kandula, J. Padhye, and P. Bahl, "Flyways to de-congest data center networks," 2009.

[79] K. Chen, A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang, X. Wen, and Y. Chen, "Osa: An optical switching architecture for data center networks with unprecedented flexibility," *IEEE/ACM Transactions on Networking*, vol. 22, no. 2, pp. 498–511, 2014.

[80] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," *ACM SIGCOMM computer communication review*, vol. 45, no. 4, pp. 183–197, 2015.

[81] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," *SIGCOMM Comput. Commun. Rev.*, vol. 45, pp. 123–137, Aug. 2015.

[82] T. Benson *et al.*, "Network traffic characteristics of data centers in the wild," in *IMC*, ACM, 2010.

[83] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pp. 65–72, ACM, 2009.

[84] P. Bodík *et al.*, "Surviving failures in bandwidth-constrained datacenters," in *SIGCOMM*, ACM, 2012.

[85] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 455–466, 2014.

[86] "Hdfs architecture." https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[87] M. Walraed-Sullivan, A. Vahdat, and K. Marzullo, "Aspen trees: balancing data center fault tolerance, scalability and cost," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pp. 85–96, 2013.

[88] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "Bcube: a high performance, server-centric network architecture for modular data centers," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 63–74, 2009.

[89] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell: a scalable and fault-tolerant network structure for data centers," in *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 75–86, ACM, 2008.

[90] P. Bakopoulos, K. Christodoulopoulos, G. Landi, M. Aziz, E. Zahavi, D. Gallico, R. Pitwon, K. Tokas, I. Patronas, M. Capitani, *et al.*, "Nephele: An end-to-end scalable and dynamically reconfigurable optical architecture for application-aware sdn cloud data centers," *IEEE Communications Magazine*, vol. 56, no. 2, pp. 178–188, 2018.

[91] "Sailing through the data deluge." https://rockleyphotonics.com/wp-content/uploads/2019/02/Rockley-Photonics-Sailing-through-the-Data-Deluge.pdf.

[92] H. Ballani, P. Costa, R. Behrendt, D. Cletheroe, I. Haller, K. Jozwik, F. Karinou, S. Lange, K. Shi, B. Thomsen, *et al.*, "Sirius: A flat datacenter network with nanosecond optical switching," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pp. 782–797, 2020.

[93] K. A. Clark, D. Cletheroe, T. Gerard, I. Haller, K. Jozwik, K. Shi, B. Thomsen, H. Williams, G. Zervas, H. Ballani, *et al.*, "Synchronous subnanosecond clock and data recovery for optically switched data centres using clock phase caching," *Nature Electronics*, vol. 3, no. 7, pp. 426–433, 2020.

[94] S. Das, W. Wang, and T. E. Ng, "Towards all-optical circuit-switched datacenter network cores: The case for mitigating traffic skewness at the edge," in *Proceedings*

of the ACM SIGCOMM 2021 Workshop on Optical Systems, pp. 1–5, 2021.

[95] T. Gerard, K. Clark, A. Funnell, K. Shi, B. Thomsen, P. Watts, K. Jozwik, I. Haller, H. Williams, P. Costa, *et al.*, "Fast and uniform optically-switched data centre networks enabled by amplitude caching," in *2021 Optical Fiber Communications Conference and Exhibition (OFC)*, pp. 1–3, IEEE, 2021.

[96] S. K. Moore, "Another step toward the end of moore's law: Samsung and tsmc move to 5-nanometer manufacturing-[news]," *IEEE Spectrum*, vol. 56, no. 6, pp. 9–10, 2019.

[97] V. Shrivastav, A. Valadarsky, H. Ballani, P. Costa, K. S. Lee, H. Wang, R. Agarwal, and H. Weatherspoon, "Shoal: A network architecture for disaggregated racks," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, (Boston, MA), pp. 255–270, USENIX Association, 2019.

[98] V. Jalaparti *et al.*, "Network-aware scheduling for data-parallel jobs: Plan when you can," *SIGCOMM*, 2015.

[99] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *ACM SIGCOMM Computer Communication Review*, vol. 43, pp. 231–242, ACM, 2013.

[100] C. Zimmer, S. Gupta, S. Atchley, S. S. Vazhkudai, and C. Albing, "A multi-faceted approach to job placement for improved performance on extreme-scale systems," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1015–1025, IEEE, 2016.

[101] X. Tang, H. Wang, X. Ma, N. El-Sayed, J. Zhai, W. Chen, and A. Aboulnaga, "Spread-n-share: improving application performance and cluster throughput with resource-aware job placement," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2019.

[102] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, “Quincy: fair scheduling for distributed computing clusters,” in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 261–276, 2009.

[103] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, “Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling,” in *Proceedings of the 5th European conference on Computer systems*, pp. 265–278, 2010.

[104] X. Meng, V. Pappas, and L. Zhang, “Improving the scalability of data center networks with traffic-aware virtual machine placement,” in *2010 Proceedings IEEE INFOCOM*, pp. 1–9, IEEE, 2010.

[105] S. Blagodurov, A. Fedorova, E. Vinnik, T. Dwyer, and F. Hermenier, “Multi-objective job placement in clusters,” in *SC’15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, IEEE, 2015.

[106] G. Fox, S. Otto, and A. Hey, “Matrix algorithms on a hypercube i: Matrix multiplication,” *Parallel Computing*, vol. 4, no. 1, pp. 17 – 31, 1987.

[107] R. A. Van De Geijn and J. Watts, “Summa: Scalable universal matrix multiplication algorithm,” *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[108] E. Solomonik and J. Demmel, “Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms,” in *European Conference on Parallel Processing*, pp. 90–109, Springer, 2011.

[109] D. Wu, A. Chen, and T. E. Ng, “The case for a rackless data center network architecture,” in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pp. 93–95, 2018.

[110] “Apache hadoop: Capacity scheduler.” https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[111] "Apache hadoop: Fair scheduler." https://hadoop.apache.org/docs/r2.7.4/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[112] "Apache hadoop yarn project." http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[113] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types.," in *Nsdi*, vol. 11, pp. 24–24, 2011.

[114] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder, "Heuristics for vector bin packing," *research. microsoft. com*, 2011.

[115] C. Doulkeridis and K. NØrvåg, "A survey of large-scale analytical query processing in mapreduce," *The VLDB Journal*, vol. 23, no. 3, pp. 355–380, 2014.

[116] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pp. 1–14, 2011.

[117] M. Katyal and A. Mishra, "A comparative study of load balancing algorithms in cloud computing environment," *arXiv preprint arXiv:1403.6918*, 2014.

[118] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, 2017.

[119] C.-H. Hsu, Q. Deng, J. Mars, and L. Tang, "Smoothoperator: Reducing power fragmentation and improving power utilization in large-scale datacenters," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, (New York, NY, USA), pp. 535–548, ACM, 2018.

[120] "Edge 640 optical circuit switch." https://www.calient.net/products/edge640-optical-circuit-switch.

[121] "Polatis series 7000 384x384." https://www.polatis.com/series-7000-384x384-port-software-controlled-optical-circuit-switch-sdn-enabled asp.

[122] "Introducing data center fabric, the next-generation facebook data center network."

[123] "Core and pod data center design." http://go.bigswitch.com/rs/974-WXR-561/images/Core-and-Pod%20Overview.pdf.

[124] "Specifying data center it pod architectures." https://www.apc.com/salestools/WTOL-AHAPRN/WTOL-AHAPRN_R0_EN.pdf.

[125] "Ibm prefabricated modular data center." https://www.ibm.com/us-en/marketplace/prefabricated-modular-data-center.

[126] K. Cooper and L. Torczon, *Engineering a compiler.* Elsevier, 2011.

[127] A. R. Dakshinamurthy, "A compiler-based framework for automatic extraction of program skeletons for exascale hardware/software co-design," 2013.

[128] M. Sottile, J. Dagit, D. Zhang, G. Hendry, and D. Dechev, "Static analysis techniques for semiautomatic synthesis of message passing software skeletons," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 26, no. 1, pp. 1–24, 2015.

[129] K. D. Cooper, K. Kennedy, and L. Torczon, "Compilers," in *Encyclopedia of Physical Science and Technology (Third Edition)* (R. A. Meyers, ed.), pp. 433–442, New York: Academic Press, third edition ed., 2003.

[130] "MPI_Comm_split function ." https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[131] P. Cousot and R. Cousot, "Basic concepts of abstract interpretation," in *Building the Information Society*, pp. 359–366, Springer, 2004.

[132] B. Blanchet, "Introduction to abstract interpretation," *lecture script*, 2002.

[133] "Hasse diagram." [Online; accessed 4-September-2021].

[134] P. Havlak and K. Kennedy, "Experience with interprocedural analysis of array side effects," in *Supercomputing'90: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pp. 952–961, IEEE, 1990.

[135] P. Havlak and K. Kennedy, "An implementation of interprocedural bounded regular section analysis," *IEEE Transactions on Parallel and Distributed systems*, vol. 2, no. 3, pp. 350–360, 1991.

[136] L. P. Huse, "Collective communication on dedicated clusters of workstations," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pp. 469–476, Springer, 1999.

[137] A. Cichocki, "Era of big data processing: A new approach via tensor networks and tensor decompositions," *arXiv preprint arXiv:1403.2048*, 2014.

[138] A. Cichocki, R. Zdunek, A. H. Phan, and S.-i. Amari, *Nonnegative matrix and tensor factorizations: applications to exploratory multi-way data analysis and blind source separation.* John Wiley & Sons, 2009.

[139] A. Cichocki, D. Mandic, C. Caiafa, A. Phan, G. Zhou, Q. Zhao, and L. De Lathauwer, "Multiway component analysis: Tensor decompositions for signal processing applications," *IEEE Signal Processing Magazine*, 2014.

[140] A. Cichocki, "Tensor decompositions: a new concept in brain data analysis?," *arXiv preprint arXiv:1305.0395*, 2013.

[141] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM review*, vol. 51, no. 3, pp. 455–500, 2009.

[142] P. M. Kroonenberg, *Applied multiway data analysis*, vol. 702. John Wiley & Sons, 2008.

[143] A. Smilde, R. Bro, and P. Geladi, *Multi-way analysis: applications in the chemical sciences*. John Wiley & Sons, 2005.

[144] W. Hackbusch, *Tensor spaces and numerical tensor calculus*, vol. 42. Springer, 2012.

[145] L. Grasedyck, "Hierarchical singular value decomposition of tensors," *SIAM Journal on Matrix Analysis and Applications*, vol. 31, no. 4, pp. 2029–2054, 2010.

[146] W. Hackbusch and S. Kühn, "A new scheme for the tensor representation," *Journal of Fourier analysis and applications*, vol. 15, no. 5, pp. 706–722, 2009.

[147] D. Kressner and C. Tobler, "htucker—a matlab toolbox for tensors in hierarchical tucker format," *Mathicse, EPF Lausanne*, 2012.

[148] A. Rodgers, "Htucker-mpi." https://github.com/akrodger/htucker-mpi, 2019.

[149] A. Rodgers and D. Venturi, "Stability analysis of hierarchical tensor methods for time-dependent pdes," *Journal of Computational Physics*, vol. 409, p. 109341, 2020.

[150] "Heat equation." https://en.wikipedia.org/wiki/Heat_equation, 2021.

[151] M. Schordan and D. Quinlan, "A source-to-source architecture for user-defined optimizations," in *Joint Modular Languages Conference*, pp. 214–223, Springer, 2003.

[152] S. Sreepathi, M. L. Grodowitz, R. Lim, P. Taffet, P. C. Roth, J. Meredith, S. Lee, D. Li, and J. Vetter, "Application characterization using oxbow toolkit and pads infrastructure," in *2014 Hardware-Software Co-Design for High Performance Computing*, pp. 55–63, IEEE, 2014.

[153] J. Vetter and C. Chambreau, "mpip: Lightweight, scalable mpi profiling," 2005.

[154] P. C. Roth, "Improved accuracy for automated communication pattern characterization using communication graphs and aggressive search space pruning," in *Programming and Performance Visualization Tools*, pp. 38–55, Springer, 2017.

[155] P. C. Roth, J. S. Meredith, and J. S. Vetter, "Automated characterization of parallel application communication patterns," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 73–84, 2015.

[156] P. C. Roth, K. Huck, G. Gopalakrishnan, and F. Wolf, "Using deep learning for automated communication pattern characterization: Little steps and big challenges," in *Programming and Performance Visualization Tools*, pp. 265–272, Springer, 2017.

[157] B. F. Cornea, J. Slawinski, J. Bourgeois, and V. Sunderam, "Towards adapting parallel programs to different platforms: Identifying interaction patterns," in *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pp. 352–359, IEEE, 2013.

[158] R. Preissl, M. Schulz, D. Kranzlmüller, B. R. de Supinski, and D. J. Quinlan, "Using mpi communication patterns to guide source code transformations," in *International Conference on Computational Science*, pp. 253–260, Springer, 2008.