OPTIMIZING PERFORMANCE OF DISTRIBUTED MEMORY WORKLOADS FOR HIGH PERFORMANCE COMPUTING (HPC) AND DATA CENTER NETWORKS

by

MD NAHID NEWAZ

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

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND INFORMATICS

2024

Oakland University
Rochester, Michigan

Doctoral Advisory Committee:

Guangzhi Qu, Ph.D., Chair
Lunjin Lu, Ph.D.
Md Atiqul Mollah, Ph.D.
Ilias Cholis, Ph.D.

© by Md Nahid Newaz, 2024
All rights reserved

To my beloved mother, Mst Kohinur Sultana,

whose unwavering support, boundless love, tireless dedication and sacrifices have shaped my journey. I am forever grateful.

And

To God,

the source of my strength and wisdom. May this achievement reflect Your grace and love in my life.

ACKNOWLEDGMENTS

I would like to express my heartfelt appreciation for the extensive mentorship provided by my PhD advisor, Guangzhi Qu, Ph.D. His guidance and support have been invaluable throughout my research journey. I also extend my sincere gratitude to my former advisors, Hua Ming, Ph.D., and Md Atiqul Mollah, Ph.D., whose insights and encouragement laid a strong foundation for my academic development.

I am grateful to my dissertation committee members, Lunjin Lu, Ph.D., and Ilias Cholis, Ph.D., for their constructive feedback and contributions, which significantly enhanced the quality of this work.

Additionally, I would like to acknowledge my internship mentors, Sayan Ghosh, Ph.D., and Nathan R. Tallent, Ph.D., at the Pacific Northwest National Laboratory. Their support and expertise during my internship were crucial in shaping my research perspective and honing my skills.

Finally, I would like to thank my family and friends for their unwavering support and encouragement throughout this journey. Their belief in me has been a constant source of motivation.

Md Nahid Newaz

ABSTRACT

OPTIMIZING PERFORMANCE OF DISTRIBUTED MEMORY WORKLOADS FOR HIGH PERFORMANCE COMPUTING (HPC) AND DATA CENTER NETWORKS

by

Md Nahid Newaz

Adviser: Guangzhi Qu, Ph.D.

Today's high-performance computing (HPC) and large-scale data center systems are constructed by interconnecting thousands of computing nodes, storage devices, hardware accelerators, and I/O devices. Due to the rapid growth in the scale of these systems, along with their complex interconnect topologies and the need to support diverse applications, the interconnection network has become one of the major bottlenecks for performance, efficiency, and resource utilization. My research focuses on developing novel, efficient, and scalable routing methods to reduce latency and optimized scheduling methods to improve throughput and system utilization. I validated these proposed approaches through various experimental studies using state-of-the-art benchmarks, as well as real scientific and synthetic applications. The goal is to facilitate the future adoption of these approaches in next-generation HPC and data center network systems.

TABLE OF CONTENTS

ACKNOWLEDGMENTS iv

ABSTRACT v

LIST OF TABLES x

LIST OF FIGURES xi

CHAPTER ONE
INTRODUCTION 1

CHAPTER TWO
BACKGROUND 4

2.1. High Performance Computing (HPC) Systems 4

2.2. Large Scale Data Center System 5

2.3. Interconnection Networks 5

2.3.1. Fat Tree Topology 6

2.3.2. Dragonfly Topology 7

2.3.3. Dragonfly+ Topology 8

2.3.4. Jellyfish/Random Regular Graph(RRG) Topology 9

2.4. Routing Methods in HPC and Data center Networks 12

2.4.1. Fat Tree Routing 13

2.4.2. Dragonfly and Dragonfly+ Routing 13

2.4.3. Jellyfish/RRG Routing 15

2.5. Scheduling Methods in HPC and Data center Networks 16

2.6. Performance Evaluation Metrics 17

TABLE OF CONTENTS—Continued

CHAPTER THREE
ROUTING OPTIMIZATION FOR DRAGONFLY+/MEGAFLY TOPOLOGY 19

3.1. Optimized Routing For Dragonfly+/Megafly Topology 20

3.1.1. K-path-adaptive Piggyback Routing (KPR) 21

3.1.2. K-path UGAL with Global Congestion Notification (KU-GCN) 22

3.2. Performance Evaluation 22

3.2.1. Topology Configurations 23

3.2.2. Synthetic Traces Results 24

3.2.3. Benchmarks/Applications Results 25

3.2.4. Discussion 27

CHAPTER FOUR
OPTIMAL PATH SELECTION FOR MULTIPATH ROUTING IN RANDOM TOPOLOGY 28

4.1. Problem Motivation 29

4.2. Proposed Optimized Solution 30

4.2.1. Parameter Selection 31

4.2.2. Optimized path selection Algorithm 32

4.3. Performance Evaluation 34

4.3.1. Experimental Results 34

4.3.2. Synthetic Trace Results 38

4.3.3. Application Benchmarks Results 40

4.3.4. Result Summary 41

4.3.5. Discussion 42

CHAPTER FIVE
PERFORMANCE COMPARISON BETWEEN STRUCTURED AND IRREGULAR NETWORK TOPOLOGIES 44

5.1. Methodology 44

5.1.1. Constructing Jellyfish RRG 45

5.1.2. Cost Estimation 46

5.1.3. Application Scenarios 49

5.2. Performance Comparison 51

5.2.1. Impact of routing schemes on the topologies 52

5.2.2. Analyzing stalls and hops in Fat tree and equivalent Jellyfish 53

5.2.3. Analyzing stalls and hops in Dragonfly and equivalent Jellyfish 53

5.2.4. Analyzing MPI Communication Times 55

5.2.5. Scalability of graph applications on Jellyfish and Dragonfly 55

5.2.6. Overall performance assessment 56

5.2.7. Discussion 58

CHAPTER SIX
OPTIMIZED SCHEDULING FOR DISTRIBUTED MEMORY GRAPH APPLICATIONS ON DRAGONFLY TOPOLOGY 64

TABLE OF CONTENTS—Continued

6.1. Process remappings for graph workloads 64

6.1.1. Vertex-based graph partitioning 65

6.1.2. Process remapping heuristics considering graph structure 66

6.2. Simulation and Empirical analysis 70

6.2.1. Simulations results on Dragonfly topology with PPN = 16 and 32 70

6.2.2. Analysis of empirical results from NERSC Perlmutter 75

6.2.3. Discussion 78

CHAPTER SEVEN
CONCLUSION 79

REFERENCES 80

LIST OF TABLES

Table 4.1	Communication time speedup (in percent) of optimized k-path selection algorithm using Uniform multi-path routing	41
Table 4.2	Communication time speedup (in percent) of optimized k-path selection algorithm using Adaptive multi-path routing	42
Table 5.1	Topology configurations used in the experiments.(First two configurations are for baseline experiments and others are for scalability experiments)	45
Table 5.2	Cost estimation of Dragonfly and Fat Tree in comparison to equivalent Jellyfish RRG topologies with reasonable assumptions	48
Table 5.3	Graphs from Suitesparse [1] used in our evaluation.(Real World Graphs, Random Geometric Graphs(RRG) and Stochastic Kronecker Graphs)	50
Table 5.4	Variability of number of iterations to convergence in graph clustering application on 256–1024 MPI ranks	56
Table 6.1	Properties of input graphs and corresponding process graphs used in empirical studies on 1024 processes (PPN=16, 32).	71
Table 6.2	Properties of input graphs and corresponding process graphs used in simulations on 512 processes (PPN=16, 32).	71

LIST OF FIGURES

Figure 2.1	Frontier Supercomputer at Oak Ridge National Laboratory (Source: ornl.gov) [2]	6
Figure 2.2	3-level fat tree topology with 2 sub-trees and each sub-tree has 4 leaf and 4 aggregate switches	7
Figure 2.3	Dragonfly topology with 3 groups and 5 switches in each group	9
Figure 2.4	Dragonfly+ topology with 5 groups and 2 leaf and 2 spine switches in each group mfly(2,5)	10
Figure 2.5	A Random Regular Graph consisting of 16 switches and 32 server endpoints	11
Figure 3.1	Communication Performance of 1024-rank synthetic workloads on megafly topologies	24
Figure 3.2	Communication Performance of NAS benchmark applications on megafly networks	26
Figure 4.1	Distribution of path lengths of an *RRG* $(96, 13)$ topology	35
Figure 4.2	Maximum Link Loads of All-to-All traffic pattern over multiple paths per communication in RRG	36
Figure 4.3	Communication Performance of 1024-rank synthetic workloads on RRG (96,13) topology	39
Figure 4.4	Communication Performance of NAS benchmark applications on RRG (96,13) topology	40
Figure 5.1	RRG(153,24) and RRG(120,38) graph statistics using Gephi [3]	46
Figure 5.2	Mean pair-wise message volumes (black denotes zero bytes exchanged) on 1024 processes	49

Figure 5.3	Network routing schemes and impact on observed stall queue lengths.	52
Figure 5.4	Hops/Stalls – Jellyfish vs. Fat Tree on 1024 MPI ranks	59
Figure 5.5	Hops/Stalls – Jellyfish vs. Dragonfly on 1024 MPI ranks	60
Figure 5.6	MPI times (X-axis) of fixed neighborhood applications on 1024 ranks – LULESH and Sweep3D	61
Figure 5.7	MPI times (X-axis) of graph applications on 1024 ranks – BFS, Clustering and Counting	62
Figure 5.8	Scalability of Jellyfish(AM) vs. Dragonfly(PAR) for graph applications in terms of queue lengths	62
Figure 5.9	Scalability of Jellyfish(AM) vs. Dragonfly(PAR) for graph applications in terms of estimated times	63
Figure 5.10	Percentage improvement of MPI time in RRG compared to relevant Dragonfly and Fat Tree	63
Figure 6.1	Generating *weighted* process graph from an input graph after vertex-based partitioning	65
Figure 6.2	High-level process remapping strategy, using local rearrangement and global reordering	67
Figure 6.3	Process Remapping generation time (secs.) between relatively small and large inputs.	69
Figure 6.4	Simulation Results with 512 MPI processes, PPN = 16	72
Figure 6.5	Simulation Results with 512 MPI processes, PPN = 32	72

Figure 6.6 Packet-level network hops and stalls analysis of logn18 for PPN = 16 73

Figure 6.7 Packet-level network hops and stalls analysis of logn18 for PPN = 32 74

Figure 6.8 Application % performance improvements on NERSC Perlmutter with PPN = 32 and 16 75

Figure 6.9 Distribution of request/response paused cycles on Slingshot NIC counters across 32 nodes 77

CHAPTER ONE

INTRODUCTION

The rapid advancement of computational technologies has created an unprecedented demand for processing power and data handling capabilities, particularly in fields such as scientific research, big data analytics, and artificial intelligence. To meet these escalating computational demands, efficient systems and infrastructures are necessary for executing workloads in parallel. High-Performance Computing (HPC) and data center systems facilitate these requirements. In such systems, thousands of computing nodes, storage devices, specialized hardware accelerators, and I/O devices are interconnected using high-bandwidth, low-latency interconnection network architectures.

With the rapid advancements in processing hardware, such as next-generation CPUs and GPUs, the overall performance of these systems is increasingly bottlenecked by the communication capabilities between interconnected components. Given a particular network topology or the characteristics of distributed memory applications, optimized routing methods can reduce communication latency, thereby improving system performance in terms of latency and throughput.

As resources in HPC and data centers are costly, ensuring optimized and efficient utilization of hardware resources—such as processing cores and memory—as well as network resources like bandwidth is essential for achieving higher system throughput and lower network contention. Therefore, novel and optimized routing methods are needed to reduce latency for parallel workloads, while improved scheduling methods are required to minimize network contention and maximize the utilization of expensive resources.

My research focuses on performance optimization of various distributed memory workloads for HPC and data center networks. Performance optimization includes

reducing job completion latency, ensuring better utilization of network resources, and improving the overall system throughput. To facilitate these improvements, my research proposes novel optimized routing approaches aimed at reducing network contention and job latency, which is critical for today's latency-sensitive scientific applications. Additionally, my research introduces enhanced scheduling and resource utilization methods to ensure better allocation of resources, such as processing cores, memory, and bandwidth. This is particularly important for bandwidth-sensitive applications and for maximizing the utilization of expensive resources to reduce operational costs.

My scholarly contributions toward the performance optimization of distributed memory workloads for HPC and data center networks are as follows:

- I proposed a novel adaptive routing algorithm for the Dragonfly+ or Megafly topology, which is currently used in many supercomputers on the Top500 list, including Italy's fastest supercomputer, Leonardo, and Canada's fastest supercomputer, Niagara. My proposed approach can reduce communication latency by more than 43% compared to other state-of-the-art routing methods.

- I proposed a novel multipath selection algorithm for multipath routing in the Jellyfish topology. The Jellyfish topology has significant potential for adoption in future data center networks due to its lower cost and higher flexibility compared to other interconnect alternatives. My proposed path selection approach, combined with multipath routing, can reduce latency by 42% compared to other state-of-the-art path selection methods.

- I evaluated the efficiency of my approaches using state-of-the-art simulation software for performance assessment in ideal network environments. I employed various well-known application benchmarks alongside real and synthetic applications for a comprehensive evaluation of my proposed methods across diverse

application behaviors and network configurations, facilitating their adoption in next-generation data center and HPC systems.

- I performed a comparative performance study of various structured and unstructured network topologies for applications with regular or structured communication behaviors, as well as for applications with unstructured communication behaviors, using diverse routing approaches. This study will help researchers and industry professionals adapt specific topologies with tailored routing approaches optimized for particular application types.

- I proposed a new process scheduling and resource allocation approach for HPC and data center systems to ensure better utilization of network and hardware resources. This study is beneficial for optimizing costly resources, reducing network contention, and increasing overall throughput.

The rest of the dissertation is organized as follows: Chapter 2 discusses the background and related work on performance optimization for distributed memory workloads in HPC and data center networks. Chapters 3 to 6 describe my four prominent contributions toward the performance optimization of distributed memory workloads. Finally, Chapter 7 concludes the dissertation.

CHAPTER TWO

BACKGROUND

This chapter begins by describing high-performance computing (HPC) and data center systems. It then explores the various network topologies used to interconnect these systems, followed by a discussion of routing methods and scheduling techniques. Finally, it addresses the various performance evaluation metrics used in this research.

2.1 High Performance Computing (HPC) Systems

High Performance Computing (HPC) systems are specialized computing environments distinguished by their high-speed processing capabilities, advanced networking technologies, and substantial memory capacity, which collectively enable the execution of extensive parallel processing tasks. These computing platforms are designed to handle complex calculations and large-scale data processing at significantly accelerated speeds. Examples include the Frontier and Summit supercomputers at Oak Ridge National Laboratory, the Aurora supercomputer at Argonne National Laboratory, and the Perlmutter supercomputer at Lawrence Berkeley National Laboratory. In such systems, thousands of computing nodes are interconnected via high-speed networks, with each node housing CPUs with multiple processing cores, from gigabytes to terabytes of memory, and often featuring specialized hardware accelerators like GPUs and TPUs. For instance, the Frontier system [4] has $9,472$ AMD CPUs, each with 64 processing cores running at 2 GHz, totaling $606,208$ CPU cores. Additionally, it includes $37,888$ MI250x GPUs, providing a total of $8,335,360$ GPU cores interconnected by a high-speed network. Each CPU has 512 GB of memory, while each GPU has 128 GB, bringing the total system memory to a staggering 9.2 petabytes. This massive HPC system is typically used for

large-scale scientific research and modeling, such as climate modeling, genomics research, fluid dynamics, and large-scale training of machine learning and deep learning models.

2.2 Large Scale Data Center System

Large-scale data center systems are specialized facilities designed to accommodate vast arrays of servers, storage, and networking equipment, optimized for efficiently managing, processing, and storing immense volumes of data. These data centers utilize advanced technologies to ensure high availability, scalability, and robust security for various applications, including cloud computing, big data analytics, and enterprise workloads. The fundamental architecture of both HPC and data center systems shares similarities, focusing on efficient resource management and performance optimization. Fig. 2.1 shows the frontier supercomputers at Oak ridge National laboratory.

2.3 Interconnection Networks

In HPC and data center systems, interconnection networks are used to facilitate communication between thousands of computing nodes or servers. These networks are designed to ensure low latency and high bandwidth, allowing for efficient data transfer during parallel processing tasks. The topology or architecture of such interconnection networks refers to the specific arrangement and organization of the connections between nodes in a network, which determines how data is transmitted across the network. In High Performance Computing (HPC) and data centers, the choice of topology significantly impacts performance, scalability, and fault tolerance. Common topologies used in HPC and data center interconnects include fat-tree, dragonfly, dragonfly+, torus, hypercube, mesh and clos networks each offering unique advantages in terms of scalability and performance.

Figure 2.1: Frontier Supercomputer at Oak Ridge National Laboratory (Source: ornl.gov) [2]

2.3.1 Fat Tree Topology

Fat trees are a standard hierarchical network used in data centers. One commonly used Fat tree model is the k-ary fat-tree architecture proposed by Al-Fares et al.[5]. A k-ary Fat tree with a switch radix of k will have k pods with k switches per pod. The k switches of a pod are equally distributed between level-0 leaf switches and level-1 aggregation switches. Half of the k ports of a level-0 switch are connected to compute nodes, and the remaining ports connect to level-1 aggregation switches. For level-1 aggregation switches, half of the ports are connected to level-0 leaf switches, and the remaining connect to level-2 core switches. Pruned Fat tree topologies where some pods are omitted from the standard k-ary Fat tree are generally used in production systems. For example, Summit [6] supercomputer at Oak Ridge National Laboratory uses a pruned

3-level Fat tree. We can represent a pruned k-ary Fat tree with three parameters: the number of levels l, switch radix k, and pods in pruned Fat tree p. The rest of the dissertation will represent any Fat tree topology as $FT(l,k,p)$. Figure 2.2 illustrates a 3 level fat tree with 4 leaf switches and 4 aggregate switches in each sub-tree or pod and there are two such sub-trees or pods $FT(3,8,2)$.

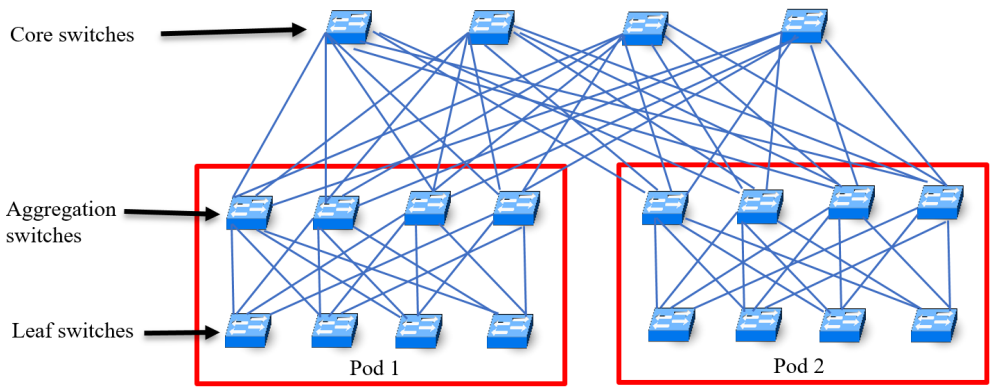


Figure 2.2: 3-level fat tree topology with 2 sub-trees and each sub-tree has 4 leaf and 4 aggregate switches

2.3.2 Dragonfly Topology

The Dragonfly network, proposed by Kim et al. [7], uses high radix routers to construct a low-diameter interconnect network. Dragonfly-based interconnect such as Slingshot [8] can be found in many of the current supercomputers listed in the Top500 [9]. Dragonfly is a hierarchical network comprised of group and system-level connections. A group in a Dragonfly is formed by connecting a network switches using an intra-group interconnection network. The system-level network is built by connecting switches of

multiple groups using an inter-group interconnection network. We can represent a Dragonfly topology using four parameters: the number of switches in a group a, the number of groups in a system g, the number of computing nodes in a switch p, and the number of global ports in each switch to connect switches to other groups h. The rest of the dissertation will represent a Dragonfly topology as $dfly(a,g,p,h)$. I follow the authors' suggestion to use $a \geq 2h$ and $2p \geq 2h$ [7] for better load balancing in my experiments. Figure 2.3 illustrates a dragonfly topology with 5 switches in each group and total 3 such dragonfly groups and 2 global ports in each switch to connect to switches of another group.

2.3.3 Dragonfly+ Topology

Dragonfly+ [10] or Megafly [11] is a variation of Dragonfly topology. Dragonfly+ based topology is currently being used in many top supercomputers in Top500 [9] lists such as Italy's supercomputer Leonardo [12] and Canada's supercomputer Niagara [13]. In Dragonfly+ topology the intra-group topology is a folded clos network. There are two types of switches in a Dragonfly+ group. One is spine switches that are connected to global links to connect switches of other groups and another is leaf switches that are connected to compute nodes. Within a group each leaf switch has a direct connection to all of its spine switches and each spine switch has direct connection to all of its leaf switches. However their is no direct connection between any two leaf switches and any two spine switches. We can represent a Dragonfly+ topology using five parameters : number of leaf switches l in a group, number of spine switches s in a group, number of groups in a system g, number of computing nodes in a switch p and number of global ports in each spine switch to connect switches of other groups h. However in a balanced Dragonfly+ topology we should keep $l = s = p = h$. In this study I used balanced Dragonfly+ topology which we can represent as $dfly+(l,g)$ or $mfly(l,g)$. Figure 2.4

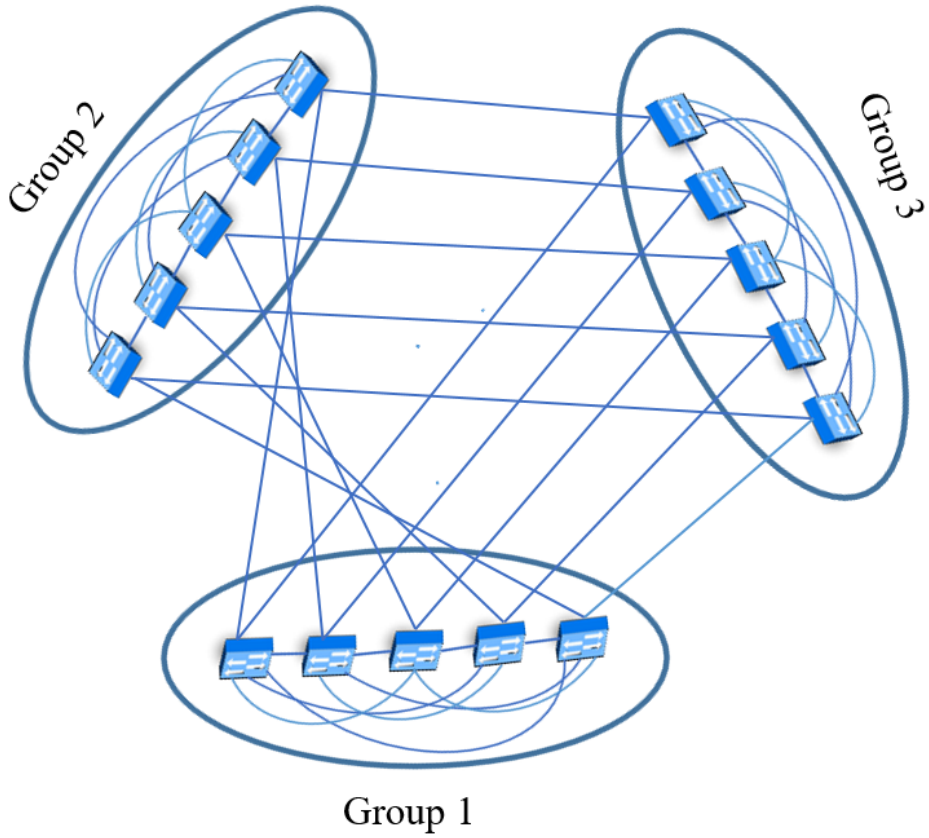


Figure 2.3: Dragonfly topology with 3 groups and 5 switches in each group

illustrates a dragonfly+ topology with 5 dragonfly+ groups and each group has 2 leaf and 2 spine level switches $mfly(2,5)$.

2.3.4 Jellyfish/Random Regular Graph(RRG) Topology

Jellyfish, also known as the Random Regular Graph (RRG) topology, is a random network based on a flexible interconnect topology proposed by Singla et al. for scalable data center designs [14]. In the Jellyfish topology, all the network switches have an equal radix, and the endpoint switches of each link are randomly chosen from a uniform

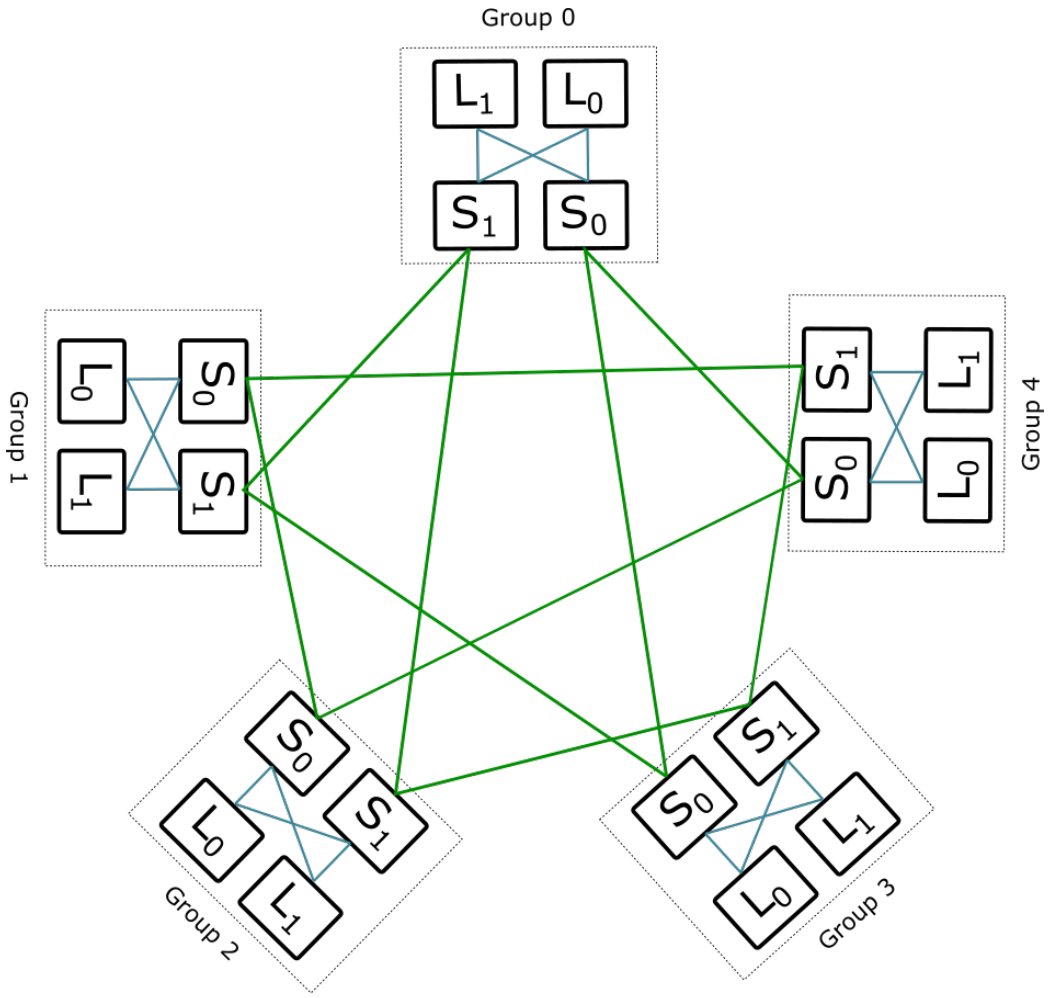


Figure 2.4: Dragonfly+ topology with 5 groups and 2 leaf and 2 spine switches in each group mfly(2,5)

distribution, offering high design flexibility at scale. Its construction eases incremental network expansion and contraction while minimally affecting its topological properties. Relative to other topologies of equal capacity, the Jellyfish/RRG topology provides a stable diameter and bisection bandwidth [15][16]. We can represent a Jellyfish or RRG topology using two parameters: the number of network switches N and the available ports r per switch used for switch-to-switch connections. Throughout the whole dissertation, I

will denote a Jellyfish topology as $RRG(N,r)$. Figure 2.5 illustrates a jellyfish topology with 16 switches and degree of each switch is 3 $RRG(16,3)$.

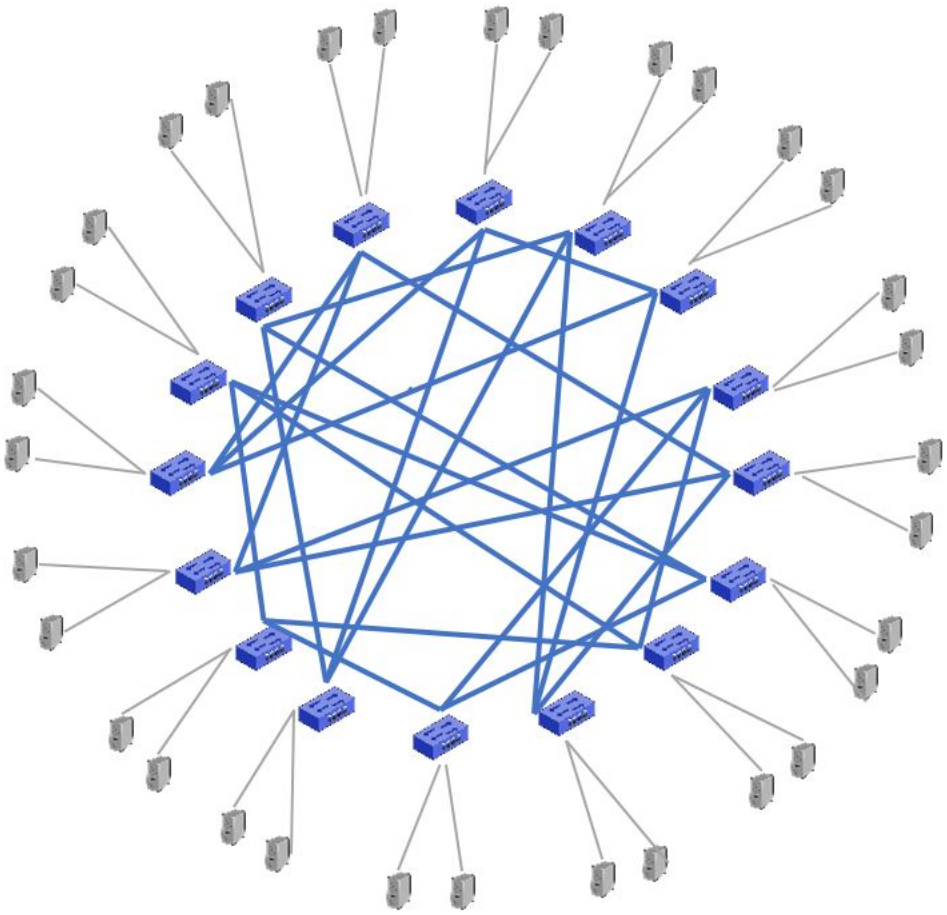


Figure 2.5: A Random Regular Graph consisting of 16 switches and 32 server endpoints. Each switch uses 3 ports connecting to other switches. We refer to such topology as $RRG(16,3)$, regardless of the number of endpoints.

2.4 Routing Methods in HPC and Data center Networks

To ensure efficient communication between computing nodes of HPC and data center system various routing methods are used. Given a particular network topology selecting an appropriate routing strategy can significantly influence performance, scalability, and fault tolerance. This section discusses various routing methods commonly employed in HPC and data center environments.

- **Static Routing** Static routing always uses predetermined paths to send and receive data packets within the network. It is very straightforward and incurs minimal overhead. One of the major disadvantages of static routing is that it lacks the flexibility to handle network congestion and node failures. This type of routing is suitable for uniform and predictable network traffic patterns.

- **Dynamic Routing** Dynamic routing adjusts routes based on real-time network conditions. It continuously monitors various network parameters and performance metrics, and based on that information, it dynamically adapts its routes to avoid network congestion and failed nodes. One of the disadvantages of dynamic routing is the overhead associated with managing complex routing tables.

- **Adaptive Routing** Adaptive routing is the most commonly used dynamic routing approach, where the algorithm modifies routing decisions based on real-time information such as buffer queue lengths and latency data. By dynamically changing routes, this type of routing provides lower congestion and better load balancing which improves the system throughput and reduce the latency. However, adaptive routing can also lead to increased computational overhead and complexity.

- **Multipath Routing:** Multipath routing employs multiple paths to send and receive data between each source and destination node or server. This method enhances

fault tolerance and load balancing by distributing traffic across several routes, which can prevent bottlenecks and improve overall network reliability. However, selecting the best multipath routes is an optimization problem that requires complex computation and routing table management.

2.4.1 Fat Tree Routing

Fat tree networks commonly use static routing schemes unaware of network communication traffic. For Fat trees, an oblivious routing algorithm chooses the same Nearest Common Ancestor(NCA) between a source-destination pair irrespective of network congestion. Source-mod-k (S-mod-k) [17][18] and Destination-mod-k (D-mod-k) [19] are the most commonly used oblivious routing schemes. For my experiments, I used oblivious schemes where the same Nearest Common Ancestor (NCA) between a given source-destination pair is chosen in a round-robin manner. Adaptive routing has been proposed for the fat tree topology [20], but, in this dissertation study I only consider the routing schemes provided in the state-of-the-art *SST/Macro* framework, which is the simulation platform that I use in our evaluations.

2.4.2 Dragonfly and Dragonfly+ Routing

I considered four of the most commonly used routing techniques for the Dragonfly topology. The following are the routing techniques used in this study.

- *Minimal Routing (MIN) :* This routing algorithm forwards traffic using the available minimal paths between source and destination pairs. This routing technique is good for uniform traffic.

- *Valiant Routing (VAL):* In the Valiant routing scheme[21, 22], traffic is first sent non-minimally towards an intermediate switch (except the source and destination switch). Next, the traffic is forwarded minimally to the destination.

- *Universal Globally Adaptive Load-balanced(UGAL) Routing:* This adaptive routing algorithm decides between MIN and VAL routing schemes using the estimated queue length (i.e., congestion) and hop count of the path to calculate an estimated delay. Traffic is routed to the path with the smallest delay. We denote the queue length and hop count of the minimal path Q_{min} and H_{min}, and the queue length and hop count of the valiant path as Q_{val} and H_{val}. Using this notation, UGAL will route a packet minimally if the following equation is satisfied where T is a constant routing threshold:

$$Q_{min} \times H_{min} \leq Q_{val} \times H_{val} + T \tag{2.1}$$

- *Progressive Adaptive Routing (PAR):* First proposed by Jiang et al. [23], PAR routes packets according to congestion similarly to UGAL except for when traffic is routed non-minimally. In VAL, the decision to route non-minimally is determined for each packet with paths chosen at random from each of the two sets of paths (i.e., minimal and non-minimal paths). PAR differs in that any non-minimal route choice becomes permanent.

- *Piggyback Routing (PR):* Piggyback routing(PR) [24] is another packet-adaptive routing that was originally proposed for the Dragonfly topology, but yet to be evaluated for the megafly design. Contrary to PAR and more similar to UGAL, PR makes the per-packet routing decision at the source/leaf router level. However, the formulation used by PR in decision making is different from UGAL. Instead of comparing local output buffer occupancy of minimal and non-minimal candidate paths, the source router of PR relies on receiving global congestion information from the globally connected routers. To use PR over megafly, each spine router would need to calculate a single bit global congestion state for each global outgoing link (GL) using the following formula given by [24]:

$$S_{GL} = Q_{GL} > 2 * \bar{Q} + T_{PR} \tag{2.2}$$

Where Q_{GL} represents the queue occupancy of the output buffer associated with the link GL, $\bar{Q}$ represents the average buffer occupancy over all global outgoing ports and T_{PR} is another offset value for calibration purposes. The original Dragonfly version of PR requires each router to broadcast its global information to all of its adjacent routers (in the same group). Clearly, this can result in flooding of congestion information packets, which is not plausible especially when the size of a group gets large. However in Megafly, only the spine routers need to disseminate and instead of broadcasting to an entire group, each spine router only needs to multicast the global congestion information to the leaf routers. As such, the total number of packets that need to propagate congestion information in a megafly group decreases to almost one-fourth of the same needed in Dragonfly.

2.4.3 Jellyfish/RRG Routing

A Jellyfish network is considered unstructured due to the randomness of its connections. Each source and destination switch may have a limited number of *shortest* paths; however, it has many *short* paths. Jellyfish topologies typically employ multi-path routing leveraging this path diversity. Singla et al. [14] showed that standard Equal Cost Multi-Path Routing (ECMP) does not provide enough path diversity in Jellyfish, and cannot utilize the entire capacity of the network. The authors suggest using K-shortest paths [25] routing where traffic is forwarded to a fixed k shortest paths sampled from all available paths between a source and destination pair. In this study, I considered three types of routing in Jellyfish.

- *Uniform Multi-path Routing (UM):* This is the most commonly used routing in Jellyfish networks [14]. Packets are routed along k shortest paths sampled from all paths between a source and destination pair.

- *Adaptive Multi-path Routing (AM):* This routing algorithm is an adaption of Universal Globally Adaptive Load-balanced(UGAL) [26] for Jellyfish. In this algorithm, the k shortest paths are partitioned into two categories based on hop lengths. Path lengths less than or equal to a particular hop count are considered minimal, and other paths are considered non-minimal. Then two paths are randomly chosen from each set and compared by estimating end-to-end congestion. Traffic is forwarded using the least congested path.

- *Minimal Routing (MIN):* In this routing we selected all the available shortest paths among each source and destination pair of switches and forwarded the traffic only on those selected shortest paths.

2.5 Scheduling Methods in HPC and Data center Networks

In HPC and data center networks, scheduling parallel tasks plays a significant role in overall resource utilization, system performance, and execution efficiency. In an HPC system, a task is partitioned into multiple subtasks, and each of these subtasks is scheduled to be performed on different computing nodes or servers interconnected by the network topology. Optimizing scheduling algorithms can enhance resource utilization, lower latency by reducing network communication bottlenecks, and improve overall system throughput. Various task scheduling algorithms are employed in HPC, including consecutive placement of tasks, random placement of tasks, and traffic-aware task placement. Each of these algorithms has its own advantages and disadvantages. For example, consecutive task placement is easier to implement, but it may result in

suboptimal resource utilization. On the other hand, random placement can create an uneven distribution of tasks due to its lack of a strategic approach.

2.6 Performance Evaluation Metrics

Following defines the metrics used in my evaluation.

- *MPI Communication Time:* MPI communication time refers to the estimated message transmission and synchronization time by each MPI rank, and reported in seconds. The reported MPI time is cumulative across all ranks.

- *Estimated Total Runtime:* Refers to the simulated time of the parallel application in seconds.

- *Hops:* Hops refers to the total number of network links (i.e., connecting the switches) traversed by each packet in an application.

- *Queue length:* Queue length corresponds to the number of outstanding packets waiting on a port when a new packet arrives. In $SST/Macro$, packet queues are maintained on a per-port basis, and ports support several virtual channels. Queue length of a port is thus calculated by accumulating queues across the virtual channels.

- *Stalls:* Stalls indicate congestion faced by each packet in its route from an arbitrary source to destination. From the instance that a packet arrives at a particular switch port, it is queued until forwarded, and we calculate the stalls in proportion to the queue length of the waiting packets on the port. The total amount of stalls that a packet faces is the sum of all the queue length values it encounters between the source and destination. The distribution of stalls across the path of packet traversals

(i.e., the critical path) is referred to as *stall queue length* or just *stall queue*; it is an essential metric to compare performances.

CHAPTER THREE

ROUTING OPTIMIZATION FOR DRAGONFLY+/MEGAFLY TOPOLOGY

Interconnection network is a key major component of High Performance Computing (HPC) clusters and supercomputers. To construct HPC systems of exascale computing capacity and beyond, thousands of computing nodes need to be interconnected with a network topology that can deliver high throughput and low latency at a large scale. To design such scalable high performance interconnect in a cost-effective way, many new network topologies have been proposed in recent years as alternatives to the prevalent yet expensive fat-tree based interconnect designs. Among those, the Dragonfly topology, first introduced by Kim et al. [27], is particularly well-known for its scalability and low diameter. Dragonfly-based interconnects have been used to construct several high-performance clusters in the current Top500 [9] list of the world's fastest supercomputers such as Piz Daint [28] and Trinity [29].

More recently, a new variation of Dragonfly named Megafly, and also known as Dragonfly+, has been introduced and successfully deployed to construct Niagara [30], the fastest supercomputer accessible for academic use in Canada. megafly offers an even higher scalability and better path diversity than the canonical Dragonfly design and thus, is a prominent candidate for use in future supercomputing interconnects. Similar to Dragonfly, megafly topology heavily relies on the load balancing features of adaptive routing among groups of routers to maximize its throughput performance.

However, the practical adaptive routing algorithms of present day, namely Universal globally Adaptive Load-Balance (UGAL) Routing [26] and Progressive Adaptive Routing (PAR) [24], do not leverage important properties specific to the megafly topology, such as path distribution and minimal path diversity.

As a result, these algorithms often fall short of a robust performance on megafly interconnects. For example, The UGAL algorithm on megafly suffers from limited congestion detection capability that leads to inaccurate and thus performance-limiting adaptive routing decisions. The PAR algorithm, on the other hand, may offer limited load balancing capability despite its accuracy to detect congestion on megafly networks.

In this work I presented a detailed analysis to demonstrate that the throughput performance of PAR - the currently best known routing scheme for megafly - is not consistent across all variations of scale and wiring schemes. More specifically, I show that the performance of PAR significantly degrades with the increase in the number of groups in the megafly network when each *spine* router of a megafly group can no longer be directly connected to all other router groups. This considerably limits the scale at which megafly can be built and run efficiently with PAR. Other competing adaptive routing schemes, namely UGAL and Piggyback Routing [24], are also shown to suffer from sub-optimal performances due to their limited compatibility with megafly designs. To overcome such limitations, I propose a new routing algorithm: K-path UGAL with Global Congestion Notification (KU-GCN), specifically for Megafly. My new routing scheme achieves load balance across a much wider group of megafly networks in compare to PAR, UGAL and Piggyback Routing. I validate my claim by presenting results obtained from trace-driven simulation of several megafly interconnects with both synthetic and application benchmark workloads. I show that my proposed modifications to megafly routing yield better, and equally importantly, more stable performances across multiple topological variations when compared with other megafly routing schemes.

3.1 Optimized Routing For Dragonfly+/Megafly Topology

As discussed earlier about the potential pitfalls of existing adaptive routing for megafly topology I postulate the following requirements to develop new adaptive routing

scheme specifically for the megafly topology that can perform efficiently across various designs and scales:

- **Requirement 1:** The adaptive scheme needs to increase the number of candidate paths considered for routing each packet as the path diversity in the topology increases.
- **Requirement 2:** To spread the non-minimal routes uniformly across the interconnect at a large scale, such paths need to be chosen at the source/leaf router level.
- **Requirement 3:** Since output buffer occupancy at the source router is not a strong indicator of inter-group congestion due to the presence of multiple links between groups, the adaptive routing scheme for megafly must have an improved mechanism to detect inter-group congestion.

3.1.1 K-path-adaptive Piggyback Routing (KPR)

My first and preliminary adaptive routing solution for megafly is a straightforward extension of Piggyback Routing with minimal modifications to satisfy all required properties of megafly routing. In KPR on $mfly(s,g)$, each leaf-level router maintains a *congestion vector* of s^2 bit length. Each bit of the congestion vector represents the congestion status of an inter-group global link. The congestion information may be piggybacked with any inbound traffic, or may be sent as independent congestion notification messages from the spine-level routers. For each packet arriving at the source router, a spread of K minimal paths is considered and the packet is forwarded along the path whose global link congestion bit is false (indicating uncongested state) and whose local outgoing port has the smallest buffer occupancy. If no such path exists, then the packet is first routed minimally towards a random intermediate leaf router that is neither in

source nor in destination group, and again minimally from the intermediate router to the destination router. I denote KPR(K) as the KPR routing scheme with K minimal path spread parameter.

Similar to PR in [24], KPR solely relies on the abstraction of minimal path quality to choose a path for routing and lacks any mechanism to evaluate non-minimal paths at the source router. Under steady traffic conditions, KPR performs at least as good as Piggyback Routing, with added advantage of being able to utilize the path diversity of Megafly.

3.1.2 K-path UGAL with Global Congestion Notification (KU-GCN)

My main solution for load-balanced megafly routing combines the key performance elements of UGAL and Piggyback Routing. In KU-GCN routing, once again each source router gathers congestion information from spine routers and maintains a local congestion vector. For each incoming packet, the source router samples K minimal candidate paths. If there exists no globally uncongested route among the K paths, the packet is forwarded non-minimally. Otherwise, a UGAL comparison is made between the globally uncongested minimal path with lowest local output buffer occupancy with a globally uncongested minimal path of lowest buffer occupancy towards a randomly selected destination obtained from another K paths sampling. I denote the scheme as KU-GCN(K) as it considers at least K minimal paths and up to another K non-minimal paths to make the routing decision for each packet.

3.2 Performance Evaluation

For evaluating the performance of my proposed optimized routing approaches I used SST-Macro [31], an open-source coarse-grain parallel discrete event simulator as my simulation platform due to its popularity and suitability to simulate large scale distributed architectures. SST-macro supports offline mode replay of MPI communication traces on a

variety of built-in interconnects including Megafly(referred as Dragonfly Plus in SST-Macro). I extended the existing megafly implementation by adding support for new wiring and routing schemes. I used SST-Macro to replay a variety of bench-marking application traces from NAS parallel benchmarks [32] representing common communication-intensive HPC workloads: Block Tri-diagonal solver (BT), Conjugate Gradient (CG), Fourier Transform (FT), Integer Sort (IS) and Multi-Grid Communication (MG). Both the synthetic and the benchmark applications are divided into 1024 MPI ranks. In my experiments, a lower cumulative MPI communication time metric corresponds to better routing performance, and vice versa.

3.2.1 Topology Configurations

The following megafly topologies are used in my evaluation:

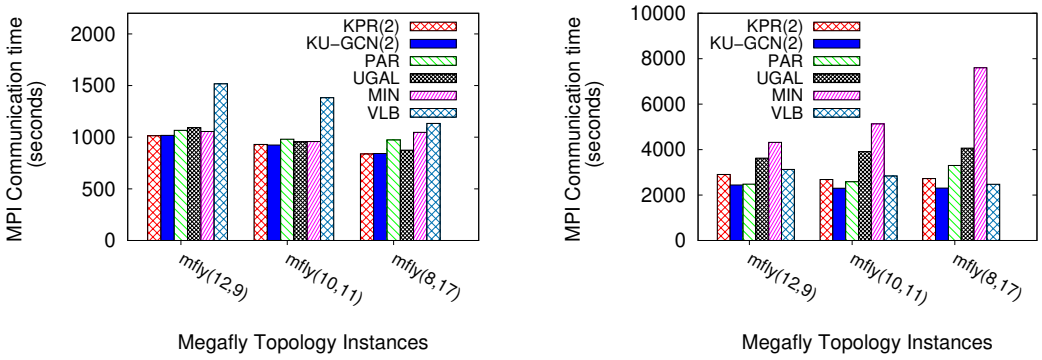
- $mfly(12,9)$: A high connectivity topology made of 9 groups of radix-24 routers and 1,296 nodes. Each group has 24 routers(12 leaf and 12 spine) and 144 global links outgoing to the remaining 8 groups. The minimal path diversity is 18.
- $mfly(10,11)$: A medium connectivity topology made of 11 groups of radix-20 routers and 1,100 nodes. Each group has 10 leaf routers, 10 spine routers and 100 global links outgoing to the remaining 10 groups. The minimal path diversity is 10.
- $mfly(8,17)$: A low connectivity topology made of 17 groups of radix-16 routers and 1,088 nodes. Each group has 16 routers and 64 global links connected with the remaining 16 groups. The minimal path diversity is 4.

Despite having similar network sizes, the three megafly topologies I consider have very different network degrees and global connectivity levels, characterizing different regions of the megafly design spectrum. I further note that any Megafly topology instance that requires scaling beyond the limits of a 3-level Clos network of equal router radix, will

belong to the "low connectivity" category. The performance results of the $mfly(8,17)$ topology is therefore, much more revealing of Megafly's performance at scale compared to $mfly(10,11)$ and $mfly(12,9)$.

Among the three topologies, only the groups of the $mfly(10,11)$ with medium connectivity can be fully wired using AlltoAll wiring scheme. The $mfly(12,9)$ would have 48 unused ports after completing AlltoAll wiring, the groups of $mfly(8,17)$ simply cannot be connected with AlltoAll wiring. To ensure the maximum utilization of global bandwidth capacity, all three topologies used in my evaluation are constructed using random wiring scheme.

3.2.2 Synthetic Traces Results



(a) Random Permutation Traffic

(b) Adversarial Shift traffic

Figure 3.1: Communication Performance of 1024-rank synthetic workloads on megafly topologies

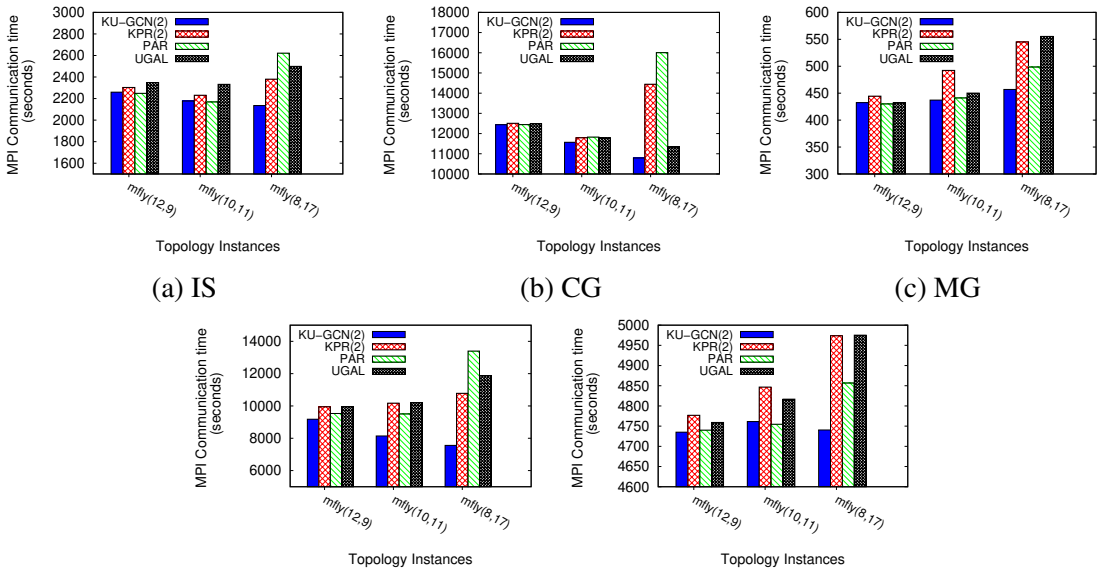
I useed synthetic traces to characterize the performance of the megafly routing schemes under steady traffic conditions. In addition to measuring the performances of

KU-GCN, KPR(K-path-adaptive Piggyback Routing), PAR and UGAL adaptive routing schemes, we report the performances of deterministic minimal routing(MIN) and deterministic non-minimal Valiant's Load-Balanced(VLB) routing as baselines. Figure 3.1*a* shows the communication time of these six routing schemes on benign random permutation traffic. As expected, the performance of all four adaptive schemes are similar to that of minimal routing on $mfly(12,9)$ and $mfly(10,11)$ topology instances, with KU-GCN and KPR slightly outperforming MIN by at least 2.9% and UGAL and PAR under-performing than MIN by 0-4%. However in $mfly(8,17)$ where the number of groups increases and the inter-group connectivity decreases, PAR's performance starts to degrade and takes at least 10% more communication time compared to the other adaptive schemes while KU-GCN, UGAL and KPR perform comparably to eath other.

Figure 3.1*b* shows the routing performances on adversarial shift traffic pattern workload where the majority of data packets need to be routed non-minimally. On Shift traffic, KU-GCN performs strictly better than all other routing schemes. Interestingly, both UGAL and KPR perform significantly worse than KU-GCN in all topologies by at least 16% and 48% . This reveals that both UGAL and the Piggyback routing struggle to detect adversarial traffic conditions, regardless of the topology or wiring scheme. Once again, PAR maintains on-par performance with KU-GCN on high-connectivity topology instances, but takes 43% more communication time than KU-GCN on the $mfly(8,17)$ topology.

3.2.3 Benchmarks/Applications Results

Figure 3.2 shows the average communication time of five application benchmarks using four adaptive routing schemes over three topology instances. A clear trend is noticeable from the Figure. In all five applications, the performances of all four adaptive routing are similar on the $mfly(12,9)$ topology with high inter-group bandwidth.



(a) IS (b) CG (c) MG

(d) FT (e) BT

Figure 3.2: Communication Performance of NAS benchmark applications on megafly networks

However, as we employ more groups and the inter-group bandwidth is reduced, the performance of PAR becomes increasingly inefficient. Once again, KU-GCN routing performs at least on par with PAR on $mfly(12,9)$ and $mfly(10,11)$ and it shows significant performance gains on $mfly(8,17)$ by margins of 18.5% (IS), 32.5% (CG), 8.3% (MG), 43.5% (FT) and 2.3% (BT). KPR and UGAL routing schemes perform worse than KU-GCN across all topologies and application benchmarks. The performance of KPR, on average, is 3.4% slower than PAR on $mfly(12,9)$ and $mfly(10,11)$ and 5.2% faster than PAR on $mfly(8,17)$. The performance of UGAL, on average, is 2.8% slower than PAR on $mfly(12,9)$ and $mfly(10,11)$ and 6.2% faster than PAR on $mfly(8,17)$. The performances of KPR and UGAL on $mfly(8,17)$ shows high variability across the benchmarks.

3.2.4 Discussion

In this research I introduced a new adaptive routing scheme that can improve the communication performances of any megafly based interconnects. The new routing scheme performs significantly better than the currently widely adopted progressive adaptive routing scheme in large scale and/or connectivity constrained megafly designs. The megafly topology is primarily proposed to build systems at a very large scale. Therefore, maximizing performance at scale is highly important to the prospects of its adoption in future exascale systems. I consider my work to be a critical first step towards designing robust megafly based interconnects.

CHAPTER FOUR

OPTIMAL PATH SELECTION FOR MULTIPATH ROUTING IN RANDOM TOPOLOGY

The network topologies used in the present generation of HPC and data center systems such as torus, fat-tree and dragonfly - have rigorous structures and require strict network sizes and wiring schemes to maintain their corresponding topological properties. However, to meet future scalability demands, a variety of new network topologies have been proposed in the last decade that use either semi-structured, or even fully randomized interconnection of switches. Due to their excellent expanding properties, random topologies can support a higher network size than the regular-structured graphs on a given diameter limit [33] and therefore, they are likely candidates for adoption in future scalable interconnect deployments.

A major roadblock to implement large scale randomized interconnects is the high routing complexity due to the requirement of large and fast memory at the switch level to store routing tables. Each switch of a randomized interconnect needs to separately store a full routing table since the forwarding entries do not have any pattern to aggregate upon due to the randomness of connections. Furthermore, it is well-observed that randomized networks perform poorly with single-path routing and thus, needs to balance traffic loads over multiple short paths to optimize its end-to-end communication performance [34]. This requirement of multi-path routing puts further strain to the memory capacities of switches. For practical deployment of randomized interconnects, a combination of routing and path selection schemes is, therefore, much needed which can maximize the performance of each communication while using a minimal number of paths.

In this work, I presented a new heuristic driven algorithm to maximize the throughput performance of interconnect networks by selecting and using k best paths to

forward traffic between any of its switch pairs for a given value of k. My proposed algorithm is especially suitable for randomized interconnection networks with high short path diversity where all short paths can not be utilized for routing due to routing table limits. My algorithm is designed to be topology-aware but oblivious to traffic patterns, making it simple for use and implementation. I further devise a novel variation of the UGAL adaptive routing scheme that further exploits my path selection scheme. The combination of my path selection scheme and adaptive routing scheme is shown to perform significantly better than the current best known combination of k-shortest paths with UGAL routing schemes while requiring only a fraction of routing table size. I back up my claim with an extensive set of experimental data obtained by using both synthetic and application workloads.

4.1 Problem Motivation

Generally, k-shortest path routing (KSP) is effective at improving the throughput performance of topologies that have a high diversity of short paths. Since random regular graphs with sufficiently large degree are known to have high path diversity at a large scale, it is intuitive to consider KSP as the routing scheme of choice for RRGs. However, it is non-trivial to ensure efficient routing of traffic flows on Random Regular Graphs due to their variable quality and quantity of paths available between their endpoint node pairs.

k-shortest path routing, by design, tends to focus solely on the quantity of available short paths that can be used to forward traffic and achieve load balance. Due to considering hop length as the gold standard measure of path quality, the current routing schemes lack the flexibility to distinguish and avoid short paths of poor quality that use the bottleneck links.

As a motivating use case, let us consider any path of an RRG that contains a bottleneck link. Let us further consider that the path is the n-th shortest path between its

source and destination node. Now, if we employ uniform k-shortest path routing and $n \leq k$ holds true, then the routing scheme will forward $\frac{1}{k}$ amount of traffic through the bottleneck link. To achieve load balance, we would need to minimize the fraction of traffic being forwarded to the bottleneck link by increasing the value of k. As mentioned in the previous section, using large value of k is prohibitive in KSP routing due to the high routing table size overheads caused by it. Arbitrarily lowering the value of k is also not feasible because n can be any value between 1 and k and any attempt to avoid the bottleneck link would also likely cause exclusion of numerous short paths from consideration for routing. Our take-away from this analysis is that finding the performance-ideal value of "k" in KSP is not trivial and leveraging "k" alone cannot effectively minimize bottleneck traffic without incurring new performance overheads. The same performance issues persist on adaptive multi-path routing as well, although to a lesser extent due to the per-packet re-routing capability assigned to routers. To minimize the severity of bottleneck congestion, the path selection we requires needs to be more flexible and informed than k-shortest path selection.

4.2 Proposed Optimized Solution

I begin my approach to solve the routing optimization problem for RRG by emphasizing on the evaluation of quality of all short paths, and then use the evaluation information to make an informed k path selection for routing. Opting for simplicity, I choose the common metric of path load (PL) to quantify the quality of a routing paths. Calculating the maximum link load on a given topology requires an assumption of an existing path selection S and a traffic communication pattern P (set of source-destination node pairs). I assume multi-path routing with k routes considered for each communication. Let a link l with a capacity C be used X times to carry $\frac{1}{k}$ amount of end-to-end in each usage, assuming uniform-split routing of traffic. Then, link load of l

corresponding to path selection S and pattern P is defined as $load_l = \frac{X}{k \times C}$. The path load (PL) of a routing path p can be expressed as:

$$PL(p) = max_{l \in p} load_l \tag{4.1}$$

The maximum link load (MLL) of a topology corresponding to the path selection S, traffic pattern P and uniform-split multi-path routing is simply the maximum of all path loads. Let $paths(s,d,S)$ be a function that returns k paths between a given source node s and a destination node d from the path selection S. Then,

$$MLL(S,P) = max_{(s,d) \in P, p \in paths(s,d,S)}(PL(p)) \tag{4.2}$$

I will use the MLL metric for performance evaluation and thus, validation of my proposed path selection scheme.

To avoid the shortcomings of KSP, I introduce a simple, greedy path selection scheme that selects one path at a time in an iterative, round robin fashion over all source-destination pairs. Each path is selected from a candidate pool of all short paths between the corresponding source and destination. Candidate paths are evaluated based on their path load values over the MLL of already selected paths, and the path that has the least amount of path load based on the already selected paths, is selected. In the following, I will describe the parameter selection to our path selection problem, followed by the path selection algorithm.

4.2.1 Parameter Selection

Formally, the goal of my path selection scheme is to choose the best performing k number of paths from a set of available candidate paths between all possible source-destination pairs of a given topology. Therefore, my approach begins with the

enumeration of candidate paths. I find all short paths between each possible pair of source-destination nodes and use them as candidate paths for k-selection in the iterative part of my algorithm. In a topology with N switches and r switch-switch ports, I consider any loop-less path of hop length H or lower as a short path where H is the smallest integer value satisfying $1 + r + r(r-1) + ... + r(r-1)^{H-1} > N$. Furthermore, if all the paths originating from a single node are uniformly spread over all other nodes, the expected number of short paths between any node pairs can be shown to be $\frac{1+r+r(r-1)+...+r(r-1)^{H-1}}{N}$ by using the upper bound of a Moore graph of degree r and diameter H. To make sure that the k-shortest path routing makes sufficient use of short paths, I propose using k path selection algorithms only when k is less than the expected number of short paths. Also, I note that my suggested bound on the value of k does not guarantee that there will be more than k short paths available in between all source-destination pairs, especially in random graphs such as RRG. For those source-destination pairs with insufficient short path diversity, I simply select paths using KSP algorirthm. In other words, my path selection scheme is only relevant to those node pairs which have more short paths available than that can be practically utilized in k-multi-path routing.

4.2.2 Optimized path selection Algorithm

Algorithm 1 formally describes my optimized k path selection scheme. The algorithm computes k optimized paths between each pair of switches, and repeats for all possible switch pairs. I note from Line 3-4 of the figure that the iteration over number of paths is performed before the iteration over all switch pairs. This guarantees that all switch pairs will be allocated i optimized path each before any switch is allocated the $(i+1)$'th optimized path and thus, the network load values that I use to make path selection decisions, is uniformly contributed by all switch pairs. For a better fairness of

Algorithm 1 Optimized k-path Selection Algorithm

Input: Topology with N switches and L switch-switch links, set of link capacities C, path spread k

Output: Load optimized path selection S containing k paths for each switch pair

$S \leftarrow \emptyset$ $load_l \leftarrow 0$, $\forall l \in L$ **for** $i \leftarrow 1 \ldots k$ **do**
  for *each node pair* (s,d) *such that* $1 \leq s,d \leq N$ *and* $s \neq d$ **do**
    Let S_{sd} := All short paths between s and d **if** $S_{sd} - S = \emptyset$ **then**
      Use *KSP* algorithm to find the ith shortest path and denote it as p'
    end
    else
      For each unselected path p in S_{sd}, calculate $cost_p = (\text{Hop length}(p)) \times PL(p)$ Let p' be the path with minimum cost value
    end
    $S \leftarrow S \cup \{p'\}$ **for** *each link* $l \in p'$ **do**
      $load_l \leftarrow load_l + \frac{1}{k \times C_l}$
    end
  end
end

path allocation to switch pairs, I suggest that the source-destination switch pair selection in Line 2 could be done in randomized order. Line 3 calculates all short paths between a selected switch pair. This can also be pre-computed to minimize the time complexity of my algorithm. Lines 3-5 addresses the cases of switch pairs with limited path diversity where I do not need to optimize path selection. Lines 6-8 outlines my greedy heuristic of path selection based on the link loads caused by the existing paths in S. Lines 9-10 updates the load values of the links that falls onto the newly added path. The algorithm iterates exactly $k \times N \times (N-1)$ times and adds one path to S in each iteration; each time minimizing the path load and thus essentially, minimizing the maximum link load of the entire network topology.

4.3 Performance Evaluation

For evaluating performance of my proposed optimized path selection algorithm (OPT) I used SST-Macro [31], an open-source coarse-grain parallel discrete event simulator as my simulation platform. I used SST-Macro to replay a variety of bench-marking application traces from NAS parallel benchmarks [32] and three synthetic traffic patterns all to all traffic, random permutation traffic and adversarial shift traffic.

4.3.1 Experimental Results

In this section, I present experimental results that demonstrates the performance characteristics of my optimized k-path selection algorithm on randomized networks. All of my experiments are conducted on an RRG (96,13) topology instance consisting of a total of 1,248 switch-switch ports interconnected by 624 random interconnections. when constructed using 24-port switches, the RRG(96,13) topology is then used to interconnect 1,056 endpoint server nodes, approximating a medium scale of an HPC or data center topology. I assume that all links of the topology have unit/uniform capacity. Throughout

the remainder of this section, I will refer to my optimized path selection algorithm as *OPT* while presenting experimental data.

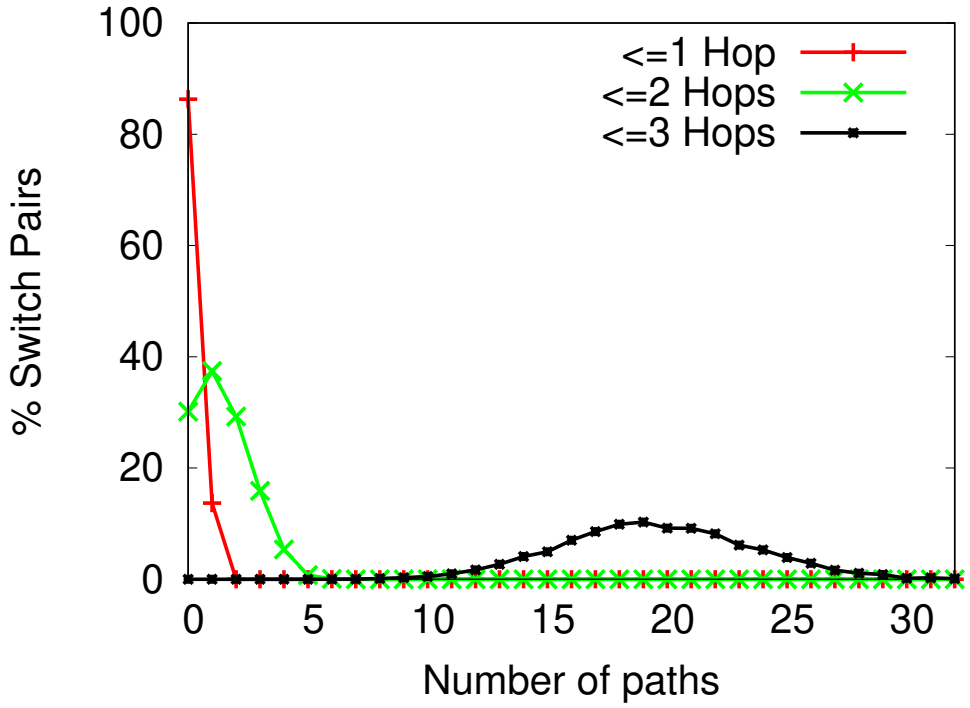
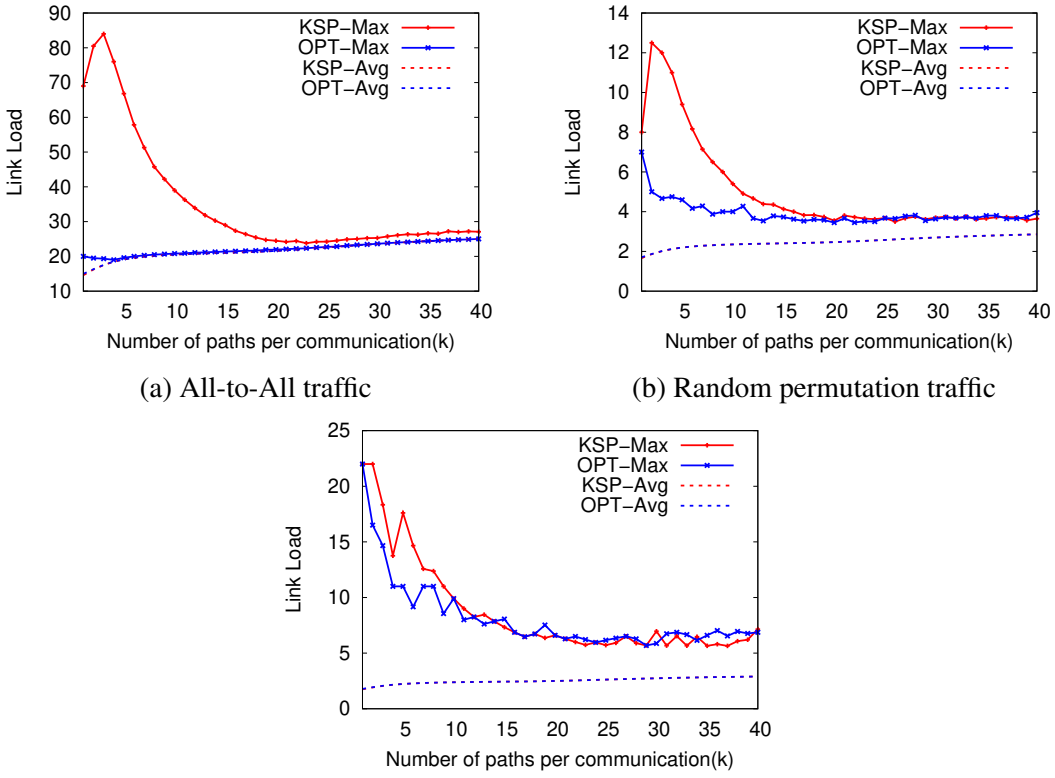


Figure 4.1: Distribution of path lengths of an *RRG* $(96, 13)$ topology

Figure 4.1 shows the distribution of short paths over all source-destination switch pairs of the RRG (96,13) topology I use. Clearly, the topology has a diameter of 3 since 30% of the switch pairs have no paths of 2 hop length or less and thus, have the shortest path length of 3. We can also observe that the topology has a large number of short paths available. 98 percent of all the switches have from 12 to 32 short (3 hops or less) paths among each other and the median number of short paths per switch pair is 20.

In a high path-diversity Random regular graph topology such as the one I consider, the switches have a high number of short paths of equal length. When the k-shortest path algorithm is applied to my topology, the k paths can be selected randomly at best, since the available paths are not distinguishable based on their lengths. To achieve load balance, KSP would need to use a k value that is high enough to consider all short paths for routing.



(a) All-to-All traffic

(b) Random permutation traffic

(c) Shift traffic

Figure 4.2: Maximum Link Loads of All-to-All traffic pattern over varied number of paths per communication in an $RRG(96, 13)$ topology

To demonstrate the impact of the choice of the parameter k in KSP and OPT, I evaluate the maximum and average link loads in the RRG(96,13) topology under three distinct traffic patterns:

- **All-to-all traffic:** Each switch communicates with all other switches and thus, results a dense traffic matrix.
- **Random permutation traffic:** Each node sends data to exactly one randomly chosen destination and receives data from exactly one randomly chosen source, results a benign, low-load traffic condition;
- **Shift traffic:** Each node s sends traffic to node $(s+11)mod96$ and receives traffic from $(s-11)mod96$. As a result, all 11 servers of one switch communicates with all 11 servers of another switch over the same selection of k paths, resulting an adversarial traffic condition.

Figure 4.2 shows the link load statistics corresponding to the simulation of the aforementioned traffic patterns over an RRG (96,13) topology. From the maximum link load plots in all three traffic scenarios, I observe that KSP suffers from severe performance bottlenecks when it uses a limited number of paths per communication due to the unbalanced distribution of those paths over the topology links. The bottleneck gradually improves when more paths are employed to carry traffic, and finally achieves the lowest point when the number of path approaches the median short path availability of 20. This is precisely the issue with KSP where the optimal value of path count k is tightly coupled with the topology configuration and depending on the topology, the k-value is often too high to be feasibly implemented in a routing table.

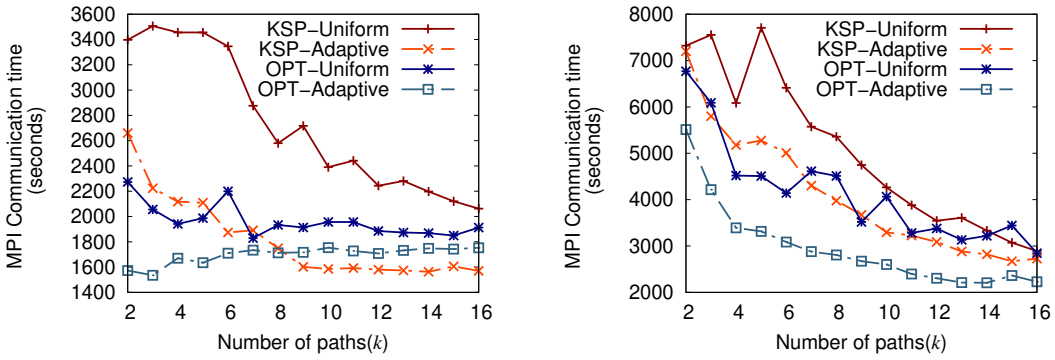
In contrast, Figure 4.2a shows the maximum link usage of my optimized path selection scheme on an All-to-All traffic. We can see that a lower bottleneck load, and

thus, a higher worst-case throughput performance can be achieved by my path selection scheme with a minimal number of paths considered per communication. The lowest maximum link load is achieved at $k = 5$ with my path selection scheme, which is only 28% of the maximum link load achieved by KSP with the same number of paths. For higher k-values, it is not possible to outperform OPT since the maximum link load of OPT converges with the average link loads. In the permuation pattern results in Figure 4.2b, the maximum link load corresponding of OPT with $k = 6$ is only 10% higher than the best achievable link load of KSP with as many as 26 paths per communication. Again if we limit the values of k to 6, then the bottleneck load of KSP is 67% higher than that of OPT and in other words, the bottleneck throughput of KSP becomes only 59% of the same in OPT. The maximum link loads on Shift traffic is again lower in OPT compared to KSP for lower values of k; although the performances converge quickly from $K = 10$ onward.

In Figure 4.2, The average link loads corresponding to both *KSP* and *OPT* are near identical, suggesting that the paths selected by the OPT algorithm are not necessarily longer than those chosen by KSP. Due to the high availability of short paths in the topology we consider, OPT simply selects an alternative set of short paths with better load balance characteristics than KSP, without having compromising on the path lengths.

4.3.2 Synthetic Trace Results

Figure 4.3 shows the total communication time obtained on synthetic workloads for four combinations of routing and path selection: k-shortest path selection with Uniform routing (KSP-Uniform), k-shortest path selection with UGAL adaptive routing (KSP-Adaptive), my optimized path selection with Uniform routing (OPT-Uniform) and my optimized path selection with adaptive routing (OPT-Adaptive). Figure 4.3a shows the performance of all schemes on random permutation traffic. Since random permutation patterns represent *benign workloads*, the majority of data traffic needs to be routed



(a) Random Permutation Traffic

(b) Adversarial Shift traffic

Figure 4.3: Communication Performance of 1024-rank synthetic workloads on RRG (96,13) topology

minimally to maximise the overall performance. Despite having a strong bias for shorter paths in this pattern, KSP-Uniform still struggles to perform well, due to the inherently poor load balance of short paths in RRG networks. KSP-Adaptive takes the advantage of using short paths and being able to balance load among paths, making it the overall best routing for $k = 9$ and above. However, an even better performance is achieved by my OPT-Adaptive at $k = 3$ and thus, at one-third of the routing table overhead. In cases where adaptive routing is not supported by switches, my OPT-Uniform still consistently outperforms KSP-Uniform.

Figure 4.3b shows the performances of the four schemes on adversarial traffic where load balance is of utmost importance to prevent degradation of communication performance. Here, the OPT-Adaptive scheme comfortably outperforms all competing schemes by up to 2.3x margin over KSP-Uniform and up to 1.6x margin over both KSP-Adaptive and OPT-Uniform when ≤ 7 paths are used per communication. Even without support for per-packet adaptiveness, OPT-Uniform performs comparably with KSP-adaptive due to its superior and load-balanced path selection.

4.3.3 Application Benchmarks Results

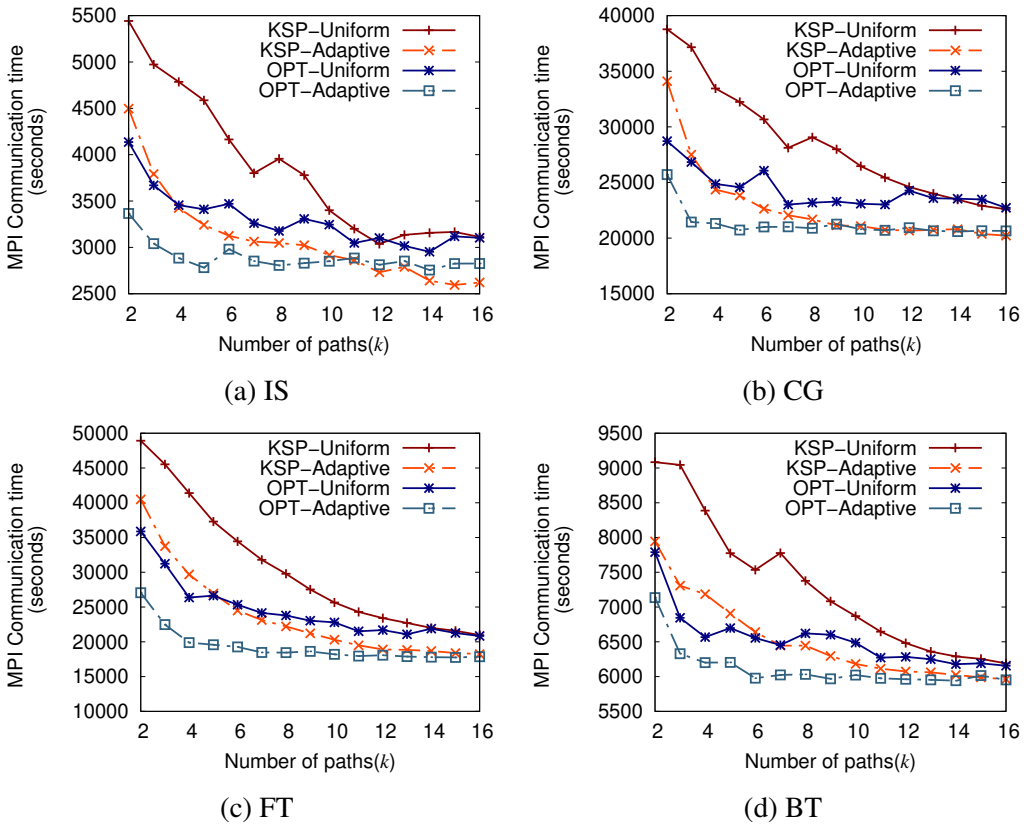


Figure 4.4: Communication Performance of NAS benchmark applications on RRG (96,13) topology

Figure 4.4 shows the average communication time of four application benchmarks using my four combinations of path selection and routing on an RRG(96,13). An interesting trend is noticeable from the Figure. In all four applications, optimization of path selection yields significantly lower communication time than the same obtained from k-shortest path selection when all routing paths are uniformly utilized. As the number of

k	Synthetic		Application				Avg.
	Perm.	Shift	IS	CG	FT	BT	
2	49.4%	8.1%	31.5%	34.4%	56.9%	27.7%	34.6%
3	70.6%	24.0%	35.4%	31.2%	40.0%	16.0%	36.2%
4	78.0%	34.6%	38.4%	17.6%	35.9%	14.9%	36.5%
5	73.8%	70.9%	34.4%	22.1%	31.6%	20.4%	42.2%
6	52.1%	54.8%	20.0%	25.3%	25.1%	11.4%	31.4%
7	57.2%	20.8%	16.5%	20.2%	19.5%	7.3%	32.3%
8	33.4%	18.7%	24.5%	14.6%	12.5%	5.9%	18.2%
9	41.9%	34.8%	14.2%	10.5%	12.9%	5.9%	20.0%
10	22.1%	4.9%	4.8%	1.3%	7.9%	3.1%	7.3%

Table 4.1: Communication time speedup (in percent) of optimized k-path selection over k-shortest path selection in an $RRG(96,13)$ using Uniform multi-path routing

paths per communication gradually increases, both path selection schemes have access to most, if not all, short paths in the topology. As as result, path selection plays a lesser role in determining interconnect performance when a large number of paths can be employed. The same trend is noticeable on adaptive routing as well, where my path selection optimization performs strictly better than KSP selection when $k \leq 10$ in all four applications, and at least as good as KSP in three out of the four applications when $k > 10$. Despite having very different communication patterns, the performance trends of the application benchmarks are surprisingly similar. This is most likely due to the randomness of the RRG topology where locality among communicating nodes are not preserved. This reveals that the RRG topology has somewhat smoothing effect over the application-level performance of its various workloads.

4.3.4 Result Summary

Tables 4.1 and 4.2 summarizes the performances of KSP and OPT during my trace-driven experiments. Table 4.1 shows the speedup of OPT-Uniform over

k	Synthetic		Application				Avg.
	Perm.	**Shift**	**IS**	**CG**	**FT**	**BT**	
2	69.1%	30.6%	33.6%	14.2%	49.2%	15.8%	35.4%
3	44.8%	37.5%	24.6%	14.9%	37.6%	11.3%	22.7%
4	26.7%	52.6%	18.7%	7.7 %	27.1%	11.0%	23.9%
5	29.0%	59.1%	16.5%	4.9 %	25.2%	6.9%	23.6%
6	9.5 %	62.2%	4.8%	3.7 %	20.4%	6.8%	17.9%
7	9.0 %	49.4%	7.3%	-0.4 %	13.9%	5.5%	14.1%
8	2.1 %	41.7%	8.6%	1.2 %	11.5%	2.6%	11.2%
9	-6.7%	37.4%	6.8%	0.2 %	8.5 %	2.3%	8.0%
10	-9.6%	26.6%	2.3%	-1.2%	4.5 %	1.8%	4.0%

Table 4.2: Communication time speedup (in percent) of optimized k-path selection over k-shortest path selection in an $RRG(96,13)$ using Adaptive routing

KSP-Uniform. As expected, communications using OPT-Uniform perform faster than KSP-Uniform by 31% of higher for the practical values of k less than or equal to 7. Even beyond that, OPT-Uniform continues to remain superior to KSP-Uniform while the performance margin gradually diminished. From the data in Table 4.2 showing the speedup margins of OPT-Adaptive over KSP-Adaptive being at least 22% for $k \leq 5$ and at least 11% for $6 \leq k \leq 8$. While KSP may theoretically perform as fast as OPT path selection under any traffic conditions, it would certainly require storage of a much wider set of paths, whereas OPT consistently achieves the near maximum achievable performance of KSP by using only between 3-to-8 paths between each switch pair.

4.3.5 Discussion

In this research, I formulate a new heuristic algorithm to find the best k paths between all switch pairs of a given topology. My algorithm is simple and practically implementable with a minimal amount of router memory footprint while it performs significantly faster than the well-adopted k-shortest path selection and routing scheme.

My experimental data empirically suggests my routing schemes to be capable of delivering high throughput performance, especially on a memory constrained randomized interconnect with high path diversity. I believe that my contribution mitigates a significant design challenge of randomized interconnect networks and thus, makes a small yet necessary step towards designing the next generation of large scale HPC clusters and data centers.

CHAPTER FIVE

PERFORMANCE COMPARISON BETWEEN STRUCTURED AND IRREGULAR NETWORK TOPOLOGIES

In HPC (High-Performance Computing) and data center systems, different network topologies have been proposed. Some topologies, like Fat Tree and Dragonfly have a structured or hierarchical design. Others, such as Jellyfish and Xpander, are unstructured and use random connections.

Similarly, the distributed applications we run in HPC and data centers can be categorized based on their communication patterns. Some applications, like stencil applications, have regular and predictable communication, while others, such as distributed memory graph applications, have irregular and unpredictable communication.

In this work, I provide a comparative analysis of various structured topologies against their unstructured counterparts, focusing on network capacity and cost. This analysis includes both regular and irregular applications using different routing methods. The goal is to determine whether irregular applications can benefit from unstructured topologies and to assess the suitability of these topologies for future HPC and data center systems in terms of performance and cost.

5.1 Methodology

Jellyfish topology is known for its flexibly in expanding/contracting interconnects in units of switches, adapting to the current scalability needs. In order to provide a fair comparison with Dragonfly, and Fat tree topologies, I construct Jellyfish RRG of equivalent costs and capacity (in terms of #switches and other network equipments). The following sections describe my methods for constructing the Jellyfish topology, followed by cost estimation and application scenarios considered for this research.

5.1.1 Constructing Jellyfish RRG

Hierarchical interconnects follow distinct organization. For example, a Fat tree follows a tree-like hierarchy, whereas Dragonfly adopts group-level hierarchy. Different types of hierarchy provide us with different degrees of flexibility in expanding the network, making it challenging to build equivalent networks in each topology in terms of network resources and cost. In comparison, Jellyfish is relatively flexible, and expansion is governed by expander graph properties. In my study, I construct equivalent Jellyfish topologies regarding resources and cost for Fat tree and Dragonfly topologies to provide a fair comparison between the networks. The configurations of the topologies are presented in Table 5.1. In my evaluation, I use a Dragonfly topology, denoted as DFLY(17,9,8,8); the network consists of 17 switches per group, 9 groups, and 8 computing nodes per switch. The total number of switches is 153. Each switch has 16 ports connecting switches within a group and 8 ports connecting switches to the other 8 groups totaling 24 switch-to-switch ports per switch. The Jellyfish equivalent, RRG(153,24), has 153 switches, each with 24 switch-to-switch ports.

Topology	Equivalent JellyFish	#Switches	Switch Radix	#Nodes	#Ports
Dragonfly DFLY(17,9,8,8)	RRG(153,24)	153	32	1224	3672
Fat Tree FT(3,48,2)	RRG(120,38)	120	48	1200	4560
Dragonfly DFLY(13,7,6,6)	RRG(91,18)	91	24	546	1638
Dragonfly DFLY(13,5,4,4)	RRG(65,16)	65	20	260	1040

Table 5.1: Topology configurations used in the experiments

For the Fat tree, I use an FT(3,48,2) comprised of 3 levels and 2 pods. Each switch has 48 ports. There are 120 switches across all three levels; level 0 and level 1 have 48, and level 2 has 24. At level 0, each switch connects 24 compute nodes to one of the 24 switches in level 1. I configured an equivalent Jellyfish topology RRG(120,38) with 120 switches and 38 switch-to-switch ports per switch. The switch-to-switch ports per switch for Jellyfish is calculated by dividing the total number of available switch-to-switch ports in the Fat tree by the total number of switches.

5.1.2 Cost Estimation

I perform network interconnection cost estimation comparing Dragonfly and Fat Tree topologies with equivalent Jellyfish RRG for 1000+ nodes (see Table 5.1). Designating short and long distance links is crucial in overall cost estimation. Practical network interconnects must consider a mix of electrical (usually for link connections within 3 metres or less) and fiber-optic cables (for rack-spine global connections, beyond 3 metres of link distances) for cost optimization and efficiency.

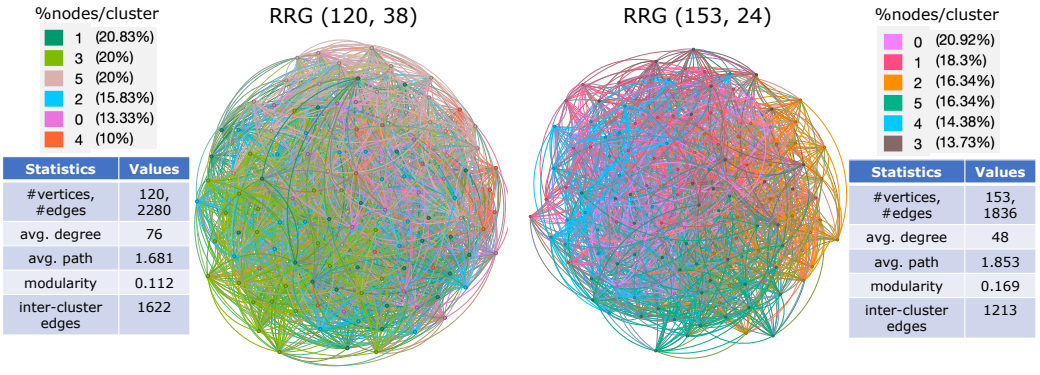


Figure 5.1: RRG(153,24) and RRG(120,38) graph statistics using Gephi [3]

I consider Jellyfish RRG topology as an undirected graph, where the switches represent nodes, and an edge represents the actual link between a pair of switches. Graphs demonstrate structures referred to as "communities" or "clusters" consisting of densely connected groups of vertices, which are sparsely connected with rest of the vertices outside a particular cluster. Despite clusters for ad hoc graphs are not well defined (no ground truth) and popular modularity maximization methods for partitioning a graph into clusters are speculated to be NP-hard [35], I perform Louvain graph clustering [36] of the RRG graphs to categorize links as intra or inter-cluster. Fig. 5.1 demonstrates the connectivity properties of the (undirected) RRG graphs, noting RRG(120,38) has about 30% more switch radix than RRG(153,24) (see Table 5.1), leading to more connections. However, both of these graphs exhibit a similar structure and connectivity (as depicted by the close modularities, average path lengths and #clusters, as per Fig. 5.1 inset). In both the cases, even after excluding the intra-cluster edges, about 70% of the nodes are still reachable.

I suppose the inter-cluster edges to be connected using varied length fiber-optic cables (with integrated transceiver modules for simplicity). Hence, I designate the intra-cluster edges to be proportional to the number of short distance (about 3 metres long) fiber optics cables, and inter-cluster edges correspond to the longer fiber optics cables required to construct an RRG interconnection network. Price variation between (short length) electrical and fiber optics cables are about $3\times$ presently, and subsequent longer variants of fiber optics cables can vary in costs at least by about 10%[1]. I consider switch hardware costs to be proportional to the number of ports, assuming $1000/port after consulting with the current prices of 400Gb/s Infiniband switches[2].

[1] `https://www.colfaxdirect.com/store/pc/viewCategories.asp?idCategory=2`

[2] `http://www.colfaxdirect.com/store/pc/viewPrd.asp?idproduct=4163&idcategory=0`

Topo.	#Swch.	#Rad.	Short (∼3m), node-switch, switch-switch		Long (>= 5m), switch-spine	Est. ($M)
			Elec. ($100)	Optics ($300)	Optics ($330)	
DFLY (17,9,8,8)	153	32	1224	1224	612	5.58
FT (3,48,2)	120	48	1152	1152	1152	6.60
RRG (153,24)	153	32	1224	623	1213	5.60
RRG (120,38)	120	48	1200	658	1622	6.61

Table 5.2: Cost estimation of Dragonfly and Fat Tree in comparison to equivalent Jellyfish RRG topologies with reasonable assumptions. For e.g., cost estimation for Dragonfly considering $1000/port is calculated as: $\$1000 * 153 * 32 + \$100 * 1224 + \$300 * 1224 + \$330 * 612 = \$5,587,560$.

Table 5.2 lists my price estimation with the cabling assumptions, confirming that the Dragonfly and Fat tree costs are complementary to the Jellyfish RRG equivalents. Cost (of more and longer cables) has remained a critical barrier for unstructured interconnects like Jellyfish RRG, despite enhanced load balancing and optimized multipath selection mechanisms which makes them attractive for applications with irregular communication and low-latency needs. With technology advancements in Optical Circuit Switching and composable disaggregated infrastructures, optical cables are positioned to be more affordable and advantageous in the near future [37, 38]; and programmable network switches (e.g., SmartNICs [39]) can support the complex routing mechanisms of RRG. My preliminary cost estimation indicates that Jellyfish RRG is a viable candidate for consideration in actual deployments.

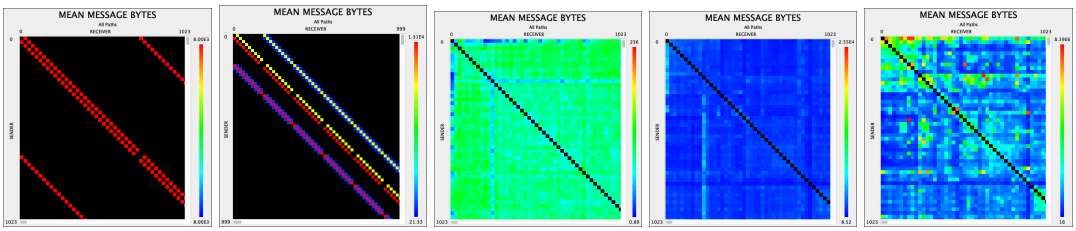


Figure 5.2: Mean pair-wise message volumes (black denotes zero bytes exchanged) on 1024 processes (from left to right, applications considered: SWEEP3D, LULESH, BFS, Clustering and Triangle Counting). Graph applications demonstrate significant irregularity in the communication.

5.1.3 Application Scenarios

Different applications shows different communication properties between its parallel process. In this section I presented the communication behaviour of various stencil and graph applications I considered for this research. The inherent unstructuredness of the input graphs in distributed-memory graph workloads can lead to significant differences in the underlying process graph topologies (i.e., the graph inferred by distributing an application graph across a range of processes), manifesting diverse and irregular process neighborhoods. Conversely, for regular stencil applications like LULESH [40] and Sweep3D [41], the process neighborhood remains fixed regardless of the number of processes used due to the underlying fixed cartesian topology. Fig. 5.2 highlights this difference by depicting the transitions in process neighborhoods across three different distributed-memory MPI-based graph workloads – graph clustering, breadth-first traversal, and triangle counting running on 256, 512, and 1024 processes (all adopting a 1D vertex-based graph distribution), compared to LULESH running on 216, 512 and 1000 processes respectively, and Sweep3D on 256–1024 processes.

I use graphs from the SuiteSparse collection [1] as my inputs (listed in Table 5.3), and further convert them into a Compressed-Sparse Row based binary format to minimize

the file I/O. Although LULESH and Sweep3D are packaged with *SST/Macro*, the graph applications used in my evaluation are external and required significant adaptations prior to simulation. I also carefully selected the graphs–overall, I have three classes of graphs: real-world graphs (e.g., social network, web crawl, etc.), Random Geometric Graphs (RGG) and Stochastic Kronecker graphs. Real-world and Random Geometric Graphs are

Table 5.3: Graphs (from Suitesparse [1]) used in our evaluation.

Graphs	**|V|**	**|E|**	**Max Deg.**	**Applications**
Real World Graphs				
com-orkut	3M	234M	245K	Clustering
stokes	11M	349M	341K	Clustering
uk-2002	18M	298M	292K	Clustering
Random Geometric Graphs (RGG)				
rgg_n_2_22_s0	4M	60M	237K	Clustering, BFS
rgg_n_2_23_s0	8M	127M	248K	Clustering, BFS
rgg_n_2_24_s0	16M	265M	258K	Clustering, BFS
Stochastic Kronecker Graphs				
kron_g500-logn16	65K	5M	20K	Triangle Counting
kron_g500-logn17	131K	10M	37K	Triangle Counting
kron_g500-logn18	262K	21M	69K	Triangle Counting
kron_g500-logn23	4M	517M	710K	BFS
kron_g500-logn24	8M	520M	663K	BFS
kron_g500-logn25	17M	1046M	1.2M	BFS

suitable for clustering (they have community structure, so graph clustering runs for several iterations until convergence unlike the Kronecker graphs), whereas the Kronecker graphs are more suitable to assess BFS and triangle counting (I also used RGGs for BFS).

The structure of the graphs play an important role in the communication; graphs can exhibit the *small world* phenomenon (relatively short path lengths between nodes and

distinct clusters of nodes, e.g., social networks such as orkut) or *scale free*, where a small amount of nodes/vertices have a high degree of connections (i.e., more edges, power law distribution; e.g., kronecker graphs). Community detection or clustering exhibit frequent communication and synchronization, using variable-sized messages (TAU profile on Fig. 5.2 shows about 8B–35KB mean messages on 8 nodes, 256 processes). The BFS implementation is similar to the reference Graph500 implementation, and uses fixed small buffered communication (256 bytes). Whereas, graph triangle counting also uses a buffered communication, but a larger buffer of size 1MB, due to high intermediate data (due to explicit enumeration of possible edges). All the applications use MPI non-blocking point-to-point communication with a mix of MPI collective communication API.

5.2 Performance Comparison

In this section I discussed the performance comparison of different hierarchical or structured topology with its equivalent Jellyfish RRG topology. The performance metrics used in this comparison is already discusses on background chapter. I begin this section by comparing the range of stall queue lengths (which affect the performance) under various routing schemes in §5.2.1. Then, I study communication characteristics of the applications by analyzing the #hops traversed and the amount of contention (i.e., stalls) encountered per packet on the critical path by comparing Jellyfish with Fat tree in §5.2.2 and Dragonfly in §5.2.3. I review the communication time performance of the fixed-neighborhood (stencil) and graph applications in §5.2.4. Scalability of Jellyfish and Dragonfly on 256-1024 ranks is discussed in §5.2.5. Then I make some high-level observations in §5.2.6.

5.2.1 Impact of routing schemes on the topologies

To study the overall impact of routing schemes on stall queue length, I combined the information over five application scenarios with varying input graphs for each routing scheme under Jellyfish, Dragonfly and Fat tree topology. Since the results are aggregated the actual number of stalls will vary for the different application scenarios. Fig. 5.3 (left) shows that RRG Adaptive Multipath (AM), RRG MIN, RRG Uniform Multipath (UM), Dragonfly Valiant, Dragonfly Progressive Adaptive Routing (PAR), Dragonfly MIN and Dragonfly UGAL have median stall queue lengths of 20, 15, 26, 35, 39, 38 and 33 respectively.

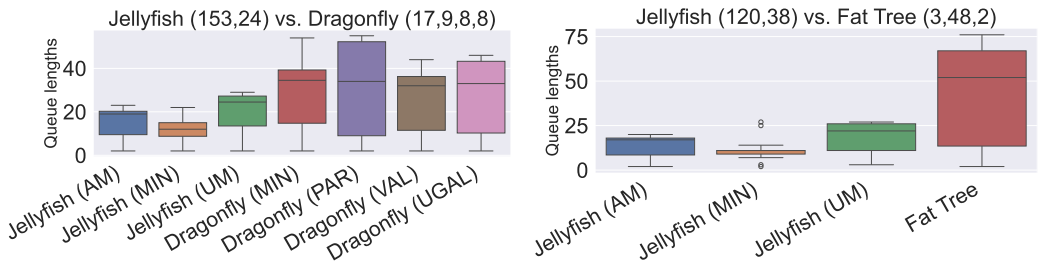


Figure 5.3: Network routing schemes and impact on observed stall queue lengths.

This demonstrates that all the three routing variants of RRG experience almost 15–60% lower contentions as compared to the Dragonfly routing schemes. I also compare the aggregate stalls for various Jellyfish routing schemes with the equivalent Fat tree in Fig. 5.3 (right), demonstrating the median stall queue lengths of RRG AM, RRG MIN, RRG UM and Fat tree routing as 16, 10, 23 and 52 respectively. Hence, these three routing variants of RRG exhibit about 40–80% lower contentions as compared to the Fat

tree routing scheme. Considering the ranges in Fig. 5.3, RRG Adaptive Multipath (AM) demonstrates overall superior performance compared to rest.

5.2.2 Analyzing stalls and hops in Fat tree and equivalent Jellyfish

In Fig. 5.4, X-axis represents the #hops traversed and the amount of contention encountered (i.e., delay or stalls per packet). The Y-axis represents the % of packets. Fig. 5.4a depicts LULESH, where I observe that Jellyfish AM and MIN demonstrates about 35% and 45% less contention respectively. For SWEEP3D, in Fig. 5.4b, Jellyfish AM improves contention by about 40%, MIN by about 30% and UM exhibits greater than 10% improvement in the maximum contention compared with Fat tree.

Fig. 5.4c represents the result for graph clustering, aggregated over three real-world graphs (Table 5.3). For graph clustering, Jellyfish with AM routing reduces contention by greater than 70%, MIN by greater than 75% and UM also reduces contention by more than 60% as compared to Fat tree. Similarly, Fig. 5.4d shows benefit with Jellyfish AM (>70%), MIN (>75%) and UM (>40%) for BFS. For graph triangle counting, we observe from Fig. 5.4e that Jellyfish AM , MIN and UM offers more than 75% , 80% and 65% improvements to the maximum contention in comparison to Fat tree routing.

5.2.3 Analyzing stalls and hops in Dragonfly and equivalent Jellyfish

Fig. 5.5a shows the statistics of LULESH on Dragonfly and the equivalent Jellyfish for various routing schemes. We observe that packets on Dragonfly using UGAL have greater than 5% lower maximum contention as compared to other routing schemes. However, the maximum contention value alone is insufficient to explain the performance range of myriad applications on different topologies; hence, I plotted the distributions across the path of packet traversals (i.e., *stall queue*). From the distribution of the number

of packets, we can see that Adaptive Multipath (AM) routing on Jellyfish has the highest amount of packets that are encountering reasonably low amounts of contention, less than 5 as compared to the rest. In Fig. 5.5b, Sweep3D demonstrates greater than 25% lower maximum contention than the rest with Dragonfly PAR routing. However, this trend does not repeat for the graph applications.

For graph applications on Jellyfish vs. Dragonfly evaluations, I provide aggregated metrics like the previous Fat tree experiments. Packets in Jellyfish AM routing encountered more than 50% lower maximum contention compared to Dragonfly, and similarly, the amount was 60% and 25% in case of Jellyfish MIN and Jellyfish UM routing respectively.

Fig. 5.5d presents the results for BFS graph traversal, aggregated over three kronecker graphs (scales 23–25). I observe again that Jellyfish AM exhibits more than 40% lower maximum contention as compared to Dragonfly, and like the previous case, noted an improvement of about 60% and 20% for Jellyfish MIN and Jellyfish UM. Like BFS, the graph triangle counting results were aggregated over three smaller kronecker graphs (scales 16–18). As shown in Fig. 5.5e, Jellyfish AM exhibits higher than 35% reduction in the maximum contention as compared to Dragonfly, and Jellyfish MIN and UM are better by about 50% and 25% respectively. The contention analysis indicates that Jellyfish and Dragonfly demonstrates similar behavior for regular stencil applications (LULESH and SWEEP3D), but Jellyfish exhibits lower contentions compared to the equivalent Dragonfly for irregular graph applications.

Simulation results indicate that for both the stencil and graph application scenarios, the latter was able to improve contention significantly with Jellyfish.

5.2.4 Analyzing MPI Communication Times

I demonstrate the (estimated) MPI communication times of different routing algorithms in Dragonfly and Fat tree with their equivalent Jellyfish topologies in Figures 5.6 and 5.7 for fixed-neighborhood (e.g., stencil) and graph applications. Fig. 5.6 shows the MPI communication time of LULESH, where Jellyfish AM delivers more than 15%, 8%, 49%, 15%, 18% and 12% improvements as compared to Jellyfish MIN , Jellyfish UM and the Dragonfly routing schemes, respectively. Comparing Jellyfish with Fat tree for LULESH, both AM and UM improves the communication times by greater than 20%. For SWEEP3D, Jellyfish AM performance matches with that of Dragonfly PAR (both exhibit the best results). Jellyfish AM and UM demonstrate improvements by 29% and 22% respectively for SWEEP3D, as compared to Fat tree.

For BFS, Jellyfish AM , MIN and UM depict improvements by 19% , 18% and 13% (over a range of input graphs) as compared to Fat tree (Fig. 5.7). In BFS, Jellyfish routing schemes demonstrate improvements by 20–32% as compared to Dragonfly. Jellyfish AM , MIN and UM depicts performance enhancements of greater than 25% for graph clustering as compared to Fat Tree. Jellyfish schemes also exhibits between 30–65% improvements as compared with Dragonfly for clustering. Due to the variations in iterations to convergence at scale (see Table 5.4), I report the per-iteration time. For graph triangle counting, Jellyfish exhibits up to 16% improvements as compared to Dragonfly, and up to 6% relative to Fat tree.

5.2.5 Scalability of graph applications on Jellyfish and Dragonfly

I discuss scalability of the graph applications in this section considering three configurations: 260, 546, and 1224 nodes topology (Table 5.1) to run the applications on 256, 512, and 1024 MPI ranks. For weak scaling in graph clustering I consider the random geometric graphs, whereas for strong scaling I used com-orkut social network. For weak

scaling scenarios in graph traversal and triangle counting I used the kronecker graphs. I keep routing method fixed for both Dragonfly and Jellyfish in this case, and used Progressive Adaptive Routing (PAR) for Dragonfly and Adaptive Multipath Routing (AM) for Jellyfish. I observed performance variability due to dissimilar #iterations to

MPI ranks	**Estimated Run Time (s)**	**#Iterations**
256	0.64	37
512	0.47	39
1024	0.38	48

Table 5.4: Variability of number of iterations to convergence in graph clustering on 256–1024 MPI ranks

convergence for graph clustering at scale. This nondeterminism is a known problem with parallel heuristics for graph clustering, and affects the convergence and quality [42]. I depict the variations on Table 5.4 for the com-orkut input, and adjust the total estimated times by dividing the #iterations. In Figures 5.8 and 5.9, I demonstrate the scalability in terms of queue lengths and estimated times. Jellyfish depicts better performances for both strong and weak scaling cases.

5.2.6 Overall performance assessment

On average, my applications demonstrate non-trivial latency along the critical path, as represented by the stalls suffered by the packets (as shown in Figures 5.5 and 5.4). The graph applications demonstrate up to 2$\times$ longer tail latencies relative to the stencil applications at the packet-level, but even for our most bandwidth-bound application (Sweep3D), about a quarter of the packets experience several hops along the

critical path to destination (refer to Figures 5.5b and 5.4b). Increasing the packet injection bandwidth does not affect the average path length, so it is unlikely that even our most bandwidth-bound application will improve if the switch bandwidth is increased substantially.

Fig. 5.10 displays the range of performance improvements (in terms of estimated MPI communication times) of Jellyfish RRG (using MIN, AM and UM routing) against the equivalent Dragonfly/Fat tree by considering all combinations of the application scenarios and routing methods across the process configurations used in our evaluation. For fairness I exclude comparisons between minimal and adaptive routing schemes across topologies.

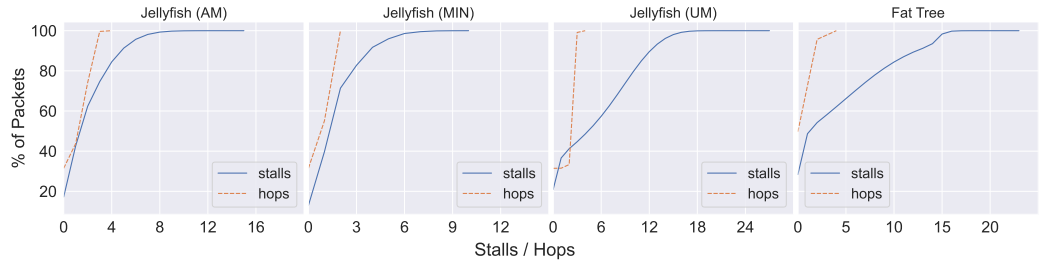
I observed on average about 12% improvements in the MPI communication time for all applications and input combinations with Jellyfish RRG adaptive routing as compared to rest of the routing schemes across the topologies considered. RRG Uniform Multipath (UM) yielded on average about 11% improvements against the appropriate topology/routing combinations, whereas RRG minimal routing (MIN) demonstrated an average improvement of about 9%.

Major performance variations across applications with the same input graphs is not uncommon. For e.g., BFS on Kronecker graphs exhibit average improvements of 10% with Jellyfish RRG as compared to others, but when the same Kronecker graphs are used in graph triangle counting, the mean improvement is observed to be about 3%. Also, due to differences in the structure of the underlying process graphs (an effect of input graph distribution across processes), specific network routing schemes might appear to be more favorable in terms of performance. For e.g., considering BFS on RGG, the best routing scheme across the topology configurations is Dragonfly Valiant. However, considering the same RGG input for graph clustering, RRG Adaptive Multipath (AM) outperforms the rest.

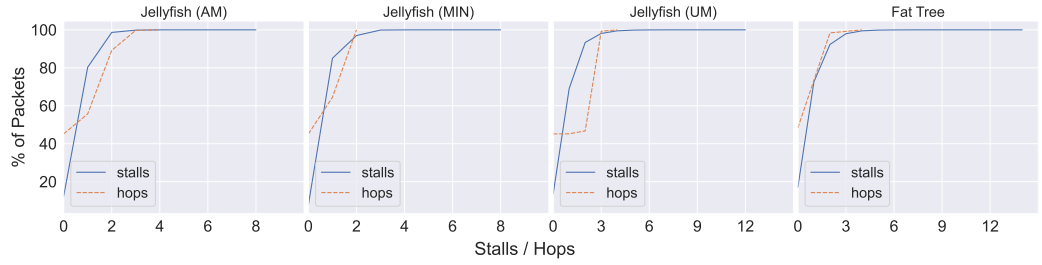
To determine the overall performance characteristics of distributed-memory graph applications, one must consider the application pattern, input graph structure, and, network routing schemes for a particular topology. Performance degradation of Jellyfish RRG MIN as compared to its Dragonfly counterpart expose the limitations of RRG minimal routing. In Jellyfish, there are a limited number of single-hop paths (i.e., *shortest* path) between arbitrary source-destination switches, remaining paths being multi-hop (still relatively short, as expander graphs have low diameters). Since minimal routing only uses the *shortest* paths, it cannot leverage the (short length) path diversity of Jellyfish (as shown in Fig. 4.1). For this reason, multi-path routing schemes such as Uniform Multipath (UM) or Adaptive Multipath (AM) are usually the practical choices for Jellyfish, as evidenced by their sustainable performance improvements relative to Dragonfly/Fat tree.

5.2.7 Discussion

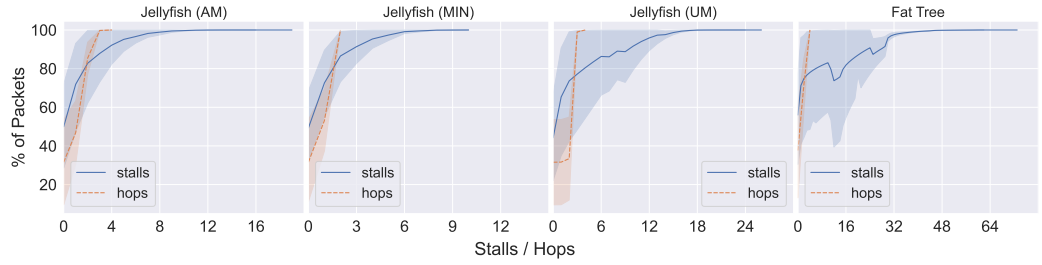
In this research I have shown that unstructured interconnects such as Jellyfish can improve the overall scalability relative to existing Dragonfly and Fat tree interconnects without raising the equipment costs. The path diversity in Jellyfish RRG can drive the performance of communication intensive complex graph applications, which are susceptible to communication bottlenecks at scale. I analyzed packet-level simulations of graph applications (and two popular stencil-based mini-applications) on Jellyfish topologies, using various routing schemes, and exhibit on average about 4–20% and 3–25% end-to-end performance improvements across applications as compared to equivalent Dragonfly and Fat tree topologies. I also showed the impact of a variety of different graph inputs, since graph structure is directly related to the communication performance at scale. Overall, my detailed stalls and hops analysis sheds new light on the performance of complex graph applications at various scales.



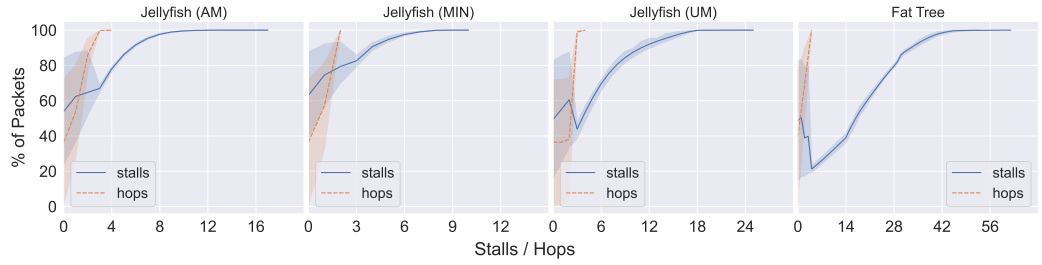
(a) LULESH (1000 ranks, problem size: 50)



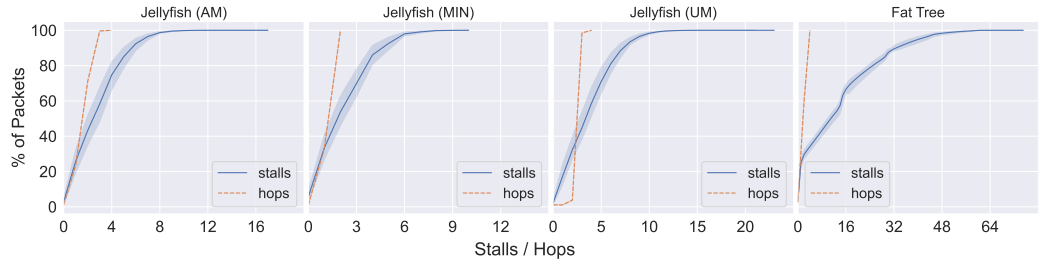
(b) SWEEP3D (32× 32 grid size)



(c) Graph Clustering

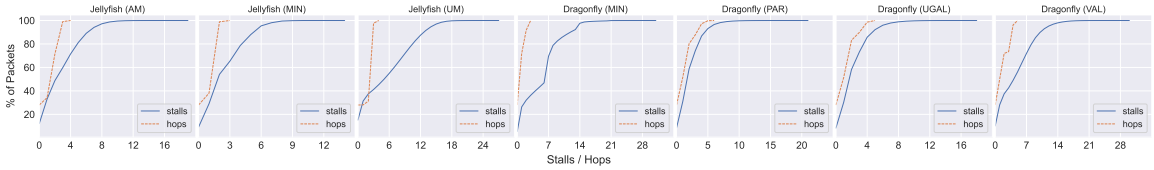


(d) Graph BFS traversal

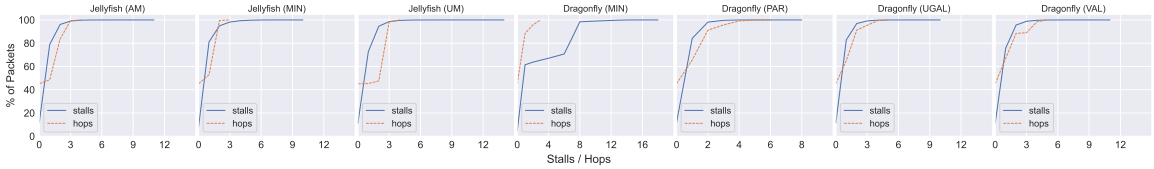


(e) Graph triangle counting

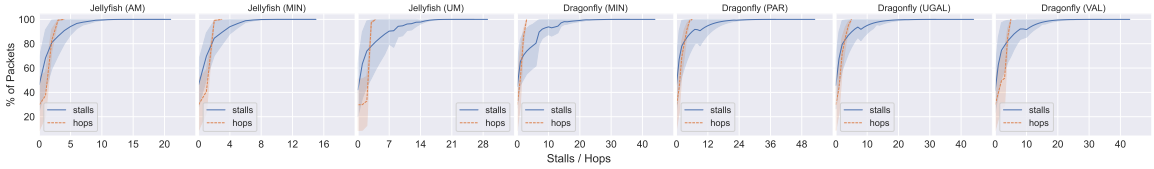
Figure 5.4: Hops/Stalls – Jellyfish vs. Fat Tree on 1024 ranks. **X-axis:** #hops & #stalls, **Y-axis:** % of Packets



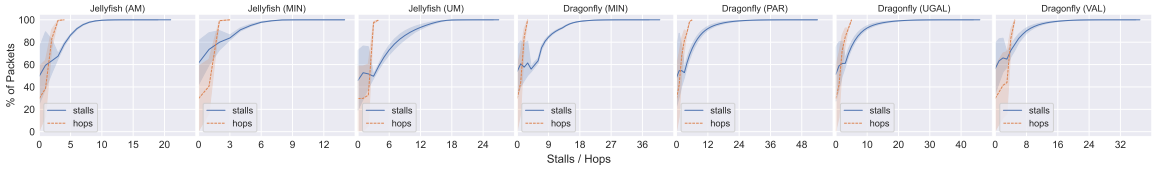
(a) LULESH (1000 ranks, problem size: 50)



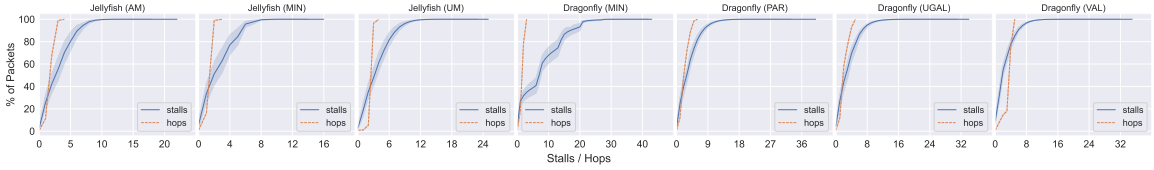
(b) SWEEP3D (32× 32 grid size)



(c) Graph Clustering



(d) Graph BFS traversal



(e) Graph triangle counting

Figure 5.5: Hops/Stalls – Jellyfish vs. Dragonfly on 1024 ranks. **X-axis:** #hops & #stalls, **Y-axis:** % of Packets

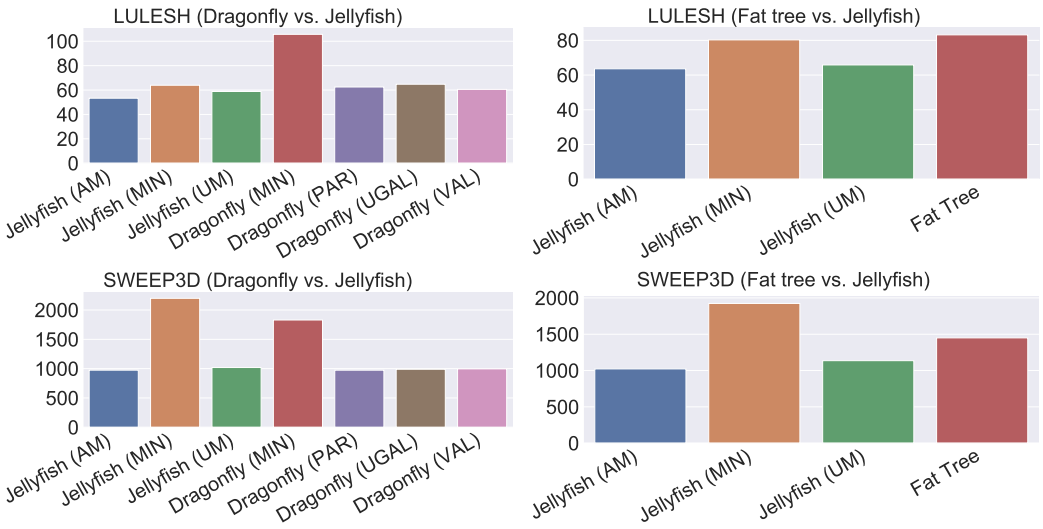


Figure 5.6: MPI times (X-axis) of fixed neighborhood applications on 1024 ranks – LULESH and Sweep3D

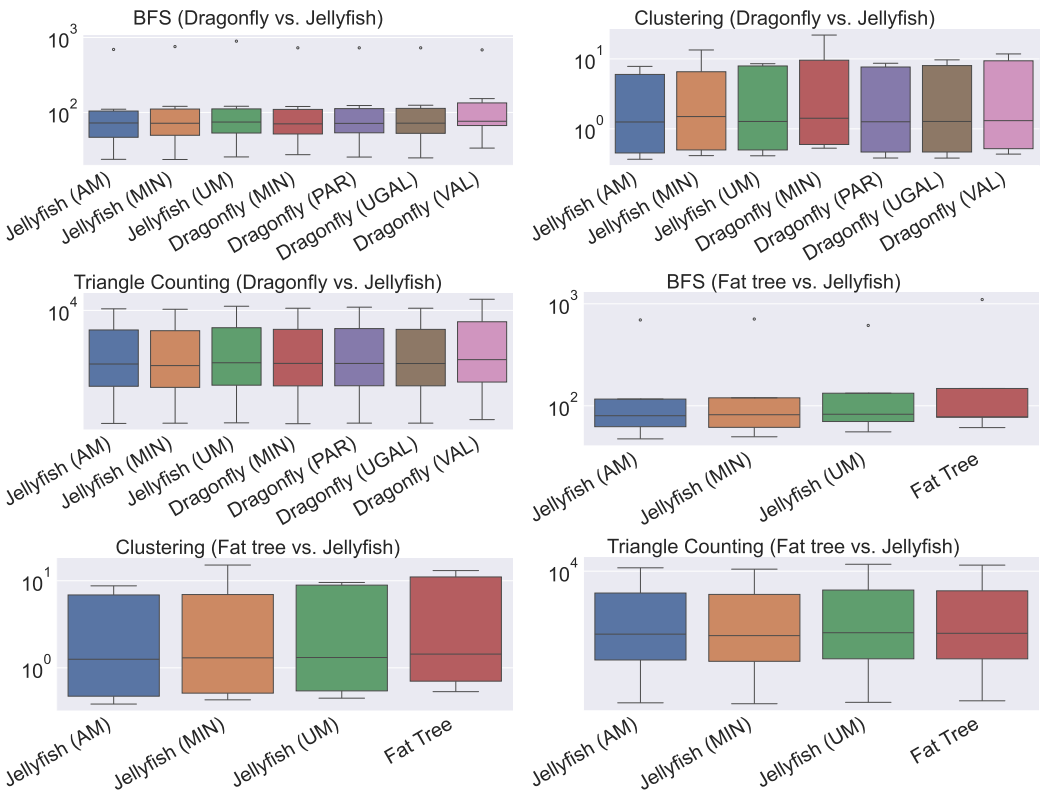


Figure 5.7: MPI times (X-axis) of graph applications on 1024 ranks – BFS, Clustering and Counting

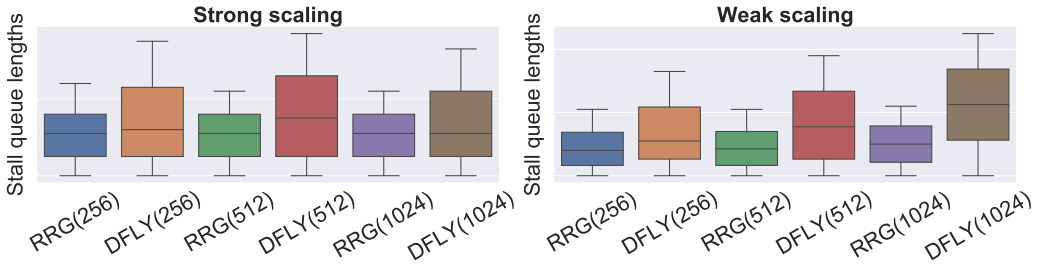


Figure 5.8: Scalability of Jellyfish (AM) vs. Dragonfly (PAR) for graph applications in terms of queue lengths on 256–1024 ranks.

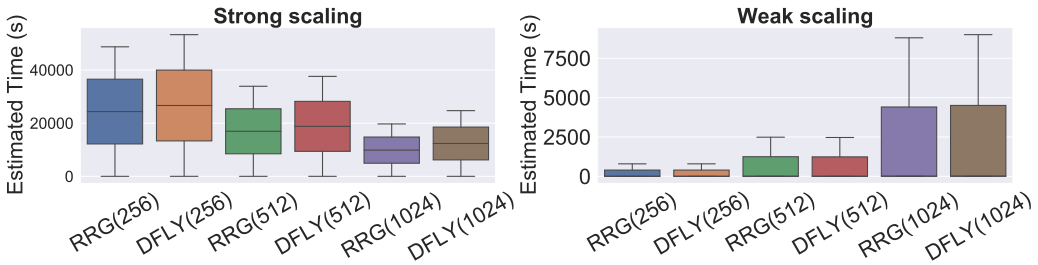


Figure 5.9: Scalability of Jellyfish (AM) vs. Dragonfly (PAR) for graph applications in terms of estimated times on 256–1024 ranks.

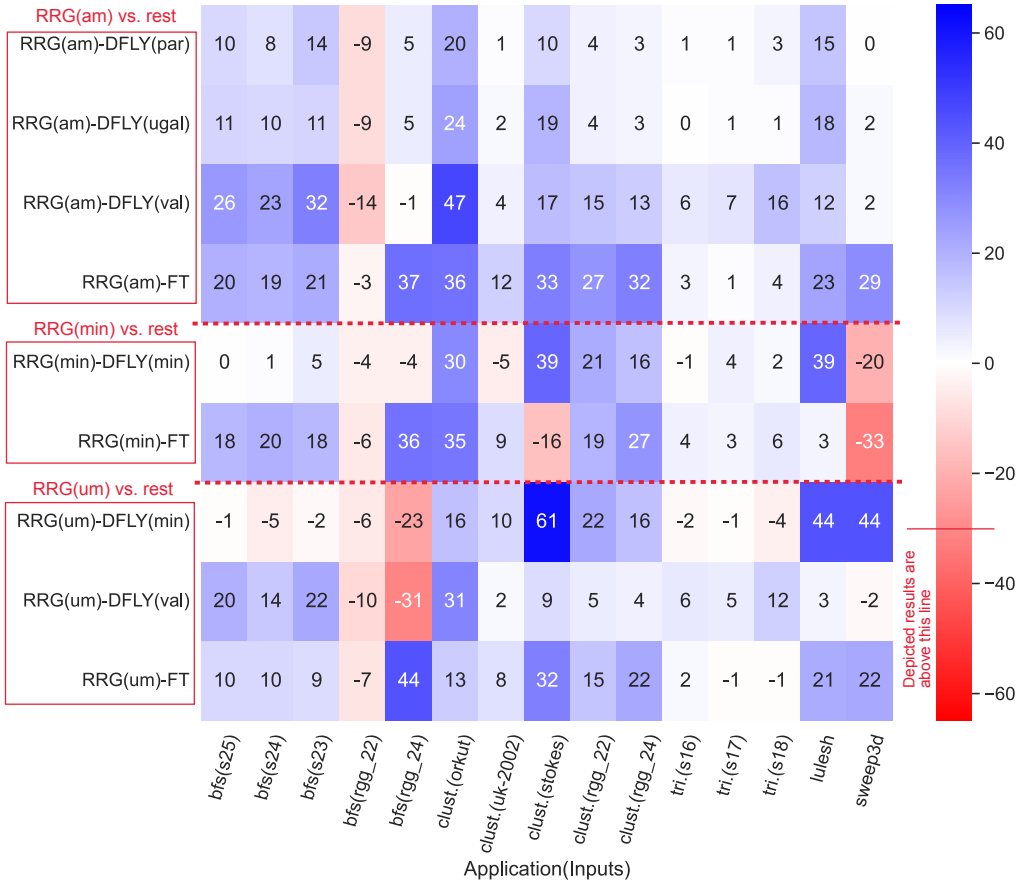


Figure 5.10: Percentage improvement of MPI time (X-axis) in Jellyfish RRG compared to relevant Dragonfly and Fat Tree counterparts for various application scenarios (Y-axis). Negative values indicate subpar RRG performance.

CHAPTER SIX

OPTIMIZED SCHEDULING FOR DISTRIBUTED MEMORY GRAPH APPLICATIONS ON DRAGONFLY TOPOLOGY

Distributed-memory graph applications, such as graph traversal, pattern finding, and cluster detection, often face challenges due to communication and synchronization delays. In these applications, parallel processes need to exchange data of varying sizes with their neighboring processes. This communication pattern is more complex to optimize compared to traditional rectangular problem layouts, making it harder to enhance efficiency.

While process assignment or remapping can improve communication performance regardless of how the data is divided, most existing solutions focus on simple grid layouts rather than the more complex graph topologies. In this work, I propose new methods for assigning processes in a way that improves communication for distributed-memory graph applications by taking advantage of the graph structures.

For four communication-heavy tasks—Breadth First Search (BFS), Louvain Clustering, Triangle Counting, and Single Source Shortest Path (SSSP)— I show that my proposed methods can improve MPI communication times by about 30-40%. I validated these strategies through packet-level simulations using the Structural Simulation Toolkit (SST), primarily on a Dragonfly network topology, and tested them on the HPE Slingshot network of the NERSC Perlmutter supercomputer.

6.1 Process remappings for graph workloads

In this section, I discuss the overall design and methodology of my MPI process reordering scheme. First, I briefly describe my vertex-based graph partitioning strategy,

since my schemes are based on the implicit process graph. Then I review the options and methodologies for generating process rank orders for graphs.

6.1.1 Vertex-based graph partitioning

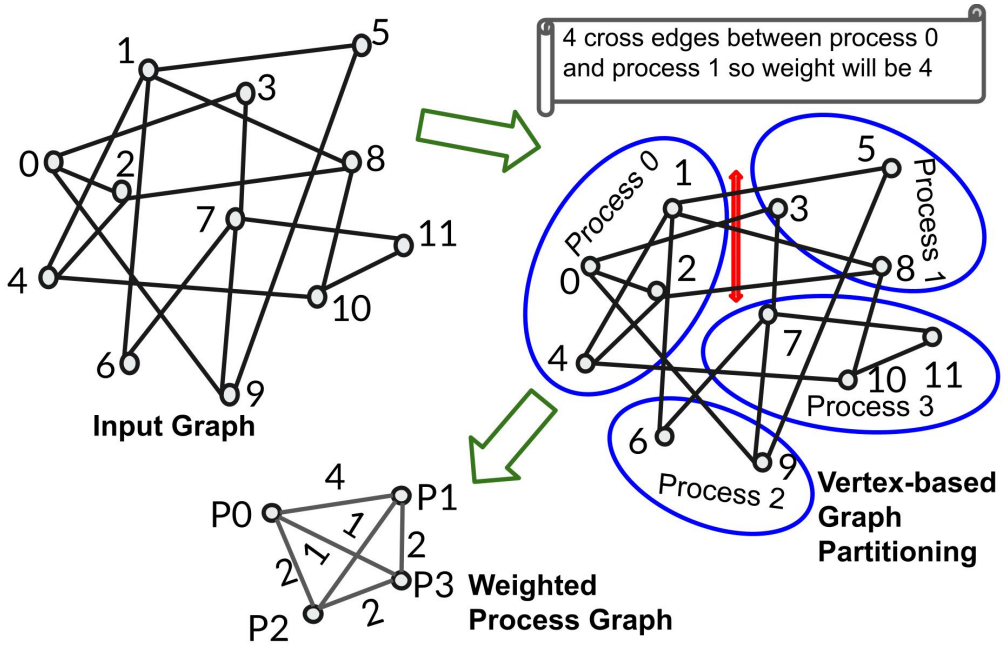


Figure 6.1: Generating *weighted* process graph from an input graph after vertex-based partitioning.

The default vertex-based graph partitioning splits graph vertices equally among the processes (i.e., about $|V|$/p, as demonstrated in Fig. 6.1), but can lead to high load imbalance due to disparity in the number of edges per process depending on the graph structure. I use a variant of the vertex-based partitioning, where I try to balance the #edges/process, leading to fewer cut-edges (a relatively compact process graph) and possibly dissimilar #vertices/process. A designated process first reads the $\mathcal{O}(|V|)$ list of

vertex degrees from the binary graph, calculates vertex partitions per process following Next Fit bin packing (use a new bin to insert an item only if the previous bin is full) [43]. Since the number of vertices is significantly less than the number of edges in a graph, we incur a minor serial overhead for this method, at an extra cost of about 10% in loading the graph. However, with the large graphs the edge-balanced partitioning method results in major communication avoidance (leading up to 80% improvement in execution times [44]), minimizing the impact of the initial serial overhead. Also, apart from performance, the balanced scheme also ensures equitable memory overhead per process.

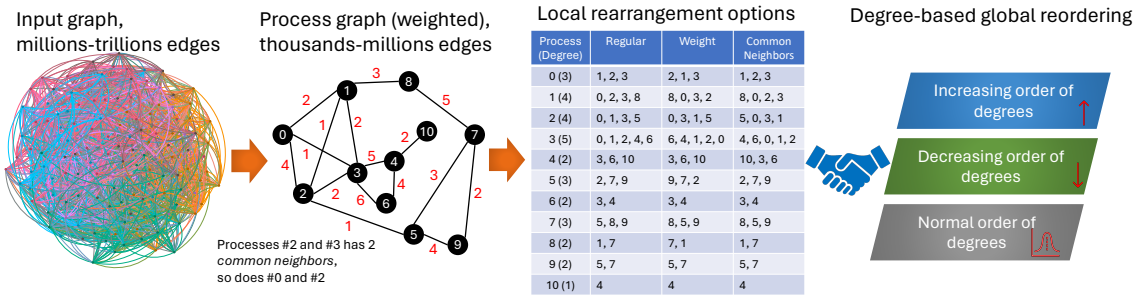
6.1.2 Process remapping heuristics considering graph structure

The default process order adopts a method in which MPI processes or ranks are assigned to consecutive physical processor cores, known as SMP (shared-memory)/block order. While this choice might be sufficient for cartesian or collective-based topologies, for irregular graph topologies, there can be substantial performance benefit by adopting a different strategy for process assignment to processor cores (a.k.a process/rank remapping or reordering, or task placement). The overall idea is to access the process graph (holding portions of the distributed input graph, see Fig. 6.1) in certain order for generating custom rank orders, particularly considering following two steps:

- **Local rearrangement**: rearrange processes in the (local) process subgraph (immediate process neighbors); aside from the default "regular" scenario (which assumes the process neighbors or edges listed in increasing order of numbering), I additionally propose two options for rearrangement: based on *weights* (#application edges shared between adjacent processes, indicative of communication volume across processes) [*intuition*: which are the neighbors with whom I share common traits (who is my friend?)], and, based on *common process neighborhoods* [*intuition*: which are the neighbors who are also neighbors of each other (who are

the friends of my friends?)]. The goal here is to improve the locality of processes within a neighborhood.

- **Global reordering**: after local rearrangement, I deal with the disparities across process neighborhoods (for e.g., varying degrees) to improve the locality of neighborhoods. Thus, I traverse the process neighborhoods in certain order (considering regular consecutive traversal (0-p, where p is the #processes in the process graph topology), or in terms of increasing, decreasing or normal distribution of the process node degrees), eliminating repetitions.



Process (Degree)	Regular	Weight	Common Neighbors
0 (3)	1, 2, 3	2, 1, 3	1, 2, 3
1 (4)	0, 2, 3, 8	8, 0, 3, 2	8, 0, 2, 3
2 (4)	0, 1, 3, 5	0, 3, 1, 5	5, 0, 3, 1
3 (5)	0, 1, 2, 4, 6	6, 4, 1, 2, 0	4, 6, 0, 1, 2
4 (2)	3, 6, 10	3, 6, 10	10, 3, 6
5 (3)	2, 7, 9	9, 7, 2	2, 7, 9
6 (2)	3, 4	3, 4	3, 4
7 (3)	5, 8, 9	8, 5, 9	8, 5, 9
8 (2)	1, 7	7, 1	1, 7
9 (2)	5, 7	5, 7	5, 7
10 (1)	4	4	4

Figure 6.2: High-level process remapping strategy, using local rearrangement of process subgraphs and global degree-based reordering.

My high-level process remapping strategy is shown in Fig. 6.2, which depicts the transformation of an input graph to a (weighted) process graph (consisting of 11 nodes/processes), subjected to local rearrangement options (regular, weighted or common neighborhood) and global reordering based degrees of the nodes in process graph. The overall scheme is listed in Algorithm 2; this approach is scalable, because each MPI rank traverses its local process subgraph (only the processes it communicates with) during

neighborhood rearrangement (Line 14), but common neighborhood rearrangement requires an extra communication round between process neighbors to gather the neighborhood sets, in addition to another communication round to gather the individual neighborhood lists on the root process (communication overhead is about $\mathcal{O}(log(p))$, where p is the number of processes). During the serial global reordering phase (Line 17), the list of processes are traversed according to specified order of degrees, prior to writing out the rank orders on a file.

Algorithm 2 Generate custom process order.
Input: $G_i = (V_i, E_i)$, (undirected) graph G_i in rank i, *root* rank, local rearrangement options: "default", "weighted" or "common", global traversal order T.
Output: R, reordered process list.

1: $R \leftarrow 0$ {local process neighbors}
2: $count \leftarrow \{0\}$ {optional, size of #processes}
3: **for** $v \in V_i$ **do**
4: **for** $u \in$ adj(v) **do**
{Neighbors of v} **if** $owner(u) \neq i$ **then**
6: $if\ owner(u) \in R \Rightarrow R \leftarrow R \cup owner(u)$
7: $count[owner(u)]$ += 1 {optional}
8: **end if**
9: **end for**
10: **end for**
11: **if** local rearrangement is "common" **then**
12: $R \leftarrow MPI_Neighbor_Gatherv(R)$ {among process neighbors}
13: **end if**
14: $R \leftarrow$ LOCALREARRANGEMENT$(count)$ {Local Rearrangement}
15: $R \leftarrow MPI_Gatherv(R, root = 0)$
16: **if** $rank == root$ **then**
17: $R \leftarrow$ TRAVERSALORDER(R, T) {Global Reordering}
18: Remove duplicates in R and write to file
19: **end if**=0

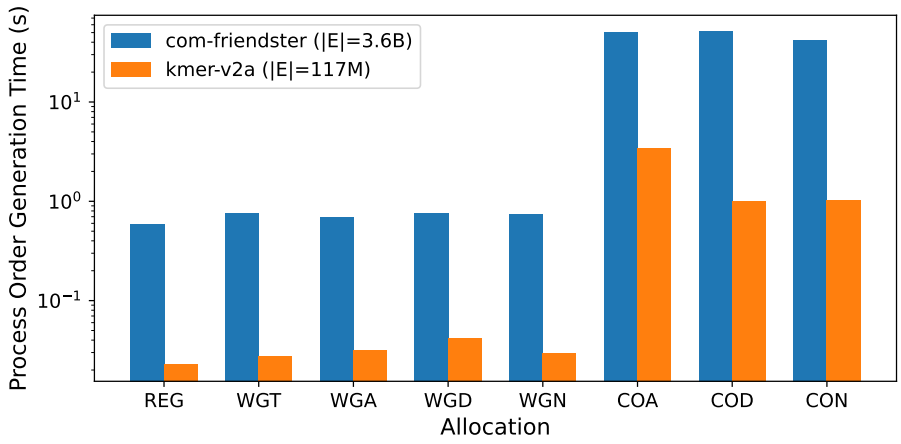


Figure 6.3: Process Remapping generation time (secs.) between relatively small and large inputs.

The "common neighborhood" rearrangement is determined by counting the number of intersections in pairwise sets (relatively expensive, owing to quadratic computational overheads), where a set is a process neighborhood. For e.g., consider process #3 (degree: 5), with a neighborhood comprising of $\{0,1,2,4,6\}$; to calculate "common neighborhood" reordering, we will first count intersecting sets of $\{0,1,2,4,6\}$ against $\{1,2,3\}$ (neighbors of process #0), $\{0,2,3,8\}$ (neighbors of process #1), $\{0,1,3,5\}$ (neighbors of process #2), $\{3,6,10\}$ (neighbors of process #4) and $\{3,4\}$ (neighbors of process #5). Then, we will sort (in a certain order) the original set $\{0,1,2,4,6\}$ as per the intersection counts.

I consider eight process remapping variants (excluding a few combinations), denoted by three-letter tokens—regular (`REG`), weighted (`WGT`), weighted ascending (`WGA`), weighted descending (`WGD`), weighted normal (`WGN`), common neighborhood ascending (`COA`), common neighborhood descending (`COD`) and common neighborhood normal (`CON`). Fig. 6.3 shows the one-time process order/remapping generation costs; common neighborhood order is expensive by an order of magnitude, which could be mitigated

using better (set intersection) algorithms and parallelism. I generate a custom process order file comprising of comma-separated ranks listed as per their order (following the format of `MPICH_RANK_ORDER` file that is understood by Cray MPICH, my platform MPI implementation), removing duplicates as neighborhoods are processed in certain order (as shown in Line 18).

6.2 Simulation and Empirical analysis

I discuss my experimental evaluations and analysis in this section, using graphs from the SuiteSparse collection [1] as my input. Graphs are distributed among processes, the original graph structure varies with the current process/node configurations (#partitions increases with #processes) and so does the resulting process graphs. For simulation analysis, I use 512 and 1024 processes (using Processes-Per-Node or PPN=1, 16 and 32 against a set of graphs), and for empirical analysis, I use 1024 processes (using PPN = 16, 32). The input graphs properties on 512 and 1024 processes are listed in Tables 6.1 and 6.2, demonstrating varied process graph characteristics, as implied by the range of average and Standard Deviation (SD) of degrees (for original and process graphs). For simulation, I additionally consider *random process placement* (denoted as `RND`), which may not be feasible on real systems, but can indicate further performance improvement opportunities via remapping.

6.2.1 Simulations results on Dragonfly topology with PPN = 16 and 32

I evaluated my approach using a Dragonfly-based topology to validate simulation results against the HPE Slingshot interconnect (Dragonfly-based) [8] on NERSC Perlmutter. Current discussion consider configurations with PPN = 16 and 32, using 512 processes (empirical evaluations also used PPN=16 and 32, on 1024 processes). Due to the simulation resource limitations, I select three graphs, listed in Table 6.2. To mitigate

Graphs	Graph Properties					Process Graphs (|V|=1024)			
	|V|	|E|	Max Deg.	Avg. Deg.	SD	|E|	Max Deg.	Avg. Deg.	SD
orkut	3M	234M	245K	229K	633	1M	1K	977	75
arabic	23M	640M	628K	625K	3.9K	230K	695	224	182
indoch.	7M	194M	196K	190K	14.6K	141K	432	138	105
logn18	262K	21M	70K	21K	5.8K	930K	964	908	224
stokes	12M	349M	342K	341K	1.3K	30K	144	30	20
uk2002	19M	298M	293K	291K	985	523K	944	510	216
kmer-v2a	55M	117M	115K	115K	10	95K	825	93	104
friendster	65M	3.6B	3.5M	3.5M	6.4K	1M	1K	1K	0

Table 6.1: Properties of input graphs and corresponding process graphs used in empirical studies on 1024 processes (PPN=16, 32).

Graphs	Graph Properties			Process Graph Properties (|V|=512)			
	Max Deg.	Avg. Deg.	SD	|E|	Max Deg.	Avg. Deg.	SD
orkut	490K	458K	6.3K	257K	511	502	26
uk2002	583K	582K	735	220K	503	430	92
logn18	65K	41K	7.6K	246K	496	481	84

Table 6.2: Properties of input graphs and corresponding process graphs used in simulations on 512 processes (PPN=16, 32).

the simulation time (simulation took about 2–30 hours for an application scenario), the experiments were conducted on 512 processes (instead of 1024), with application adaptations such as considering only 5 source-vertices for BFS and SSSP (for empirical evaluations, we use 64). Figures 6.4 and 6.5 illustrates the performance improvements across different input graphs in terms of estimated total time (compute + inactive + communication) for four distinct applications with PPN = 16 and PPN = 32 respectively. For PPN = 16, proposed process assignments improves upon the default ordering by up to

27% in BFS, 59% in SSSP, 29% in clustering, and 34% in Triangle Counting across various input graphs. For PPN = 32, our best assignment shows improvements over the default by up to 34% in BFS, 50% in SSSP, 31% in clustering, and 36% in Triangle Counting across different inputs. The greatest improvements among the input graphs was observed with “logn18” (>50%), and across applications, the most significant enhancement was achieved in SSSP (about 60%). Every process assignment method achieved substantial improvements over the default for nearly all combinations of the applications and input graphs. The only exception is the input graph “uk-2002”, where improvements were observed only in Clustering, while BFS showed no improvement and SSSP and Triangle Counting exhibited performance degradation for both PPN = 16 and 32 cases. These results may be attributed to the specific structure and properties of both the input graph and its corresponding process graph; “uk-2002” has the lowest Standard Deviation (SD) of 735 compared to the other input graphs (Table 6.2).

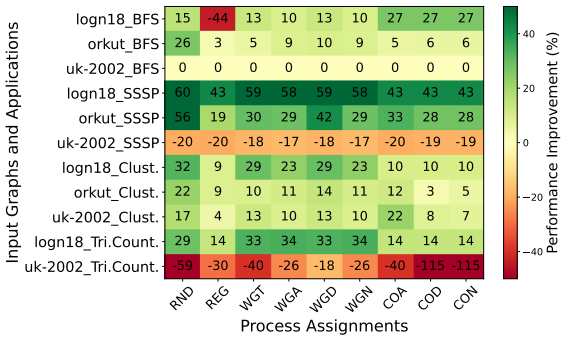


Figure 6.4: Simulation Results with 512 MPI processes, PPN = 16

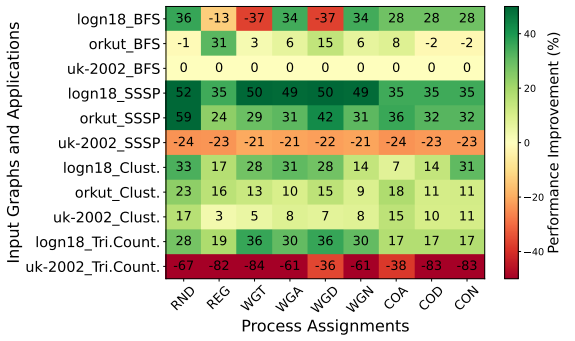


Figure 6.5: Simulation Results with 512 MPI processes, PPN = 32

6.2.1.1 Packet stalls and hops statistics To assess the reasons behind the performance improvements of my proposed assignment methods over the default assignment, I collected statistics on packet-level network hops and stalls for the PPN = 16 and 32 cases. If the local rearrangement phase (refer §6.1.2) enhances the locality, a higher percentage of the network packets should communicate locally, resulting in higher intranode communications, and fewer network hops. Conversely, if the global reordering phase of the process assignment can enhance the load balancing by reducing the network contention, stall queue lengths for the network packets should be lower.

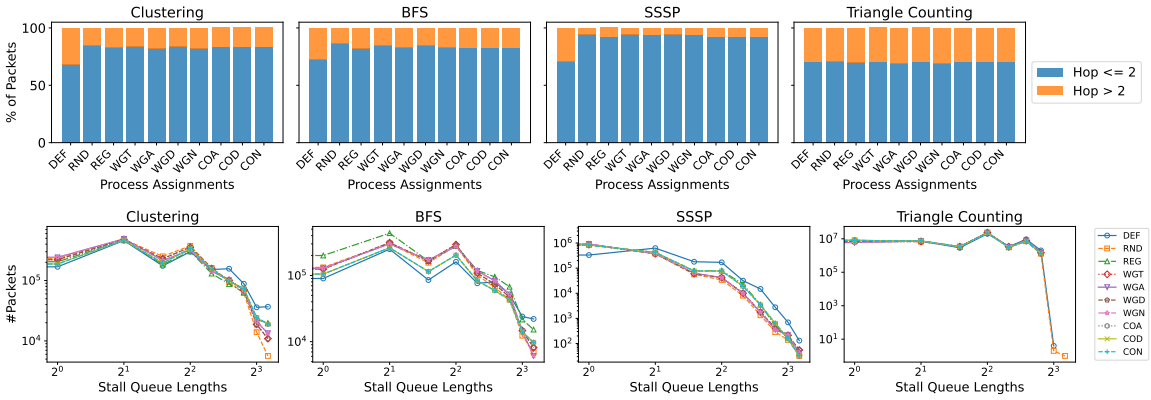


Figure 6.6: Packet-level network hops and stalls analysis of logn18 for PPN = 16; *lower* > 2 hops and stall queue lengths is *better*.

Fig. 6.6 presents the network hops and stalls statistics for the selected input graph, "logn18" with PPN=16 across the four applications: Clustering, BFS, SSSP, and Triangle Counting. The first sub-figure shows the percentage of packets traversing <=2 network hops and >2 network hops. With default assignment, 32%, 27%, 29%, and 30% of total network packets traverse more than 2 network hops in Clustering, BFS, SSSP, and

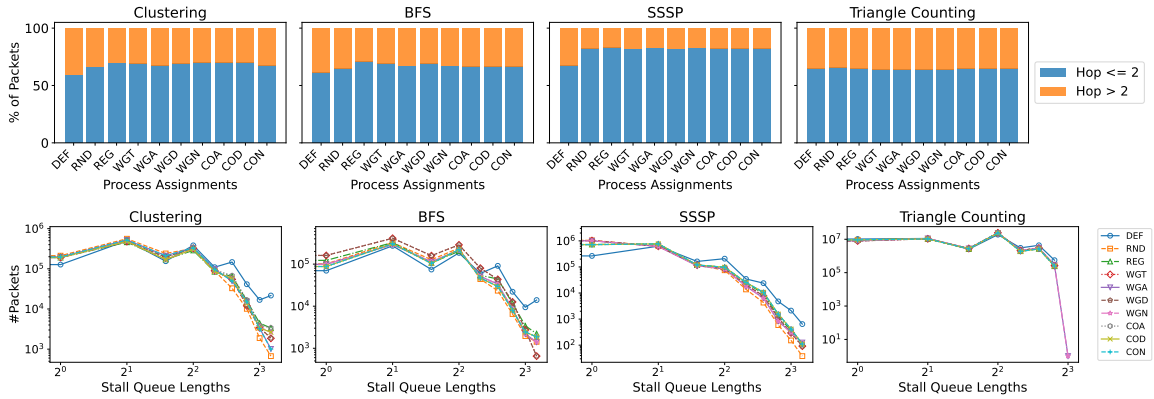


Figure 6.7: Packet-level network hops and stalls analysis of logn18 for PPN = 32; *lower* $>$ 2 hops and stall queue lengths is *better*.

Triangle Counting applications, respectively. In contrast, my proposed approaches result in 15–17%, 13–18%, 5–8%, and 28–30% of the network packets traversing more than 2 network hops in the same applications. My approaches significantly reduce the network hops in Clustering, BFS, and SSSP, while in Triangle Counting, improvements are mild. The second sub-figure displays the distribution of the stall queue lengths across network packets. It is evident that with the default process assignment, fewer packets experience lower stalls, while a larger proportion (represented in the tail part of the distribution) encounter higher stalls as compared to our proposed approaches in Clustering, BFS, and SSSP. In Triangle Counting, the distribution of the stall queue lengths is similar for the proposed approaches. Fig. 6.7 depicts the network hops and stalls statistics for "logn18" with PPN = 32. Similar improvement patterns are observed for PPN = 32 as well. With the default assignment, 41%, 38%, 33%, and 35% of the network packets traverse more than two hops, whereas with my proposed approaches, 30–34%, 31–35%, 17–18%, and 34–36% of network packets traverse more than 2 network hops in the respective applications. Regarding the distribution of stall queue lengths for network packets,

similarly, with the default assignment, more packets experience higher stall queue lengths compared to proposed approaches.

6.2.2 Analysis of empirical results from NERSC Perlmutter

Experiments discussed in this section are performed on the NERSC Perlmutter supercomputer consisting of 3,072 CPU-only compute nodes. Each node uses 64-core 2.4GHz AMD EPYC 7763 CPUs with 256GB of DDR4 memory, 256MB L3 cache and 8 memory channels with HPE Slingshot 11 interconnect, providing up to 200Gbps (25GB/s) bandwidth [45].

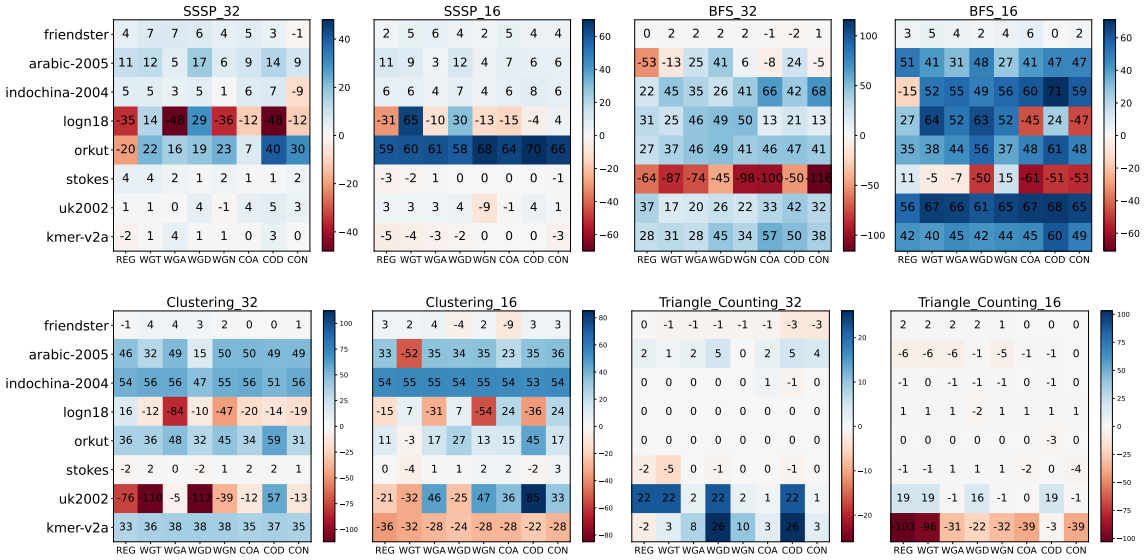


Figure 6.8: Application % performance improvements on NERSC Perlmutter with PPN = 32 and 16 (1024 processes) for various inputs graphs.

6.2.2.1 Baseline performance comparisons for process remappings Fig. 6.8 demonstrates the performance improvements achieved by our proposed assignment methods compared to the default assignment across SSSP, BFS, Clustering, and Triangle Counting applications using 8 input graphs, with PPN = 16 and 32 on 64/32 nodes (1024 processes, see input graph properties in Table 6.1). For SSSP, best approaches achieve improvements ranging from 5% to 40% with PPN = 32 and from 4% to 70% with PPN = 16 for most of the input graphs. However, there are exceptions: "stokes" and "kmer-v2a" does not show noticeable improvements. In BFS, process remappings achieved improvements of 41% to 68% and 48% to 71% with PPN = 32 and 16, respectively, across most input graphs. Exceptions include "friendster" (largest graph in our set, with over 3 billion edges), where improvements were only 2% to 6%, and 'stokes', which often showed negative improvements. The process graph of "friendster" indicate dense connections (SD = 0, see Table 6.1, in contrary, "stokes" has low SD and least maximum degree, indicating major sparsity), proposed process remappings can lead to more improvements with higher #nodes (such that the process graph is relatively sparser). For Clustering, my best approaches resulted in improvements of 16% to 59% and 24% to 85% across most applications. However, "friendster" and "stokes" showed marginal improvements, and with "kmer-v2a" on PPN = 16, I observed performance degradation. For Triangle Counting, we generally observe minimal or no improvements, this could be attributed to adversarial medium-large message exchanges, an outlier among graph workloads. Notably, "kmer-v2a" showed significant improvements with PPN = 32 in Clustering and Triangle Counting, but with PPN = 16, I observe entirely opposite behavior. Since the ratio of max to average degree of the process graph for "kmer-v2a" is close to 10, performance of certain graph workloads (with an abundance of medium-sized messages, like Clustering and Triangle Counting) can be susceptible to the volume of intranode to internode data transfers (controlled using PPN).

6.2.2.2 Network contention analysis on NERSC Perlmutter Fig. 6.9 demonstrates the statistics gathered from counters integrated into the HPE Cassini Network Interface Card (NIC); every Perlmutter CPU node has a single NIC, so these counters provide detailed packet/flit-level telemetry. The plot illustrates the distribution of request/response stalls per node via NIC counters: HNI_TX_PAUSED_<n> and HNI_RX_PAUSED_<n> [46]. The former counts cycles where the transmit path is paused for traffic class <n>, while the latter counts cycles where pause is applied on the receive path for the traffic class <n>. The default class is 0 for requests and 1 for responses (not all workloads record values for both counters).

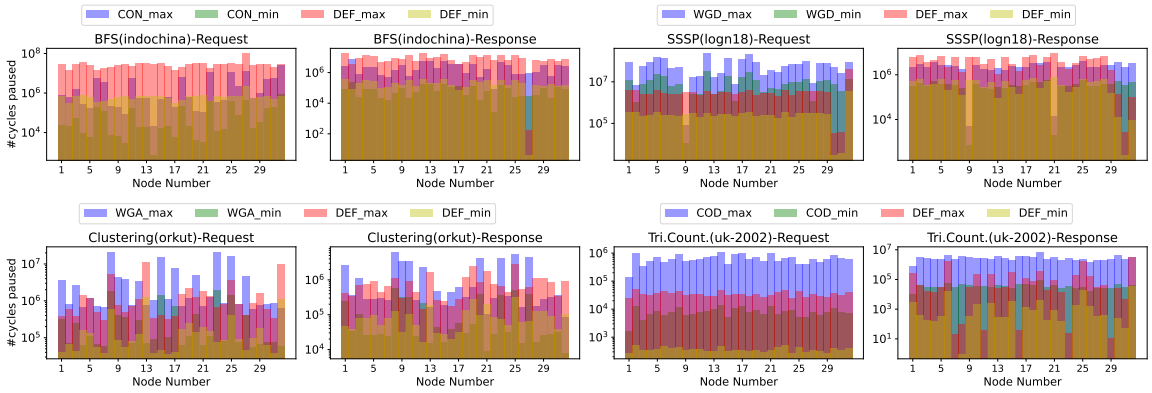


Figure 6.9: Distribution of request/response paused cycles corresponding to Slingshot NIC counters across 32 nodes of NERSC Perlmutter.

I selected process assignments from 4 input graphs covering BFS, SSSP, Clustering, and Triangle Counting. My approach is locality-based, and our global reordering aims to achieve load balancing based on the node degrees (i.e., reordering many-to-many connections), which is directly related to the network contention. In BFS

using the “indochina” graph, my approach shows a lower number of cycles paused than the default approach for both request and response traffic classes. For SSSP using “logn18”, proposed remappings lead to slightly higher paused cycles in the request traffic class but a lower number in the response traffic class. In Clustering using ’orkut’, some nodes exhibit higher cycle pauses with the default mapping, while others show higher pauses with proposed remappings, in both request and response traffic classes. For Triangle Counting with “uk-2002”, proposed remappings exhibit higher cycles paused than the default process assignment in request class, but offer some improvements for the response class.

6.2.3 Discussion

In this research, I propose heuristic-based process assignments for distributed-memory graph workloads. My approach involves generating process orders using heuristics, which is a one-time effort and independent of applications, process per nodes (PPN), and the network topology. Through packet-level simulation and empirical results from the NERSC Perlmutter supercomputer, I demonstrate performance improvements ranging from 30% to 40% across most of the input graph and complex application combinations using my best approaches. Additionally, I analyze low-level metrics such as hops, stall queue lengths, and network counters to explain the reasons behind these performance gains with my assignment strategies. For future work, I suggest integrating topology-level information into my approach to enable secondary reordering for specific network topologies.

CHAPTER SEVEN

CONCLUSION

Recent advancements in training machine learning and deep learning models with large-scale datasets, including large language models and deep learning recommendation models, have significantly increased the demand for high-performance computing (HPC) and data center infrastructures. As data centers expand to meet this demand, performance challenges have emerged, particularly concerning interconnect networks. In this work, I propose novel routing algorithms, optimized multipath selection methods, and improved process scheduling techniques aimed at enhancing the performance of distributed memory applications across various state-of-the-art interconnect architectures. Additionally, I conducted a comparative performance analysis of different interconnect architectures in various application scenarios, employing multiple routing algorithms. The algorithms and optimization methods I developed aim to mitigate performance bottlenecks in future large-scale HPC and data center networks. By doing so, they will help decrease application latency and improve system throughput, ultimately contributing to more efficient and effective computing solutions.

REFERENCES

[1] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu, and R. Sandstrom, "The suitesparse matrix collection website interface," *Journal of Open Source Software*, vol. 4, no. 35, p. 1244, 2019.

[2] O. R. L. C. Facility, "Frontier," 2024. Accessed: 2024-10-09.

[3] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: an open source software for exploring and manipulating networks," in *Proceedings of the international AAAI conference on web and social media*, vol. 3, pp. 361–362, 2009.

[4] "Frontier." `https://www.olcf.ornl.gov/frontier/`. [Online; accessed 18-March-2024].

[5] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM computer communication review*, vol. 38, no. 4, pp. 63–74, 2008.

[6] "Summit at oak ridge national laboratory." `https://www.olcf.ornl.gov/summit/`, 2023.

[7] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3, pp. 77–88, 2008.

[8] D. De Sensi, S. Di Girolamo, K. H. McMahon, D. Roweth, and T. Hoefler, "An in-depth analysis of the slingshot interconnect," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, IEEE, 2020.

[9] E. S. J. D. Meuer, Hans and H. Simon, "Top 500 list." `https://www.top500.org/lists/top500/2023/11/`.

[10] A. Shpiner, Z. Haramaty, S. Eliad, V. Zdornov, B. Gafni, and E. Zahavi, "Dragonfly+: Low cost topology for scaling datacenters," in *2017 IEEE 3rd International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*, pp. 1–8, IEEE, 2017.

[11] M. Flajslik, E. Borch, and M. A. Parker, "Megafly: A topology for exascale systems," in *High Performance Computing: 33rd International Conference, ISC High Performance 2018, Frankfurt, Germany, June 24-28, 2018, Proceedings 33*, pp. 289–310, Springer, 2018.

[12] "LUMI." `https://leonardo-supercomputer.cineca.eu/`. [Online; accessed 31-March-2024].

[13] "Dragonfly+ in canada's supercomputer niagara." https://www.scinethpc.ca/niagara/, mar 2018.

[14] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking Data Centers Randomly," in *NSDI*, vol. 12, pp. 17–17, 2012.

[15] B. Bollobás and W. Fernandez de la Vega, "The diameter of random regular graphs," *Combinatorica*, vol. 2, no. 2, pp. 125–134, 1982.

[16] B. Bollobás, "The isoperimetric number of random regular graphs," *Eur. J. Comb.*, vol. 9, pp. 241–244, May 1988.

[17] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE transactions on Computers*, vol. 100, no. 10, pp. 892–901, 1985.

[18] S. R. Öhring, M. Ibel, S. K. Das, and M. J. Kumar, "On generalized fat trees," in *Parallel Processing Symposium, 1995. Proceedings., 9th International*, pp. 37–44, IEEE, 1995.

[19] X.-Y. Lin, Y.-C. Chung, and T.-Y. Huang, "A multiple lid routing scheme for fat-tree-based infiniband networks," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, p. 11, IEEE, 2004.

[20] J. Kim, W. J. Dally, and D. Abts, "Adaptive routing in high-radix clos network," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pp. 92–es, 2006.

[21] L. G. Valiant, "A scheme for fast parallel communication," *SIAM journal on computing*, vol. 11, no. 2, pp. 350–361, 1982.

[22] M. García, E. Vallejo, R. Beivide, M. Odriozola, and M. Valero, "Efficient routing mechanisms for dragonfly networks," in *2013 42nd International Conference on Parallel Processing*, pp. 582–592, IEEE, 2013.

[23] N. Jiang, J. Kim, and W. J. Dally, "Indirect adaptive routing on large scale interconnection networks," in *Proceedings of the 36th annual international symposium on Computer architecture*, pp. 220–231, 2009.

[24] N. Jiang, J. Kim, and W. J. Dally, "Indirect adaptive routing on large scale interconnection networks," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 220–231, ACM, 2009.

[25] D. Eppstein, "Finding the k shortest paths," *SIAM Journal on computing*, vol. 28, no. 2, pp. 652–673, 1998.

[26] A. Singh, *Load-Balanced Routing In Interconnection Networks*. PhD thesis, Stanford University, 2005.

[27] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, (Washington, DC, USA), pp. 77–88, IEEE Computer Society, 2008.

[28] Swiss National Supercomputing Centre, "Piz Daint| CSCS." `https://www.cscs.ch/computers/piz-daint/`.

[29] B. J. Archer and M. Vigil, "The Trinity system," in *Nuclear Explosive Code Development Conference (NECDC)*, (Los Alamos, New Mexico), Oct. 20–24, 2014. Also appears as Los Alamos Technical Report LA-UR-15-20221.

[30] M. Ponce, R. van Zon, S. Northrup, D. Gruner, J. Chen, F. Ertinaz, A. Fedoseev, L. Groer, F. Mao, B. C. Mundim, M. Nolta, J. Pinto, M. Saldarriaga, V. Slavnic, E. Spence, C.-H. Yu, and W. R. Peltier, "Deploying a top-100 supercomputer for large parallel workloads: The niagara supercomputer," in *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, PEARC '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[31] H. Adalsteinsson, S. Cranford, D. A. Evensky, J. P. Kenny, J. Mayo, A. Pinar, and C. L. Janssen, "A simulator for large-scale parallel computer architectures," *Int. J. Distrib. Syst. Technol.*, vol. 1, pp. 57–73, Apr. 2010.

[32] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The nas parallel benchmarks," tech. rep., The International Journal of Supercomputer Applications, 1991.

[33] B. Bollobás and F. R. K. Chung, "The diameter of a cycle plus a random matching," *SIAM Journal on Discrete Mathematics*, vol. 1, no. 3, pp. 328–333, 1988.

[34] X. Yuan, S. Mahapatra, W. Nienaber, S. Pakin, and M. Lang, "A new routing scheme for jellyfish and its performance with hpc workloads," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), Association for Computing Machinery, 2013.

[35] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner, "Maximizing modularity is hard," *arXiv preprint physics/0608255*, 2006.

[36] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.

[37] G. Shainer, E. Gutkind, B. Lee, M. Kagan, and Y. Kliteynik, "Optics for enabling future hpc systems," in *2009 17th IEEE Symposium on High Performance Interconnects*, pp. 138–142, IEEE, 2009.

[38] M. Taubenblatt, P. Maniotis, and A. Tantawi, "Optics enabled networks and architectures for data center cost and power efficiency," *Journal of Optical Communications and Networking*, vol. 14, no. 1, pp. A41–A49, 2022.

[39] B. Stephens, A. Akella, and M. M. Swift, "Your programmable nic should be a programmable switch," in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pp. 36–42, 2018.

[40] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[41] A. S. C. Initiative *et al.*, "The asci sweep3d benchmark code," 1995.

[42] H. Lu, M. Halappanavar, and A. Kalyanaraman, "Parallel heuristics for scalable community detection," *Parallel Computing*, vol. 47, pp. 19–37, 2015.

[43] D. S. Johnson, "Fast algorithms for bin packing," *Journal of Computer and System Sciences*, vol. 8, no. 3, pp. 272–314, 1974.

[44] S. Ghosh, M. Halappanavar, A. Tumeo, and A. Kalyanarainan, "Scaling and quality of modularity optimization methods for graph clustering," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, IEEE, 2019.

[45] C. Yang and J. Deslippe, "Accelerate science on perlmutter with nersc," *Bulletin of the American Physical Society*, vol. 65, 2020.

[46] L. DeRose, B. Homer, D. Johnson, S. Kaufmann, and H. Poxon, "Cray performance analysis tools," in *Tools for High Performance Computing: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart*, pp. 191–199, Springer, 2008.