

© 2016 by Nikhil Jain. All rights reserved.

OPTIMIZATION OF COMMUNICATION INTENSIVE APPLICATIONS
ON HPC NETWORKS

BY

NIKHIL JAIN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kalé, Chair
Professor William D. Gropp
Professor Josep Torrellas
Professor D. K. Panda, The Ohio State University

Abstract

Communication is a necessary but overhead inducing component of parallel programming. Its impact on application design and performance is due to several related aspects of a parallel job execution: network topology, routing protocol, suitability of algorithm being used to the network, job placement, etc. This thesis is aimed at developing an understanding of how communication plays out on networks of high performance computing systems and exploring methods that can be used to improve communication performance of large scale applications.

Broadly speaking, three topics have been studied in detail in this thesis. The first of these topics is task mapping and job placement on practical installations of torus and dragonfly networks. Next, use of supervised learning algorithms for conducting diagnostic studies of how communication evolves on networks is explored. Finally, efficacy of packet-level simulations for prediction-based studies of communication performance on different networks using different network parameters is analyzed.

The primary contribution of this thesis is development of scalable diagnostic and prediction methods that can assist in the process of network designing, adapting applications to future systems, and optimizing execution of applications on existing systems. These methods include a supervised learning approach, a functional modeling tool (called *Damselfly*), and a PDES-based packet level simulator (called *TraceR*), all of which are described in this thesis.

Acknowledgments

First and foremost, I would like to thank my labmates and colleagues without whom it would have been impossible to keep my sanity while working for all these years. Xiang and Jonathan have played an incredibly important role in my life in last four years. All my co-authors, Bilge, Ehsan, Eric B, Eric M, Harshitha, Michael, and Ronak have helped me in more ways than I can count. Words of wisdom from the more experienced Esteban, Osman, Phil, Gengbin, Celso, Lukasz, and Abhishek have guided me well through uncertain times. Interaction with all other PPLers, Akhil, Chao, Pritish, Ram, Sam, and Yanhua has left me with a positive frame of mind more often than not.

Next, I have no words to express my gratitude towards my mentors. Prof. Kale is an amazing person, probably the most kind and positive person I have met. This thesis, and my impending research career, would not exist if not for his supervision and guidance. Abhinav's role in my growth as a researcher and in completion of this thesis is critical. He has spent countless hours discussing and developing ideas that are at the heart of this thesis. Todd has always been approachable and provided extremely valuable suggestions/feedback. Finally, I thank Yogish Sabharwal and Manish Gupta - my mentors at IBM Research India - if not for their nurturing and guidance, I may have never pursued higher studies in the US.

I would like to thank the members of my thesis committee - Prof Gropp, Prof Panda, and Prof Torrellas - for their time and feedback. I would also like to thank many other researchers who have helped me in one way or another during my thesis - Jim Phillips, Glenn Martyna, Sohrab Ismail-Beigi, Minjung Kim, Subashish Mandal, Fabrizio Petrini, Nicholas J. Wright, Kalyan Kumaran, Francesco Miniati, Mark F. Adams, Timo Bremer, Jayaraman Thiagarajan, Yarden Livnat, Misbah Mubarak, Chris Carothers, Jae-Seung Yeom, Andrew Titus, and Steven H. Langer.

Last, but not the least, I am forever indebted to my family in India for their unconditional love and encouragement.

Grants

This work was partially supported by many funding sources and made use of resources from several supercomputing centers and projection allocations. I would like to thank all of the following sources that contributed towards completion of this work:

Fellowship sources: Department of Computer Science, UIUC (Andrew and Shana Laursen Fellowship, 2011-2012); IBM (IBM Ph.D. Fellowship, 2014-2015).

Assistantship sources: NSF Award 1339715; NSF Award OCI 07-25070; NSF Award OCI-0832673; Project tracking code 13-ERD-055 under US DOE Contract DE-AC52-07NA27344.

Supercomputing centers: Argonne Leadership Computing Facility, Argonne National Laboratory; CSCS, Swiss National Supercomputing Centre, ETH Zurich; Livermore Computing Center, Lawrence Livermore National Laboratory; National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign; National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory; Oak Ridge Leadership Computing Facility, Oak Ridge National Laboratory; Pittsburg Supercomputing Center, Carnegie Mellon University and University of Pittsburgh; Texas Advanced Computing Center, University of Texas at Austin.

Machine allocations: PEACEndStation, PARTS, HPC.PAMS, CharmRTS (ALCF, ANL systems); PEACEndStation, ALCC (ORNL systems); DD_jtg, ILL_jrc, namd, PRAC_jnk (Blue Waters, NCSA); XSEDE (PSC and TACC systems).

Table of Contents

List of Figures	vii
List of Tables	xiii
CHAPTER 1 Overview	1
1.1 Thesis organization	3
CHAPTER 2 Background and Related Work	5
2.1 Topologies in HPC networks	5
2.2 Interaction patterns	7
2.3 Task mapping and job placement	9
2.4 Indicators of performance	11
2.5 Offline performance prediction	13
2.6 Communication algorithms	13
CHAPTER 3 Job Placement and Task Mapping	15
3.1 Task mapping on torus	16
3.2 Job placement on the dragonfly network	33
CHAPTER 4 Causes of Network Congestion	55
4.1 Contention on torus networks	57
4.2 Experimental setup	60
4.3 Performance prediction of communication kernels	67
4.4 GBRT and production applications	73
4.5 Identifying relevant feature subsets	75
4.6 Summary	80
CHAPTER 5 TraceR: PDES Simulator	81
5.1 Background	82
5.2 Design and implementation of TRACER	83
5.3 Network models in CODES	86
5.4 Simulation configuration	91
5.5 Impact of simulation configuration	93

5.6	Performance comparison	97
5.7	Summary	99
CHAPTER 6 Comparison of Networks		100
6.1	Network prototypes	100
6.2	Communication performance comparison	104
6.3	Network cost comparison	111
6.4	Performance Per Dollar	116
6.5	Summary	119
CHAPTER 7 Impact of Configuration on Performance		121
7.1	Stencil with unbounded resources	121
7.2	Stencil with practical resources	125
7.3	Spread with unbounded resources	129
7.4	Spread with practical resources	132
7.5	Summary and discussion	136
CHAPTER 8 Communication Algorithms		137
8.1	Analysis of collectives on dragonfly networks	137
8.2	Charm-FFT	145
8.3	Summary	151
CHAPTER 9 Conclusion		152
REFERENCES		154

List of Figures

1.1	Motivating examples for research on communication optimization.	2
2.1	Example hypercubes of various dimensions: note the recursive construction of an n -dimensional hypercube using two $n - 1$ -dimensional hypercubes.	6
2.2	Variations of fat-tree/folded Clos topology.	6
2.3	Torus are constructed by wrapping the corresponding grids at all boundaries.	7
2.4	Dragonfly: A two-tier network with dense all-to-all connections among logical routers at each level.	8
2.5	Mapping of a 6×5 grid to a 10×3 mesh using different mapping strategies. (source [34])	10
2.6	Two stages of recursive bisection topology-adapted partition mapping. The dark lines represent the sorted node traversal order for bisection. Grid element color represents recursive bisection domains after each stage.	11
3.1	Mapping 2D sub-partitions to 3D shapes in Rubik.	17
3.2	Average, minimum, and maximum time spent in communication by pF3D for weak scaling.	21
3.3	Average time spent in different MPI routines by pF3D for weak scaling.	21
3.4	A Rubik script to generate tiled mappings for pF3D.	22
3.5	Reduction of time spent in different MPI routines by using various task mappings for pF3D running on 1,024, 2,048 and 4,096 nodes of Blue Gene/Q (Note: y-axis has a different range in each plot).	23
3.6	pF3D plots comparing the time spent in point-to-point operations to average and maximum hops (top) and comparing the MPI time to average and maximum load on network links (bottom) (Note: y-axis has a different range in each plot).	24
3.7	Average time spent by pF3D in different MPI routines on 4096 nodes (includes <code>MPI_Isend</code> optimization).	25
3.8	Evaluation of the baseline performance of MILC with the default mapping.	27
3.9	Reduction of time spent in different MPI routines by using various task mappings for MILC running on 1024, 2048 and 4096 nodes of BG/Q (Note: y-axis has a different range in each plot).	27

3.10	MILC plots comparing the time spent in point-to-point operations with average and maximum hops (top) and the MPI time with average and maximum load on network links (bottom) (Note: y-axis has a different range in each plot).	28
3.11	Four sub-tori showing the D (blue), C (red, long), and B (red, short, diagonal) links for the same E. The colors represent the number of packets passing through individual links.	30
3.12	Minimaps showing aggregated network traffic along various directions for the TABCDE (left) and the <i>Tile3</i> mappings (right).	31
3.13	pF3D: A scaling comparison of the time spent in different MPI routines with the default mapping (top-left), best mapping discovered, and with the best mapping using the Isend optimization. A scaling comparison of the benefits of task mapping for MILC is also shown (bottom-right). The percentage values shown are improvement over the default mapping.	32
3.14	The structure of a dragonfly network.	34
3.15	Comparison of the predictions by the presented model with predictions by SST/macro, a packet-level simulator, for a 4D Stencil simulated on a 36,864 router system.	41
3.16	Example to explain the data displayed in the plots.	42
3.17	Unstructured Mesh Pattern (UMesh): blocking helps in improving the traffic distribution.	43
3.18	Structured Grid Pattern (4D Stencil) and Random Neighbors Pattern (Spread).	45
3.19	4D Stencil: distribution of traffic on L2 links for RDG.	46
3.20	Many to many pattern (M2M): direct routing with randomized placement has lower average and maximum traffic.	47
3.21	Traffic distribution for M2M on 66% and 33% cores.	49
3.22	Parallel workloads traffic distribution.	51
3.23	Job-specific routing traffic distribution (All Links).	53
4.1	Performance variation with prior metrics for five-point halo exchange on 16,384 cores of Blue Gene/Q. Points represent observed performance with various task mappings. A large variation in performance is observed for the same value of the metric in all three cases.	56
4.2	Message flow on Blue Gene/Q - a task initiates a message send by putting a descriptor in one of its memory injection FIFOs; the messaging unit (MU) processes these descriptors and injects packets into the injection network FIFOs from which the packets leave the node via links. On intermediate switches, the next link is decided based on the destination and the routing protocol; if the forward path is blocked, the message is stored in buffers. Finally on reaching the destination, packets are placed in network reception FIFOs from where the MU copies them to either the application memory or memory reception FIFOs.	57
4.3	Example decision tree and random forests generated using scikit.	63

4.4	Parameterized loss functions for gradient tree boosting: Huber loss function with the cutting-edge parameter δ (left), quantile loss function (right).	65
4.5	Performance variations with different task mappings on 16,384 cores of BG/Q. As benchmarks become more communication intensive, even for small message sizes, mapping impacts performance.	67
4.6	Prediction success based on prior features on 16,384 cores of BG/Q. The best RCC score is 0.91 for most cases - 38 mispredictions out of 378.	68
4.7	Prediction success based on new features on 16,384 cores of BG/Q. We observe a general increase in RCC, but R^2 values are low in most cases resulting in empty columns.	69
4.8	Prediction success based on hybrid features from Table 4.5 on 16,384 cores of BG/Q. We obtain RCC and R^2 values exceeding 0.99 for 3D Halo and Sub A2A. Prediction success improves significantly for 2D Halo also.	71
4.9	Prediction success: summary for all benchmarks on 65,536 cores of BG/Q. Hybrid metrics show high correlation with application performance.	72
4.10	Summary of prediction results on 65,536 cores using 4 MB messages. For all benchmarks, prediction is highly accurate both in terms of ordering and absolute values.	73
4.11	Highest prediction scores obtained for the individual datasets using Extremely Randomized Trees (left) and Gradient Boosted Regression Tree (right). Adjoining pairs of vertical bars represent the RCC and R^2 values for each of the sixteen datasets.	73
4.12	Ranks of different features in the models that yield the highest RCC (left plot) and R^2 scores (right plot) for individual datasets using Gradient Tree Boosting (loss function = ‘Huber’). Each stacked bar represents the ranks of the nineteen features (colored by categories) for one of the sixteen datasets.	74
4.13	GBRT regression on the Apps dataset using different quantile loss functions. The lower quantile regression function underpredicts for those samples with high execution time, while predicting effectively for those with low execution times.	75
4.14	Ranks of different features obtained using GBRT with quantile loss functions at $\alpha = 0.1$ and $\alpha = 0.9$ respectively: left plot is for a combined set of the three communication kernels (twelve datasets) and the right plot is for a combined set of the two applications (four datasets).	76
4.15	Comparison of the feature ranks obtained using the feature selection technique applied to the eight larger datasets. Note that the marker colors for each row/dataset are scaled independently (red is high and blue is low).	78
4.16	Prediction performance of the features selected using the proposed quantile analysis on different datasets.	79
5.1	Integration of TRACER with BigSim emulation and CODES (left). Forward path control flow of trace-driven simulation (right).	84
5.2	Fat-tree construction using switches of same radix.	88
5.3	Optimistic vs. conservative DES	94

5.4	Effect of batch size and GVT interval on performance: 8K simulated nodes are simulated using 8 cores (top 2 plots), and 512K using 256 cores (bottom 2 plots).	95
5.5	Impact of #LPs per KP.	96
5.6	Sequential simulation time.	97
5.7	Scalability of TRACER when simulating networks of various sizes.	98
6.1	Communication performance of different networks for 4D Stencil. Torus outperforms dragonfly, which in turn performs better than fat-tree for large message sizes. For smallest message size, all networks show similar performance.	107
6.2	Communication performance of different networks for Near-Neighbor (NN). Irrespective of the message size, torus is faster by 2x in comparison to dragonfly and fat-tree.	108
6.3	Communication performance of different networks for Subset All-to-All (A2A). With careful mapping, the execution time is similar for all the networks, with dragonfly being marginally better and fat-tree being marginally worse than torus.	108
6.4	Communication performance of different networks for Perm. Fat-tree provides the best performance for all message sizes, followed by dragonfly which is better than torus by $\sim 25\%$	109
6.5	Communication performance of different networks for Spread. Fat-tree outperforms dragonfly by a small margin, while both of them are significantly faster than torus for all message sizes.	110
6.6	Cost model for copper and optical cables.	111
6.7	Cost model for routers.	113
6.8	Estimated router cost for building networks for prototype systems based on different interconnect topologies.	114
6.9	Estimated cable cost for building networks for prototype systems based on different interconnect topologies.	114
6.10	Comparison of estimated cost for building different networks for a given node count. Only router and cable cost are considered.	116
6.11	Although torus provides the best performance, its performance per dollar is worst among the three networks. As the message size increases, the superior performance of dragonfly leads to a better performance per dollar.	117
6.12	Fat-tree shows the best performance per dollar due to its low cost, although its performance is similar to dragonfly. Due to its superior performance, torus' performance per dollar is only 10% lower than the dragonfly.	117
6.13	Given the similar performance of all networks, performance per dollar is significantly impacted by the cost of the networks.	118
6.14	The performance difference among the three network is further enhanced by the cost difference. As a result, fat-tree show very high performance per dollar in comparison to the dragonfly, which in turn is much higher than the performance per dollar of torus.	119

6.15	Summary of the communication rate and performance per dollar for large message sizes. Plotted values are normalized using the values for the fat-tree.	120
7.1	(left) When all other resources are practically unlimited, the communication performance is directly proportional to the router delay/latency. (right) Size of router buffer has no effect on the performance of the dragonfly network. Its impact on the torus network can be significant, but is hard to model.	122
7.2	As the link or injection bandwidth is increased, the execution time drops linearly. While the dragonfly network saturates at the link bandwidth of 400 GBps, the torus network shows performance improvement till 1000 GBps. In contrast, the dragonfly network provides performance improvement till 1000 GBps when injection bandwidth is increased, but the torus network saturates at 400 GBps.	123
7.3	(left) Impact of changing both link and injection bandwidth on execution time. For both the networks, significant improvement in performance are observed; for dragonfly, the relative improvement reduces as the bandwidths are increased to very large values. (right) Impact of routing policy and injection policy. When other configuration parameters are not the bottleneck, both policies impact the observed performance on torus; in contrast, on a dragonfly, good choice of one makes the other irrelevant.	124
7.4	(left) For torus, link bandwidth acts as a primary bottleneck, with injection bandwidth requirement saturating at large values. (right) For dragonfly, both link and injection bandwidth are critical, with link bandwidth being more important at lower bandwidth and injection bandwidth being the bottleneck at larger values.	125
7.5	(left) For torus, the impact of routing is minimal for FCFS injection policy, when resources are limited. Adaptive routing improve performance for dragonfly, especially for large bandwidth. (right) With RR injection policy, Adaptive routing provides significant performance improvement for both torus and dragonfly networks.	125
7.6	(left) On torus, using RR injection policy is highly beneficial when Adaptive routing is used. (right) When the link and the injection bandwidths are the primary bottlenecks, RR provide significant performance improvement on a dragonfly.	126
7.7	As the router delay increases, the bottleneck changes from link and injection bandwidth to the fixed delays. Conversely, for a fixed router delay, as the link/injection bandwidth decreases, the fixed delays cease to be the performance bottleneck.	127
7.8	Effect of variations in the link bandwidth, injection bandwidth, and latency on execution time of 4D Stencil on torus. Increasing link bandwidth reduces the execution time, but if injection bandwidth is much lower, it limits the performance. When both link and injection bandwidth are high, very high latency can be the performance bottleneck.	128

7.9	Unlike torus, an increasing injection bandwidth impacts the execution time both positively and negatively, even when link bandwidth is low. Similarly, the router latency has impact on the performance even when the link and injection bandwidth are relatively low.	129
7.10	(left) The execution time increases as the router latency is increased; performance of the torus is similar to the dragonfly. (right) As for 4D Stencil, size of router buffer has no effect on the performance of the dragonfly network. Its impact on the torus network is significant, but does not follow a pattern.	130
7.11	The execution drops almost linearly as the link or injection bandwidth is increased. The dragonfly network saturates at link bandwidth of 400 GBps, but the torus network observes good performance improvement till 1000 GBps. The reverse is true for injection bandwidth.	131
7.12	(left) For both torus and dragonfly, significant improvement in performance is observed, even for very large bandwidth. (right) Impact of routing policy and the injection policy: RR + Adaptive provides the best performance on torus, while only FCFS + Static performs badly on dragonfly.	131
7.13	(left) Adaptive routing provides significant improvement for torus, but has minimal impact on the dragonfly network . (right) With RR injection policy, Adaptive routing provides significant performance improvement for both torus and dragonfly networks.	132
7.14	(left) The RR injection policy provides better performance when Adaptive routing is used, but FCFS shows similar performance for Static routing. (right) On the dragonfly network, performance improvements are high for Adaptive routing and for Static routing if the bandwidth is high.	133
7.15	For low bandwidth, the router delay does not impact the execution time, but as the router delay increases, it becomes the performance bottleneck. . .	134
7.16	Performance improves with increasing link bandwidth, but saturates quickly when injection bandwidth is increased. Impact of latency is less prominent in comparison to 4D Stencil.	135
7.17	An increasing injection bandwidth impacts the execution time positively, even beyond the link bandwidth value. The impact of the router latency is prominent and proportional to the delay.	135
8.1	Phase 1 and 2 of Charm-FFT.	147
8.2	Phase 3 and 4 of Charm-FFT.	148
8.3	Performance of Charm-FFT.	150

List of Tables

3.1	Shape and connectivity of the partitions allocated on Vulcan (Blue Gene/Q) for different node counts.	20
3.2	Tile sizes used for the Blue Gene/Q 5D torus and pF3D in different mappings.	23
3.3	Details of communication patterns.	40
3.4	Percentage cores allocated to patterns in workloads.	51
4.1	*Prior and †new metrics that indicate contention for network resources. . . .	61
4.2	List of communication metrics (features) used as inputs to the machine learning model. The colors in this table correspond to different hardware components in Table 4.1	61
4.3	Dimensions of the allocated job partitions on BG/Q.	61
4.4	Sizes of the input datasets in terms of the number of executions or samples for the different codes.	62
4.5	List of hybrid features that achieve strong correlations.	70
6.1	Design choices for prototype systems. Specific values shown are for the systems compared in the next section. The job placement choices have been made after comparing different types of placement schemes for each of the network.	104
8.1	Commonly used algorithms.	139
8.2	Cost model based comparison.	143
8.3	Link usage comparison for Scatter and Broadcast.	144
8.4	Link usage comparison for Allgather.	145
8.5	Link usage comparison for Reduce-scatter and Reduce.	145
8.6	Effect of decomposition on time to perform FFT on a $300 \times 300 \times 300$ grid. Representative best values are shown for each of the object counts.	149

CHAPTER 1

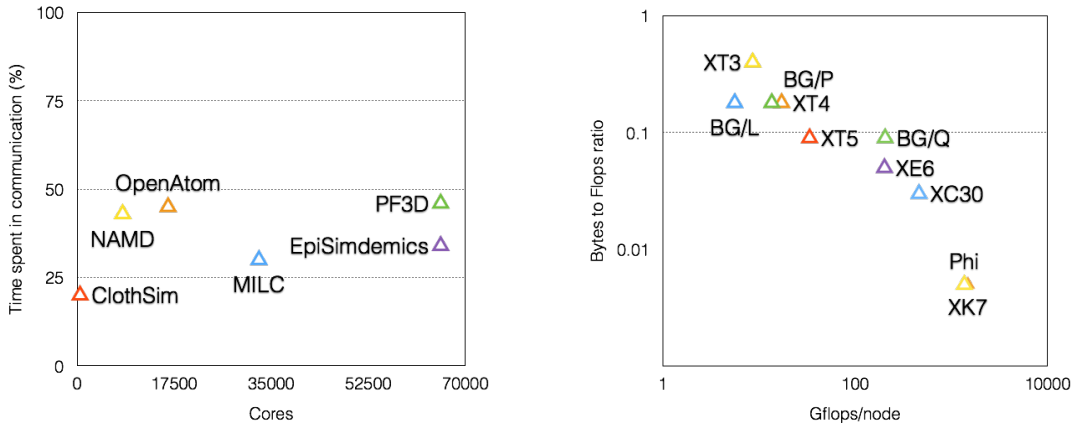
Overview

Efficient communication is a must for completing successful projects, even more so for parallel projects.

Communication is a necessary but overhead inducing component of high performance computing (HPC). It is imperative that it is *optimized* when parallel applications are implemented and executed in order to make the best use of large scale systems. The challenge of optimizing communication in parallel applications is analogous to the challenge of completing a multi-person project in several ways. A successful project requires many things to be done right: creation of a capable team, availability of a suitable work environment, distribution of work among the team members, coordination within the team, etc. Obtaining scalable *communication performance* requires similar tasks at multiple levels described in this section. Here, we use the term communication performance loosely to represent the impact of communication on application execution time.

The communication performance of an application is impacted by several related aspects of a parallel job execution: the network topology of the system used for the job execution, the placement of the job within the system, the message injection and reception mechanism, the routing protocol, suitability of the interaction pattern of the application executed to the network, etc. The multiplicity of these factors provides a challenge as well as an opportunity to study them and optimize performance by exploiting them; this forms a generic theme of this thesis.

As an application developer and user, optimizing for communication is also challenging due to the limited control over the system environment. Unlike the computation resources that are typically known apriori and are fully under the control of an executing application, the availability of network resources may neither be predictable nor be completely under the control of an executing application. For example, the placement of a job and the topology of



(a) Fraction of time spent in communication by scientific applications.

(b) Trend shown by bytes to flops ratio for HPC systems.

Figure 1.1: Motivating examples for research on communication optimization.

allocated nodes is decided by the scheduler based on the availability of resources. In many systems, the network resources may also be shared by multiple jobs.

As a system administrator, the goal is to make the best use of the system. Dependence of application performance on communication make this goal hard to achieve. How should the jobs be placed in the system? What parameters of the network should be universally fixed across jobs, and which should be decided by the users? Questions such as these require significant research to be answered correctly.

The diversity of the applications that are typically executed on large systems adds another level of complexity to the task of optimizing communication. Several studies, such as Kamil et al. [1], show that the communication requirements of various application classes are significantly different. Figure 1.1a shows that many of these diverse applications spend a significant fraction of time in performing communication. As a result, many types of communication optimization techniques are required to improve performance of common applications.

Finally, what makes studying communication performance more important than ever is the increasing scarcity of network resources. In Figure 1.1b, the x-axis represents the computation capability of a node in a given system, while the y-axis shows the bytes (injection bandwidth) to flops ratio (B/F ratio). This ratio, which represents the communication capability vis-a-vis the computation capability, is computed by dividing the injection bandwidth of a node with its computation capability. The trend is easy to notice in Figure 1.1b: as the capability of nodes have increased, the B/F ratio has gone down. In other words, as nodes have become computationally more capable in last 10 years ($256\times$ increase from Blue

Gene/L to Xeon Phi), the B/F ratio has gone down from 0.18 to 0.005.

In summary, optimization of communication for parallel application is a multi-facet challenge, in part due to the multitude of the factors that impact communication performance, and in part due to the diversity of the use cases. As a result, this thesis consists of efforts that have been directed towards developing a better understanding (and associated tools) of different aspects of communication on HPC networks. Broadly speaking, the presented work aims at achieving the following three goals: improve runtime configuration and environment to facilitate communication, propose prediction tools and explore their applications in understanding and optimizing communication performance, and develop software that uses communication-aware algorithms.

A recurring theme in many of the chapters in this thesis is the use of *prediction* methodologies. As supercomputers continue to become larger and more complex, the prevalent practice of customized hand-tuning of applications and testing on production systems is no longer sufficient for efficient parallel executions. This is because such optimizations may take a significant fraction of the available life time of an expensive machine. Additionally, one will have to wait to start doing the tuning until the machine is available. Thus, prediction tools based on data analytics, functional modeling, and detailed simulation are a must to make the best use of systems we have today. Moreover, these tools are required in various ways at different stages of system and application development: during the design of machines with targeted applications being brought into the fold, during offline analysis and optimization of applications, and finally when the applications are launched and executed. Different chapters in this thesis present few such tools, and demonstrate their efficacy for the aforementioned use cases.

1.1 Thesis organization

Chapter 2 presents background and related work on six sub-topics that are closely related to communication and are repeatedly visited in the following chapters. These sub-topics are: common topologies in HPC networks, prevalent interaction or communication patterns, task mapping and job placement, metrics for modeling performance, simulation tools, and communication-centric algorithms.

Chapter 3 is focused on the problem of task mapping and job placement on the torus and dragonfly networks. For the torus network, which has been studied extensively in the past, this chapter suggests a three-step methodology to prepare applications for efficient

production runs. A modeling-based prediction tool, Damsel^{fly}, and job placement studies based on it are the primary contributions of this chapter for the dragonfly network.

Chapter 4 proposes use of machine learning to correlate observed communication performance with various measured and estimated metrics. For many communication kernels and production applications, this chapter shows that the execution time is strongly correlated to a small set of metrics, and fast prediction models can be built using large training sets.

Chapter 5 presents a packet-level network simulator, TRACER, which is a successor to BigSim. By combining BigSim, CODES, and ROSS, TRACER provides a scalable way of simulating large scale networks for real applications accurately. Additionally, it can be used to generate input data for machine learning based methods in a fast manner when its fidelity is reduced.

Chapter 6 makes use of TRACER to compare three commonly used network topologies at the scale of the next generation supercomputers: torus, fat-tree, dragonfly. Using a set of benchmarks and cost prediction models, this chapter provides insights about performance capabilities and cost efficiency of these networks. Such studies can be important when networks are being compared for building future systems.

Chapter 7 also uses TRACER to study the impact of various network configuration parameters, such as link bandwidth, router delay, etc., on the performance of two mini-applications. Via these case studies, this chapter proposes a generic methodology that can be used to study network designs and understand the changes in network behavior because of individual configurations. Using this method, optimal network configuration parameters can be obtained in an efficient manner.

Chapter 8 is dedicated to communication-centric algorithms and software. The first part of this chapter discusses topology-aware algorithms for performing collectives on the dragonfly network. The second part of this chapter presents a 2D-decomposition based FFT library, which provides high performance by utilizing the network better.

Finally, **Chapter 9** summarizes the presented work and suggests possible directions for future work.

Background and Related Work

Significant amount of research has been done on communication and networks for parallel computing in general, and for HPC in particular. This chapter provides a background and summarizes the past work relevant to various topics that this thesis is closely associated with. These topics are divided among the following sections: Topologies in HPC networks 2.1, Interaction patterns 2.2, Task mapping and job placement 2.3, Indicators of performance 2.4, Offline performance prediction 2.5, and Communication algorithms 2.6.

2.1 Topologies in HPC networks

Parallel computers have been built upon many types of interconnection topologies in the last three decades or so. At the one end of spectrum are the linear topologies where $n - 1$ links are used for connecting n routers (or nodes). A *star* or a linear chain are examples of this type. The other extreme is the *full* topology in which every router is connected to every other router. As expected, most of the real systems deploy network topologies that are between these extremes. In this thesis, we pay attention to four of these topologies that have commonly been used in large scale supercomputers: hypercube, fat-tree, torus, and dragonfly.

Hypercube: Geometrically speaking, hypercubes are n -dimensional analog of squares and cubes wherein each dimension is of length two. An n -dimensional hypercube consists of 2^n nodes (or vertices) with $\log n$ neighbors per node. A hypercube of n dimensions can be defined recursively in terms of two $(n - 1)$ -dimensional hypercubes: take two $(n - 1)$ -dimensional hypercubes and connect the corresponding nodes. Many large supercomputers in the late eighties and nineties, such as nCUBE [2], used hypercubes as their network's

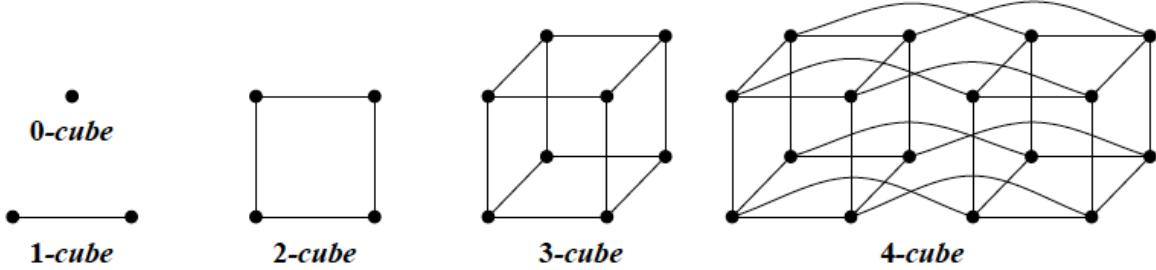


Figure 2.1: Example hypercubes of various dimensions: note the recursive construction of an n -dimensional hypercube using two $n - 1$ -dimensional hypercubes.

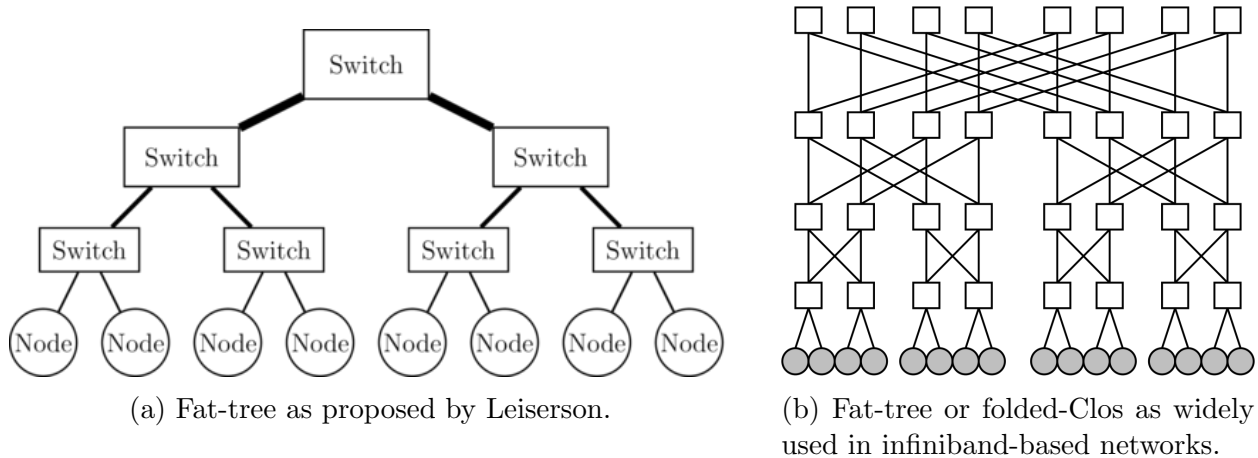
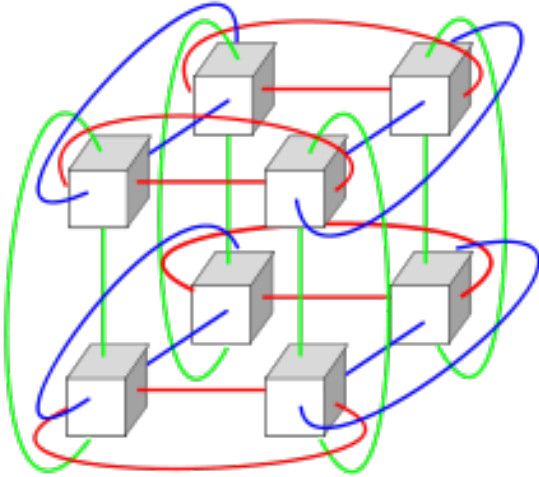


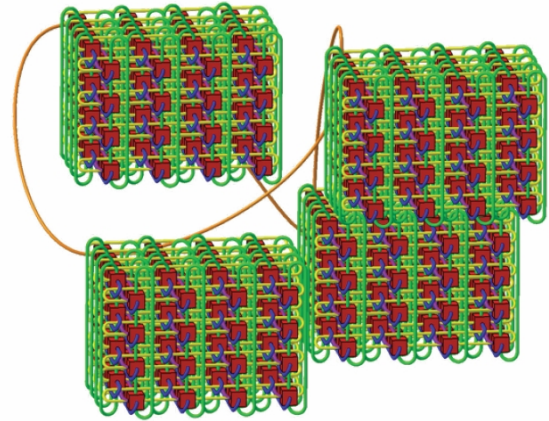
Figure 2.2: Variations of fat-tree/folded Clos topology.

topology. Figure 2.1 shows hypercubes of up to four dimensions and highlights the recursive construction of hypercubes.

Fat-tree, Folded-Clos and variations: Fat-tree topology was proposed by Leiserson as a network with provably efficient communication [3]. The key idea behind the fat-tree network is the following: in a tree topology, the loads on the links increase as we approach the root; thus link bandwidth should be higher for the links closer to the root (Figure 2.2a). The Connection Machine [4] was based on this topology. In terms of practical deployment, fat-tree requires the switches closer to the root to have more ports to support more links. This makes the construction more expensive as building switches with large number of ports is costly. Figure 2.2b shows a variation of fat-tree (which can also be viewed as a folded-Clos [5]) that builds a fat-tree using switches with smaller radix. The key idea here is to replace the fat switches near the root with many smaller switches that logically behave as one big switch. This topology has been widely adopted for deployment of infiniband-based supercomputers [6, 7].



(a) A $2 \times 2 \times 2$ 3D torus (source: [13])



(b) IBM's construction of 5-D torus (source: [14])

Figure 2.3: Torus are constructed by wrapping the corresponding grids at all boundaries.

Torus: In geometry, a torus is 3-dimensional structure generated by revolving a circle about an axis coplanar with the circle [8]. A generic n -dimensional torus can be viewed as an extension of a n -dimensional grid where the edges are wrapped around to form rings. It can also be viewed as a k -ary n -cube. Figure 2.3 shows two tori - one of dimension 3 and other of dimension 5. Torus of various dimension (two to six) have been used in many supercomputers in the last decade [9–12]. Note that commonly deployed torus are asymmetric form of k -ary n -cube, where the length of each dimension (the k 's) can be different.

Dragonfly: Invented by Kim et al. [15], dragonfly is a multi-level dense topology aimed at making use of high-radix router. A typical dragonfly consists of two levels of connections. At the first level, routers are connected via a dense topology, e.g. an all-to-all, to form groups. The groups behave as virtual routers which are then again connected in an all-to-all manner (Figure 2.4). Dragonfly and its variations have been used in recent supercomputer networks such as Cray's XC30 [16], IBM PERCS [17], etc.

2.2 Interaction patterns

Communication patterns of various HPC applications can be significantly different. In their survey of many common applications, Kamil et al. [1] found that the average number of communicating neighbors per core can range from 6 to 256 for different applications executed on 256 cores. In addition to the variation in the number of communicating neighbors, the

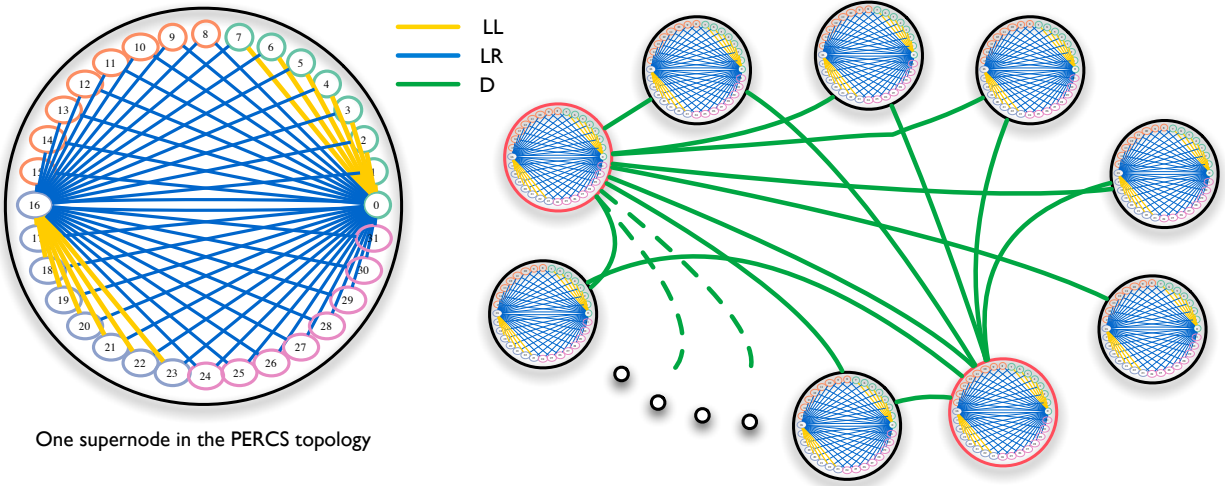


Figure 2.4: Dragonfly: A two-tier network with dense all-to-all connections among logical routers at each level.

choice of neighbors has also been shown to be significantly different. As a result, at one end of the spectrum, processes (or nodes) in HPC applications can have less than 10 neighbors all of which are close to them in terms of the given rank space (and topology). On the other hand, many applications, including the ones that perform parallel FFTs, have all-to-all communication pattern within large subsets of processors that may be far away from each other. To account for these variations, most studies in this thesis have been performed on a set of benchmarks and applications that are representative of distinct communication sets. Here, we provide generic descriptions of these representative patterns:

Permutation/Transpose: In this pattern, each process communicates with only one other process in a given phase of application. The selection of paired processes is often dependent on the input to the application, and may change as the application’s phase changes. For example, in a multi-phase transpose of matrix distributed among a 2D grid of processes, every process interacts with a different process in each of the phase of the transpose operation. Given the data-dependent nature and presence of multiple permutations during an application execution, topology aware mapping of communicating processes is difficult to achieve. Thus, this operation often leads to large hop-counts for each message transferred and stresses the bisection bandwidth of the network.

nD-Stencil: Structured grid based near neighbor communication is one of the most commonly found communication pattern in parallel applications. A nD-Stencil pattern overlays the processes onto a nD-grid. Every process in the grid communicate with $2n$ nearest-neighbor, two in each dimension. Example applications with such patterns include WRF [18]

(2D-Stencil), MILC [19] (4D-Stencil), pF3D [20] (1D- and 3D-Stencil), etc.

Unstructured Near-neighbor: This pattern is the unstructured form of nD-Stencil. A typical formulation consists of an unstructured mesh divided among processes along the mesh edges. As a result, processes have different number of neighbors with whom they may communicate different amount of data. Another scenario which results in this pattern is when certain processes multicast their data to a small subset of processes that are mapped close to them. Example applications with such patterns include material modeling, Cloth Simulation [21], OpenAtom [22], etc.

Many-to-Many/A2A: When a subset of processes communicate among themselves in an all-to-all manner, we obtain the many-to-many communication pattern. Presence of parallel FFT operation often leads to such a pattern in many application such as NAMD [23], pF3D [20], Qbox [24], OpenAtom [22], etc. Parallel IO, parallel sorting, and other similar operations also result in many-to-many communication pattern.

Uniform Spread: In this pattern, every process communicates with a few other processes that are selected randomly from the remaining set of processes. One can also view this pattern as multiple permutation patterns communicating simultaneously. While less commonly induced by application requirement, the uniform spread is commonly observed due to the way jobs are placed by the job scheduler. For example, consider an application with nD-Stencil pattern placed alongside other jobs running on a large system in a non-contiguous manner. Due to this placement, the nD-Stencil pattern manifests itself as a uniform spread pattern.

2.3 Task mapping and job placement

In parallel computing, several techniques have been developed to map communication graphs to hypercubes in the 1980s [25–27] and to torus networks in the early 2000s [28, 29]. More recently, several application and runtime system developers have studied techniques for mapping [30–33] to three-dimensional torus topologies with the emergence of supercomputers like the IBM Blue Gene and Cray XT/XE series. Bhatele et al. [34] explored use of information about application’s communication patterns, such as structured grid and unstructured mesh, and network’s topology (e.g. 3D torus) to create automated tools for generating better mappings. In [34], several methods such as affine mapping, corner to center mapping, step embedding, etc. are described. Most of these schemes are designed to minimize hop-bytes: the total number of hops all the messages will take during the application execution. Fig-

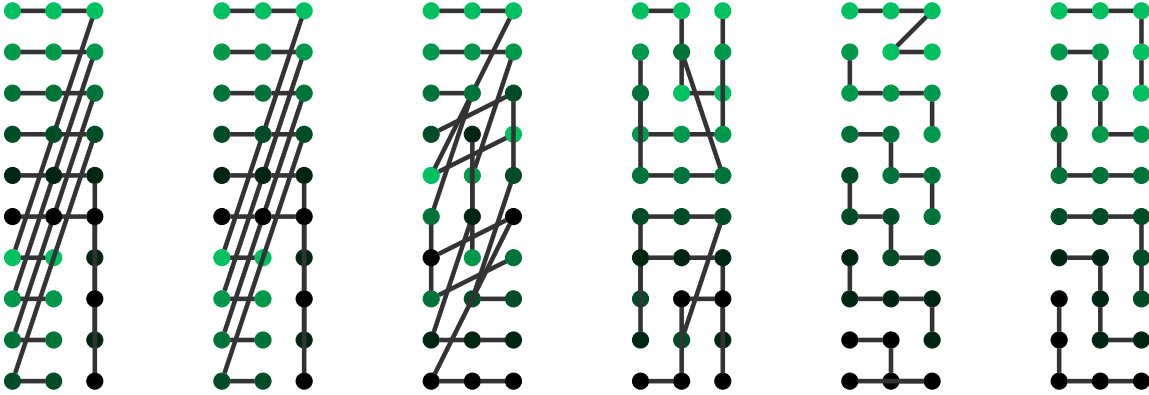


Figure 2.5: Mapping of a 6×5 grid to a 10×3 mesh using different mapping strategies. (source [34])

Figure 2.5 demonstrates how these different methods map a 2D-grid of size 6×5 to a 2D-mesh of size 10×3 , in order to obtain better performance.

Hoefler et al. [35] discuss generic mapping algorithms to minimize contention and demonstrate their applicability to torus, PERCS, and fat-tree networks through mapping simulations of sparse matrix-vector multiplication on up to 1,792 nodes. The algorithms compared in [35] includes a greedy heuristic, recursive bisection, and mapping based on graph similarity. In addition to reducing hop-bytes, these schemes also try to minimize the maximum load on any link in the network.

Fiedler et al. [36] have proposed methods for mapping applications with 2D, 3D, and 4D Cartesian topologies onto Cray systems with 3D torus networks and service nodes. This scheme considers the scenario where nearby nodes are allocated to jobs, preferably as prisms. Phillips et al. [37] have proposed space filling curved based mapping approaches for a even more generic case - mapping of unstructured communication patterns onto an arbitrary allocation. The key idea in the scheme proposed in [37] is to create an ordering of nodes using a topology aware space filling curve, and then deploy recursive bisection method on both network and application topologies. Figure 2.6 shows two stages of recursive bisection onto an ordered list of processes as proposed in [37].

Practical use of task mapping of an HPC application requires generation of an *assignment* of process task IDs or *ranks* to the cores and nodes in the network. Traditionally, programmers have written custom scripts to generate such assignments from scratch. This process is tedious and error-prone, especially with many tasks and high-dimensional networks. Bhatele et al. developed *Rubik* [38], a tool that abstracts several common mapping operations into

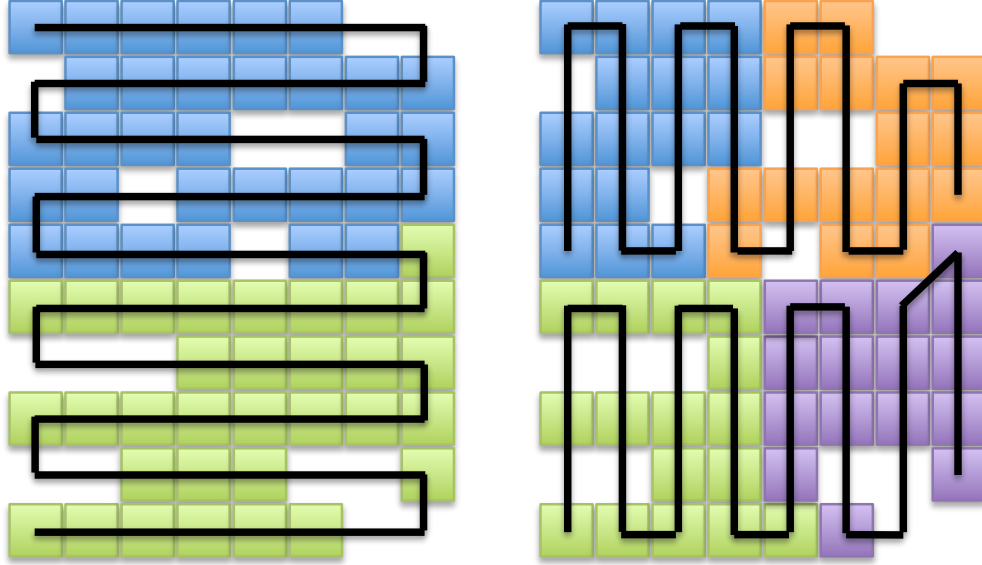


Figure 2.6: Two stages of recursive bisection topology-adapted partition mapping. The dark lines represent the sorted node traversal order for bisection. Grid element color represents recursive bisection domains after each stage.

a concise syntax. Rubik allows complex mappings to be generated using only a few lines of Python code. It supports a wide range of permutation operations for optimizing latency or bandwidth. The full range of transformations possible with Rubik is covered in [38]. Bhatele et al. also developed Chizu, a mapping tool based on graph partitioning. Chizu lets user choose different graph algorithms for recursively partitioning the application and network graphs. Once the partitioning is complete using the selected algorithms, Chizu maps them in a one-to-one manner, and generates the corresponding mapping file.

2.4 Indicators of performance

Let us assume a guest graph, $G = (V_g, E_g)$ (communication graph between tasks) and a host graph, $H = (V_h, E_h)$ (network topology of the parallel machine). M defines a mapping of the guest graph on the host graph (G on H). Several metrics have been proposed in the literature to evaluate communication performance offline by finding the suitability of the mapping M . To the best of our knowledge, the earliest metric that was used to compare the effectiveness of task mappings is dilation [39, 40]. Dilation for a mapping M can be defined as,

$$\text{dilation}(M) = \max_{e_i \in E_g} d_i(M) \quad (2.1)$$

where d_i is the dilation of the edge e_i for a mapping M . Dilation of an edge e_i is the number of hops between the end-points of the edge when mapped to the host graph. This metric aims at minimizing the length of the longest wire in a circuit [39]. We refer to this as maximum dilation to avoid any confusion. We can also calculate the average dilation per edge for a mapping as,

$$\text{average dilation-per-edge}(M) = \frac{\sum_{e_i \in E_g} d_i(M)}{|E_g|} \quad (2.2)$$

Hoeffler et al. overload dilation to describe the “expected” dilation for an edge and “average” dilation for a mapping [35]. Their definition of expected dilation for an edge can be reduced to equation 2.1 above by assuming that messages are only routed on shortest paths, which is true for the IBM Blue Gene and Cray XT/XE family (if all nodes are in a healthy state). The average dilation metric, as coined by Hoeffler and Snir, is a weighted dilation and has been previously referred to as the *hop-bytes* metric by Sadayappan [25] in 1988 and Agarwal in 2006 [30]. Hop-bytes is the weighted sum of the edge dilations where the weights are the message sizes. Hop-bytes can be calculated by the equation,

$$\text{hop-bytes}(M) = \sum_{e_i \in E_g} d_i(M) \times w_i \quad (2.3)$$

where d_i is the dilation of edge e_i and w_i is the weight (message size in bytes) of edge e_i .

Hop-bytes gives an indication of the overall communication traffic being injected on to the network. We can derive two metrics based on hop-bytes: the average number of hops traveled by each byte on the network,

$$\text{average hops-per-byte}(M) = \frac{\sum_{e_i \in E_g} d_i(M) \times w_i}{\sum_{e_i \in E_g} w_i} \quad (2.4)$$

and the average number of bytes that pass through a hardware link,

$$\text{average bytes-per-link}(M) = \frac{\sum_{e_i \in E_g} d_i(M) \times w_i}{|E_h|} \quad (2.5)$$

The former gives an indication of how far each byte has to travel on average. The latter gives an indication of the average load or congestion on a hardware link on the network. They are derived metrics (from hop-bytes) and all three are practically equivalent when used for prediction.

Another metric that indicates congestion on network links is the maximum number of

bytes that pass through any link on the network,

$$\text{maximum bytes}(M) = \max_{l_i \in E_h} \left(\sum_{e_j \in E_g | e_j \implies l_i} w_j \right) \quad (2.6)$$

where $e_j \implies l_i$ represents that edge e_j in the guest graph goes through edge (link) l_i in the host graph (network). Hoeffler and Snir use a second metric in their paper [35], worst case congestion, which is the same as equation 2.6 above.

2.5 Offline performance prediction

Formal models such as LogP [41] and LogGP [42] have been used to analyze the communication in parallel applications for a long time. Subsequently, based on the LogP model, models such as LoPC [43], LoGPC [44], LoGPG [45], LogGPO [46], and LoOgGP [47] were developed to account for network congestion. Simulators based on these models, e.g. LogGOPSim [48], simulate application traces and predict communication performance.

Discrete event simulation has also been extensively deployed to predict communication performance and study communication. BigSim is one of the earliest simulators that supports packet-level simulations [49]. Structural Simulation Toolkit(SST) [50] provides online (skeleton application based) and offline (DUMPI [51] trace based) modes for simulation. Booksim [52] and IBM’s Mambo [53] are sequential cycle accurate simulator that supports several topologies, but are extremely slow for simulating networks of size 10K and higher. There are several other network simulators that are either sequential and/or do not provide detailed packet-level (or flit-level) network simulation. These include the Extreme-scale Simulator(xSim) [54], DIMEMAS [55], PSINS [56], MPI-Netsim [57], OMNet++ [58], and SimGrid [59].

2.6 Communication algorithms

Application-level task-aware mapping has been shown to reduce the communication time for point-to-point communication operations (Section 2.3). However, typically, a good mapping has much less impact on collective operations. This is because in collective operations, the interaction among a large set of processes requires carefully designed algorithms for obtaining good performance. As a result, optimizing collectives via algorithmic design has been an important topic of interest in high performance computing [60]. Most of the existing algo-

rithms can be grouped into two classes. The first class comprises of generic algorithms such as the binomial algorithm and recursive doubling/halving [61], which work well for many network topologies and a wide range of message sizes. Due to their simplicity and broad applicability, these algorithms are part of many MPI implementations including MPICH [62] and IBM’s MPI. The second class of algorithms are specifically tailored for a given network topology. These algorithms outperform the generic algorithms for the specific target topologies and message sizes. For example, Van de Geijn et al. [63] proposed an algorithm for large message broadcast that has been shown to outperform the binomial algorithm [61]. Jain et al. [64] demonstrate that the bucket algorithm [65] can be generalized optimally, both for communication and computation, for an n -dimensional torus.

Optimization schemes for 3D torus networks have been presented in [66] and [64]. Faraj et al. [66] show how to carefully overlay six spanning trees over a 3-dimensional torus without contention. The algorithms in [66] can also be extended to rectangular sub-communicators. In general, when multiple spanning trees are directed towards or away from the root, performance improves for *Broadcast*, *Scatter*, *Gather* and *Reduce* because of increased link throughput utilization by multiple trees. Derived collectives, such as *Allreduce* that is performed by pipelining *Reduce* with *Broadcast*, benefit from two edge-disjoint trees, one in each direction towards and away from the root.

Unlike the collective operations discussed above, an all-to-all operation is extremely difficult to optimize. However, it is a critical operation since it is needed for widely used parallel algorithms such as 2D/3D-FFT. Hence, research has focused on improving its performance for specific cases. Kumar et al. [67] have explored Blue Gene system specific heuristics to improve the performance of all-to-all communication. FFTW [68] provides a generic implementation for 1D-decomposition based parallel n D-FFT, which relies on efficient implementation of MPI’s all-to-all collective for performance. Various efforts have been made to improve the performance of 2D/3D-FFT on specific topologies. For Blue Gene systems, IBM provides a customized implementation of 3D-FFT, which takes advantage of the machine topology [69]. Anthony et al. [70] also did an in depth analysis of parallel 3D-FFT on Blue Gene style mesh topologies. They show that remapping processes on nodes and rotating the mesh by considering the communication properties of specific applications, such as P3DFFT, can reduce the communication time significantly. Special hardware based solutions for achieving fast 3D-FFT have also been explored [71].

Job Placement and Task Mapping

Job placement and task mapping are techniques to optimize communication of parallel applications on the interconnection network without having to modify the source code [34]. These techniques place tasks or processes on compute nodes based on a careful consideration of the interconnection topology between the nodes and the communication behavior of the application. As discussed in Section 2.3, several researchers have studied and exploited job placement and task mapping for improving the performance of their applications.

Topology aware placement of tasks is especially important on torus networks because their large diameters can require messages to travel multiple hops to reach their destination, thereby increasing the burden on the shared links. A torus or mesh network topology has been commonly used to connect compute nodes since the Cray T3D machine was developed twenty years ago. Six of the ten fastest supercomputers in 2014 used a torus network for message passing between compute nodes. At the same time, understanding the tradeoffs of various job placement policies and task mapping algorithms for new networks such as the dragonfly is a must to make the best use of the system. The dragonfly networks also support many routing policies, whose impact on system performance is unknown.

In this chapter, our focus is on exploring answers to the open questions related to torus and dragonfly networks. This research agenda has been decided based on the state-of-the-art of the mapping research, and by anticipating the issues that may be important in coming years.

1. For torus-based systems, it is now accepted that convex or isolated cuboidal allocations are optimal for communication performance [72]. Given a compact allocation, what methods can be used to find good mappings for applications executing on n -dimensional logical grid? To answer this question, we present a step-by-step approach for analyzing communication-intensive applications and finding good topology-aware mappings for

them on torus-based networks.

2. In our past work, we have established that for single job runs on dragonfly networks with near-neighbor communication pattern, randomized placement and/or indirect routing is required for good performance [73]. However, many issues remain unsolved in relation to job placement and routing policies for various communication patterns. In this chapter, we present a performance comparison of mapping and routing schemes when a job that requires n cores is executed on a n -core dragonfly system. Further, we study parallel workloads, in which different jobs (each of size n_i cores) is to be executed on a m core dragonfly system ($m \gg n_i$).

3.1 Task mapping on torus ¹

Traditionally, programmers have written custom scripts to generate assignments of MPI ranks to network nodes. This process is tedious and error-prone, especially with many tasks and high-dimensional networks. We developed *Rubik* [38], a tool that abstracts several common mapping operations into a concise syntax. Rubik allows complex mappings to be generated using only a few lines of Python code. It supports a wide range of permutation operations for optimizing latency or bandwidth, of which we describe a subset here. The full range of transformations possible with Rubik is covered in [38].

Partitioning: Figure 3.1 shows a Rubik script that describes the application’s process grid (a $9 \times 3 \times 8$ cuboid) and a Cartesian network (a $6 \times 6 \times 6$ cube) by creating a “box” for each. Each box is divided into sub-partitions using the `tile` function, resulting in eight $9 \times 3 \times 1$ planes in the application and eight $3 \times 3 \times 3$ sub-cubes in the network. Rubik provides many operations like `tile` for partitioning boxes, allowing users to group communicating tasks. These partitioning operations can also be applied hierarchically.

Mapping: The `map` operation assigns tasks from each sub-partition in the application box to corresponding sub-partitions in the network box. Partitions can be mapped to one another if they have the same size, *regardless of their dimensions*. This means we can easily map low-dimensional planes to high-dimensional cuboids, changing the way in which communicating tasks use the network. Thus, the user is able to convert high-diameter shapes of the application, like planes, into compact, high-bandwidth shapes on the network, like boxes.

Permutation: In addition to partitioning and mapping operations, Rubik supports permu-

¹Based on [74]

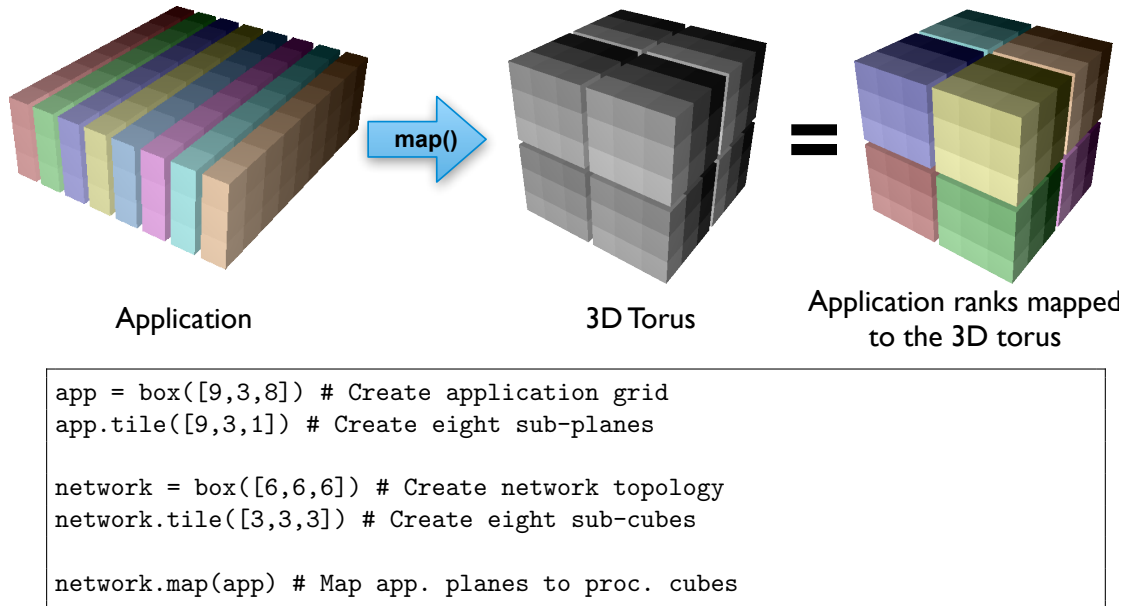


Figure 3.1: Mapping 2D sub-partitions to 3D shapes in Rubik.

tation operations that reorder ranks within partitions. The `tilt` operation takes hyperplanes in a Cartesian partition and maps them to diagonals. Tilting is a bandwidth-optimizing operation – if tasks are laid out initially so that neighbors communicate with one another (e.g., in a stencil or halo), tilting increases the number of routes between communicating peers. Successive tilting in multiple directions adds routes in additional dimensions. Tilting can be applied at any level of the partition hierarchy – to specific partitions or to an entire application grid.

3.1.1 Mapping, congestion and performance

We present a step-by-step methodology to improve application performance using task mapping, developed based on our experience with optimizing production applications on the IBM Blue Gene/Q architecture. There are three steps involved in this process: 1) Performance debugging via profiling, 2) Performance optimization via task mapping, and 3) Performance analysis via profiling and visualization. Each of these steps is broken down further and explained in detail below.

Performance debugging

Application scientists are often unaware of the reason(s) for performance issues with their codes. It is important to determine if communication between parallel tasks is a scaling bottleneck. Performance analysis tools such as mpiP [75], HPCToolkit [76], and IBM's MPI trace library [77] can provide a breakdown of the time spent in computation and in communication. They also output times spent, message counts and sizes for different MPI routines invoked in the code. Some advanced tools can also calculate the number of network hops traveled by messages between different pairs of tasks. The first step is to collect performance data for representative input problems (weak or strong scaling) on the architecture in question.

Performance data obtained from profiling tools can be used to determine if communication is a scaling bottleneck. As a rule of thumb, if an application spends less than 5% of its time in communication when using a large number of tasks, there is little room for improving the messaging performance. If this is not the case, we can attempt to use topology-aware task mapping to improve performance and the scaling behavior. As we will see in the application examples, task mapping can even be used to reduce the time spent in collective operations over all processes.

Performance optimization

There are several tools and libraries that provide utilities for mapping an application to torus and other networks [29, 35, 38, 78–80]. We use Rubik, described earlier, to generate mappings for two production applications we have used, pF3D [20] and MILC [81]. Since the solution space for mappings is so large, there are several factors to consider when trying out different mappings:

- Are there multiple phases in the application with conflicting communication patterns?
- Is the goal to optimize point-to-point operations or collectives or both?
- Is the goal to optimize network latency or bandwidth?
- Is it beneficial to consolidate communication on-node or spread communication on the network?

The previous performance debugging step can provide answers to the questions above and guide us in pruning the space of all possible mappings. Once we have identified a few

mappings that lead to significant improvements, it is crucial to understand the cause of the performance improvements, which is the next step in the process.

Performance analysis

Performance analysis tools can also be used to dissect the communication behavior of the application under different mapping scenarios. Several metrics have been used in the literature to evaluate task mappings and correlate them with the network behavior – dilation [39, 40], hop-bytes [25, 30] and maximum load or congestion on the network [35]. A more detailed analysis on correlating different task mappings with different metrics can be found here [82].

Comparing the communication and network behavior under different mappings can enable us to understand the root cause of performance benefits and help us in finding near-optimal mappings. In this work, we explore three different metrics that influence communication performance:

- Time spent in different MPI routines
- The average and maximum number of hops traveled over the network
- The average and maximum number of packets passing through different links on the network

All three metrics reflect the state of the network and congestion to different extents and correlate, to differing degrees, with the messaging performance of the application. An iterative process of trying different mappings and analyzing the resulting performance can bring us closer to the optimal performance attainable on a given architecture.

3.1.2 Experimental setup

We use Vulcan, a 24,576-node, five Petaflop/s IBM Blue Gene/Q installation on the unclassified (OCF) Collaboration Zone network at Lawrence Livermore National Laboratory (LLNL) for all the runs in this section. The BG/Q architecture uses 1.6 GHz IBM PowerPC A2 processors with 16 cores each, 1 GB of memory per core, and the option to run 1 to 4 hardware threads per core. The nodes are connected by a proprietary 5D torus interconnect with latencies of less than a microsecond and unidirectional link bandwidth of 2 GB/s. Ten links, two in each direction (A, B, C, D, and E), connect a node to ten other nodes on the system. The E dimension has length two, so the bandwidth between a pair of nodes in E is twice the bandwidth available in other directions. When running on Vulcan, the shape of

the torus and the connectivity for a given node count can change from one job allocation to another. The jobs shapes that were allocated for most of the runs in this section are summarized in Table 3.1.

#nodes	A	B	C	D	E	Torus or Mesh
128	1	4	4	4	2	Torus in all directions
256	4	4	4	4	1	Torus in all directions
512	4	4	4	4	2	Torus in all directions
1024	4	4	4	8	2	Mesh in D, Torus in rest
2048	4	4	4	16	2	Torus in all directions
4096	4	8	4	16	2	Torus in all directions

Table 3.1: Shape and connectivity of the partitions allocated on Vulcan (Blue Gene/Q) for different node counts.

Both pF3D and MILC were run on 128 to 4096 nodes. Based on our previous experience with the two applications, the performance sweet spot for hardware threads is at 2 threads per core for pF3D and 4 threads per core for MILC. Both applications were run in an MPI-only weak scaling mode, keeping the problem size per MPI task constant. We used mpiP [75] to obtain the times spent in computation and communication in different MPI routines. We used a tracing library by IBM designed specifically for the BG/Q to obtain the average and maximum number of hops traveled by all messages. An in-house library for accessing network hardware counters was used to collect the packet counts for different torus links.

We compare different partitioning and permutation operations from Rubik with two system provided mappings on BG/Q. ABCDET is the default mapping on BG/Q in which MPI ranks are placed along T (hardware thread ID) first, then E, D, and so on. This mapping fills the cores on a node first before moving on to the next node. In the TABCDE mapping, T grows slowest which is similar to a round-robin mapping. MPI ranks are assigned to a single core of each node before moving onto the next core of each node.

3.1.3 Mapping study of pF3D

pF3D [20] is a scalable multi-physics code used for simulating laser-plasma interactions in experiments conducted at the National Ignition Facility (NIF) at LLNL. It solves wave equations for laser light and backscattered light. With respect to communication, the two main phases are: 1) wave propagation and coupling and 2) light propagation. The former is solved using fast Fourier transforms (FFTs) and the latter is solved using a 6th order advection scheme.

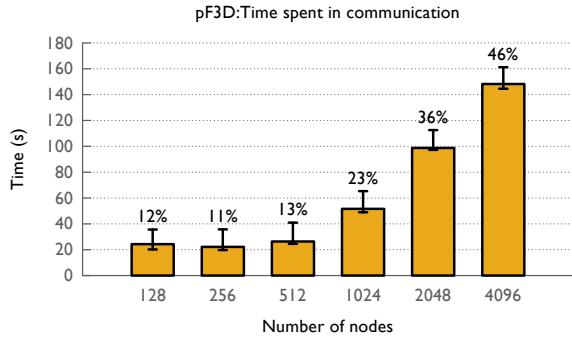


Figure 3.2: Average, minimum, and maximum time spent in communication by pF3D for weak scaling.

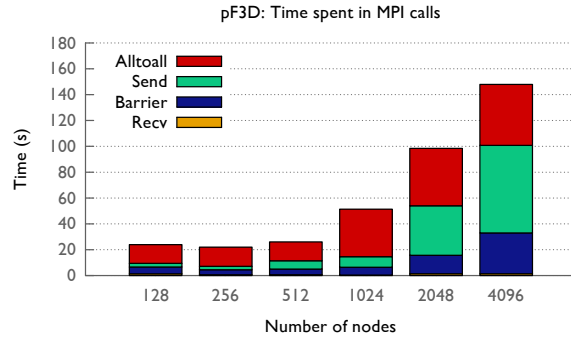


Figure 3.3: Average time spent in different MPI routines by pF3D for weak scaling.

A 3D Cartesian grid is used for decomposing pF3D’s domain among MPI processes. For the input problem that we used in this section, the X and Y dimensions of the process grid are fixed at 32 and 16, respectively. As we scale the application from 4,096 to 131,072 processes, the number of planes in Z increases from 8 to 256. In the wave propagation and coupling phase, the 2D FFT is broken down into several 1D FFTs, one set involving processes with the same X and Z coordinate and another involving processes with the same Y and Z coordinate. The advection messages are exchanged in the Z direction between corresponding processes of the XY planes. MPI_Alltoalls over sub-communicators of size 32 and 16 are used for the FFT phase and MPI_Send and MPI_Recv are used for the advection phase.

Performance debugging: baseline performance

We start with profiling pF3D using mpiP to understand the relative importance of the two phases described above and the contribution of communication to the overall time. Figure 3.2 shows the average, minimum, and maximum time spent in messaging by MPI processes on different node counts. The percentage labels on top of each vertical bar denote the contribution of communication to the overall runtime of the application. For a weak scaling study, we would expect the communication time to be constant, but it continues to grow, especially beyond 1,024 nodes, and adds up to 46% of the total time at 4,096 nodes.

A careful look at the breakdown of this time into different MPI routines (Figure 3.3) shows that the messaging performance is dominated by three MPI routines – MPI_Alltoall from the FFT phase, MPI_Send from the advection phase and MPI_Barrier. The all-to-alls are over sub-communicators of size 32 and 16 and the message sizes between each pair of processes are 4 and 8 KB, respectively. The advection send messages are of size 256 and 384

```

from rubik import *

# processor topology -- A x B x C x D x E x T
torus = autobox(tasks_per_node=32)
numpes = torus.size

# application topology -- mp_r x mp_q x mp_p
mp_r = torus.size / (16*32)
app = box([mp_r, 16, 32])

ttitle = [int(sys.argv[i]) for i in range(1, 7)]
torus.tile(ttitle) # tile the torus

atitle = [int(sys.argv[i]) for i in range(7, 10)]
app.tile(atitle) # tile the application

# map MPI ranks to their destinations
torus.map(app)

f = open('mapfile', 'w') # write out the mapfile
torus.write_map_file(f)
f.close()

```

Figure 3.4: A Rubik script to generate tiled mappings for pF3D.

KB. We spend ~ 200 ms in each send, which is much higher than expected. At 4,096 nodes, we also spend a significant amount of time in an `MPI.Barrier`. We believe communication imbalance due to network congestion manifests itself as processes waiting at the barrier. We hope that an intelligent mapping can reduce this time as well.

Performance optimization: mapping techniques

We now know that for pF3D, the all-to-all and send messages are a scaling bottleneck and any mappings that we develop should try to optimize these two operations. The first two mappings that we tried are ABCDET and TABCDE. ABCDET keeps the all-to-all in the X direction within the node. However, this mapping is very inefficient for the Sends because 32 tasks on one node try to send messages to corresponding tasks on a neighboring node over a single link. The TABCDE mapping spreads the pF3D XY planes on the torus thereby reducing the congestion and time spent in both the all-to-all and the send. The first and second bar in the plots of Figure 3.5 show the reduction in time of those two operations, 78% and 52% respectively on 4,096 nodes (ABCDET is referred to as Default and TABCDE is referred to as RR for round-robin in all the figures).

The next mapping operation that we try with pF3D is tiling which can help group com-

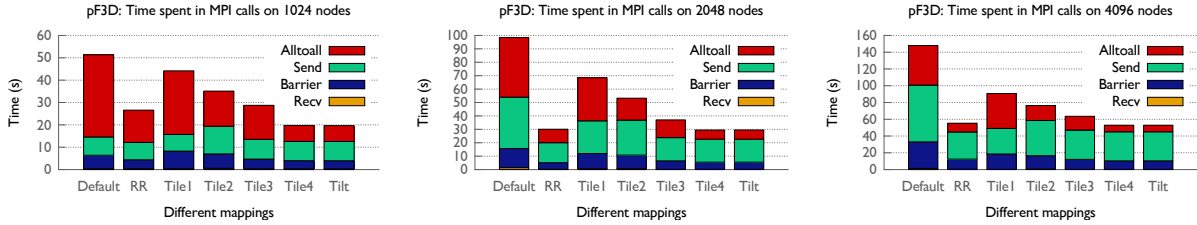


Figure 3.5: Reduction of time spent in different MPI routines by using various task mappings for pF3D running on 1,024, 2,048 and 4,096 nodes of Blue Gene/Q (Note: y-axis has a different range in each plot).

Mapping	Torus tile ($A \times B \times C \times D \times E \times T$)	pF3D tile
Tile1	Use fewest possible dimensions	$8 \times 8 \times 8$
Tile2	$4 \times 4 \times 4 \times 4 \times 2 \times 1$	$8 \times 8 \times 8$
Tile3	Use fewest possible dimensions	$32 \times 16 \times 1$
Tile4	$4 \times 4 \times 4 \times 4 \times 2 \times 1$	$32 \times 16 \times 1$

Table 3.2: Tile sizes used for the Blue Gene/Q 5D torus and pF3D in different mappings.

municating tasks together on the torus. The entire code for doing this is shown in Figure 3.4. Rubik obtains the torus dimensions for the allocated partition automatically at runtime (we only need to specify the number of MPI tasks per node). Then we tile the torus and the application and finally call the map operation.

The various tile sizes that we tried for pF3D at different node counts can be handled as inputs by the same script. We tried four different combinations of tile sizes for the torus and the application which are listed in Table 3.2. *Tile1* and *Tile3* use as few dimensions of the torus as possible. *Tile2* and *Tile4* use as many torus dimensions as possible which results in a $4 \times 4 \times 4 \times 4 \times 2 \times 1$ tile. In *Tile1* and *Tile2*, we make cubic tiles out of the pF3D process grid and in *Tile3* and *Tile4*, we tile by XY planes in the application.

Performance analysis: Comparative evaluation

We now compare the performance of various mappings across different node counts with respect to the reduction in time spent in MPI routines and the amount of network traffic that they generate. Figure 3.5 shows the MPI time breakdown for seven different mappings on 1,024, 2,048 and 4,096 nodes. The first six mappings have already been described above; in the *Tilt* mapping, we create 3D tiles in the 5D torus partition and tilt BC planes in the 3D sub-tori along B. This operation led to significant performance benefits on Blue Gene/P [38] but does not seem to help on BG/Q.

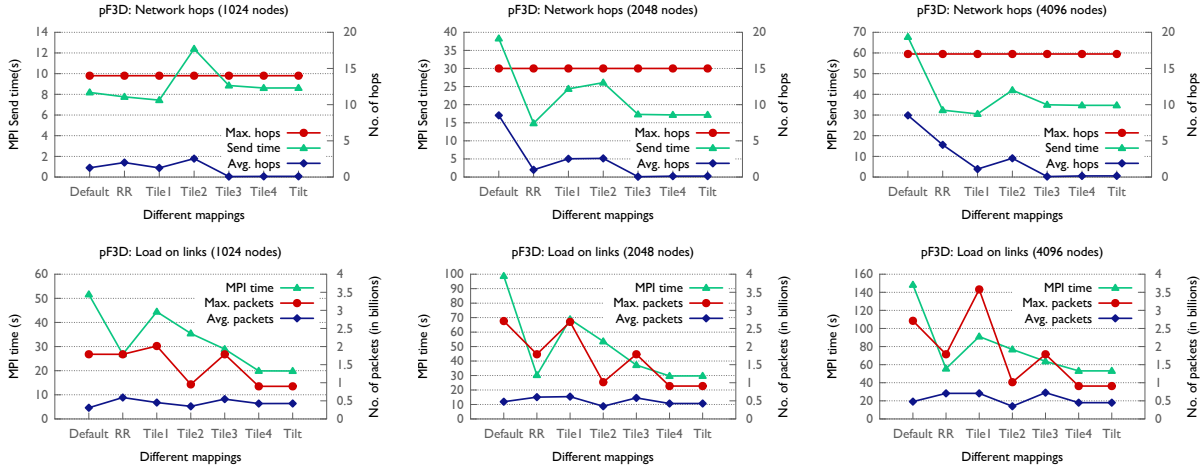


Figure 3.6: pF3D plots comparing the time spent in point-to-point operations to average and maximum hops (top) and comparing the MPI time to average and maximum load on network links (bottom) (Note: y-axis has a different range in each plot).

We can make several observations from these scaling plots. The first trend we notice is that an intelligent tiling of the application on to the torus reduces the time in both the all-to-all and the send operation. We also see a reduction in the time spent in the barrier which suggests reduced congestion on the network and/or less communication imbalance. In the case of pF3D, torus tiles that use all the dimensions of the torus perform better than cubic tiles. This is because the messages, especially the all-to-alls, can use more directions to send their traffic. Finally, the time for sends decreases with better mappings but levels off after a certain point – more analysis on this is described in Section 3.1.3. Overall, *Tile4* gives the best performance by reducing the time spent in communication by 52% on 1,024 nodes and 64% on 4,096 nodes as compared to the default ABCDET mapping.

In Figure 3.6, we use the output from the IBM MPI profiling tool and the network hardware counters library to understand the traffic distribution on the network for different mappings. In the top figures, we see that the maximum number of hops traveled is constant for different mappings (it is also higher as compared to MILC as we will see in Figure 3.10). The time spent in the `MPI_Send` calls closely follows the average number of hops. This suggests that if the application primarily does point-to-point messaging, then reducing the average number of hops is a good idea. The bottom figures plot the average and maximum number of packets passing through any link on the torus network. We notice that the trends for the total MPI time and the maximum load are similar which suggests that it is important to minimize hot-spots or links with heavy traffic on the network.

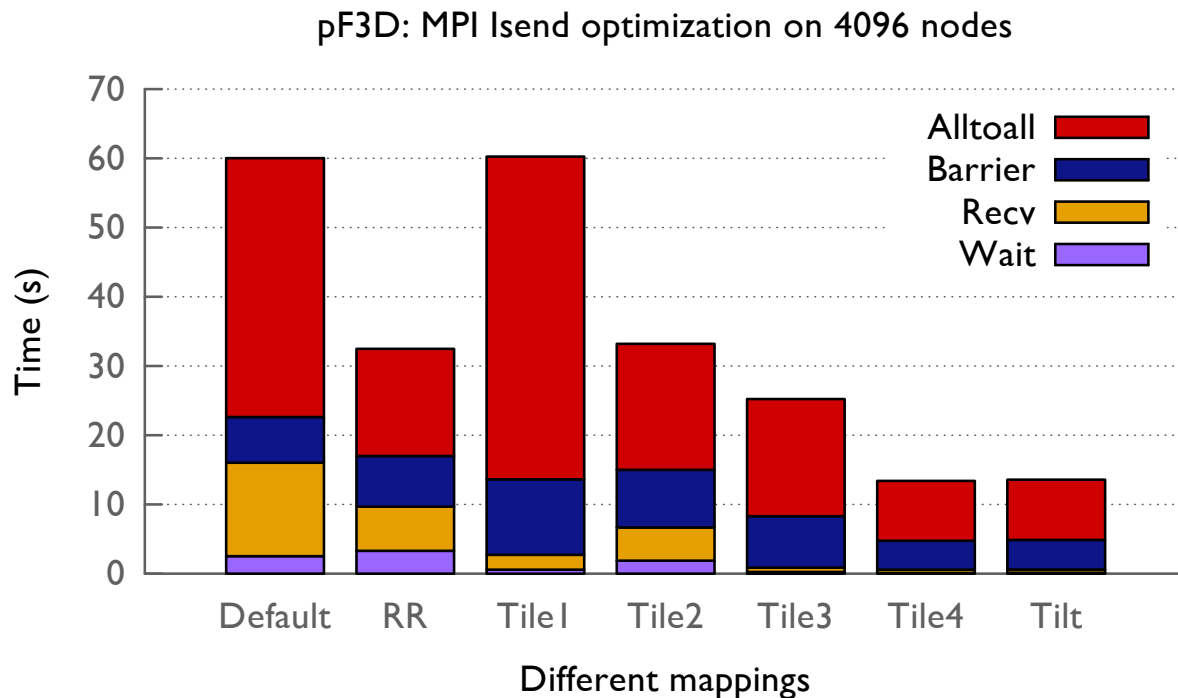


Figure 3.7: Average time spent by pF3D in different MPI routines on 4096 nodes (includes MPI_Isend optimization).

Performance refinement: Iterative process

The plateau in the MPI_Send time reduction (Figure 3.5) prompted us to look further into the problem. We looked at the stack trace to find the origin of these calls in the source code. These calls are made in the `syncforward` and `syncbackward` functions which are a part of the advection phase.

A closer look at the MPI standard and BG/Q’s implementation of large sends revealed that the use of MPI_Send followed by MPI_Recv resulted in an unintended serialization of the advection messages. For large messages, MPI_Send uses a direct copy to the receive buffer and returns after the data is transferred to the destination. However, the location of the receive buffer is known only when an associated MPI_Recv is posted. Hence, when the MPI processes in the rightmost XY plane (which do not have to send any data) post their receives, actual transfer of data begins from the MPI processes in the XY plane penultimate to it. When this transfer is completed, the sends posted by the MPI processes in the penultimate plane return and their receives are posted. At this point, the data transfer from the processes in the plane to its left begins. Such inefficient serialized transfer of data continues till we reach the leftmost XY plane.

The solution is simple – use a non-blocking send, post receives, and then wait on completion of posted sends. We replaced the `MPI_Send` calls with `MPI_Isend` and observed significant improvement in the rates for advection messages. Figure 3.7 shows the new distribution of the time spent in different MPI routines and we can see that most of the time spent in `MPI_Send` has been eliminated. This also has a positive effect on mapping – the same mappings now lead to higher performance benefits compared to the default. For example, *Tile4* reduces the communication time by 77% w.r.t the default mapping as compared to 64% before.

3.1.4 Mapping study of MILC

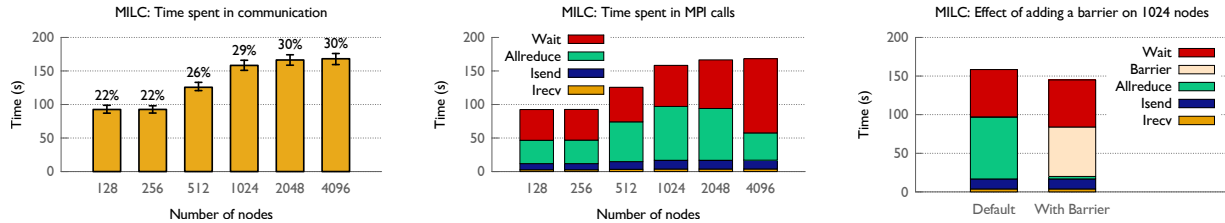
MILC [19], developed by the MIMD Lattice Computation collaboration, is a widely used parallel application suite for studying quantum chromodynamics (QCD), the theory of strong interactions of subatomic physics. It simulates four dimensional $SU(3)$ lattice gauge theory in which strong interactions are responsible for binding quarks into protons/neutrons and holding them together in the atomic nucleus.

We use the MILC application `su3_rmd` distributed as part of the NERSC-8 Trinity Benchmark suite [83]. In `su3_rmd`, the quark fields are defined on a 4-dimensional grid of space time points. The grid is mapped onto a four-dimensional (4D) grid of MPI processes. In every simulation step, each MPI process exchanges information related to the quarks mapped onto it with its nearest neighbors in all the dimensions. Thereafter, computation (primarily a conjugate gradient solver) is performed to update the associated states of the quarks. Global summations are also required by the conjugate gradient solver.

In order to obtain the best performance, MILC was executed on BG/Q with 4-way hyper-threading per core. As a result, when running from 128 to 4,096 nodes, the number of MPI processes ranges from 8,192 to 262,144 respectively. The grid size per MPI process is held constant at $4 \times 4 \times 8 \times 8$, which leads to a weak scaling of the global grid as the node count increases. Determining the dimensions of the MPI process grid is left to the application code.

Performance debugging: baseline performance

As stated in Section 3.1.1, the first step in our methodology is to evaluate the communication characteristics of an application. Figure 3.8a shows that MILC spends between 20% and 30% of its execution time performing communication. As the node count is increased from 128 to 4,096, the overheads of communication increase by 80% (from 92 to 168 seconds).



(a) Average time spent in communication. (b) Time spent in different MPI routines. (c) Adding a barrier before the all-reduce reduces its time significantly.

Figure 3.8: Evaluation of the baseline performance of MILC with the default mapping.

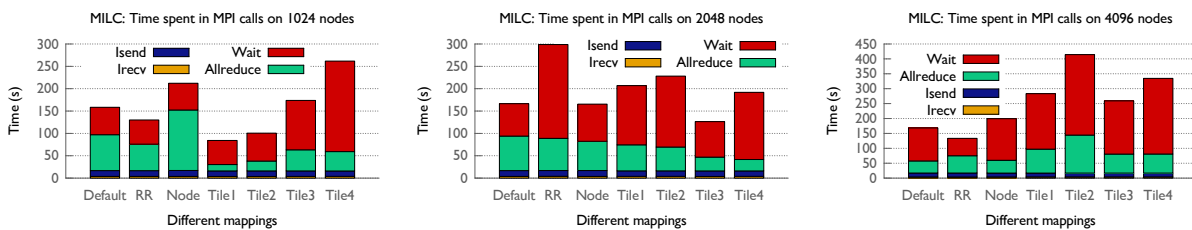


Figure 3.9: Reduction of time spent in different MPI routines by using various task mappings for MILC running on 1024, 2048 and 4096 nodes of BG/Q (Note: y-axis has a different range in each plot).

Given the weak-scaling nature of these experiments, this increase in the MPI time has a negative impact on the overall performance.

The next step is to obtain a detailed profile of MILC to find the predominant MPI routines. Figure 3.8b reveals that `MPI_Wait` (following `MPI_Isend/MPI_Irecv`) and `MPI_Allreduce` over all processes are the key MPI calls. These results are not very encouraging w.r.t using mapping for performance optimization. While mapping may help reduce the wait time, it is typically not useful for improving performance of global collectives.

Reason for the apparently slow all-reduce: While the MPI profiles show significant amounts of time spent in the all-reduce, the data exchanged per all-reduce per process is only 8 bytes. The long time spent in the all-reduce is puzzling because a typical 8-byte all-reduce on 1,024 nodes of BG/Q takes only 77 microseconds. A possible explanation of these results is that MILC suffers from either computation or communication imbalance which leads to increased time spent in all-reduce, an MPI routine that causes global synchronization as a side effect. In order to verify this hypothesis, we did multiple runs in which an `MPI_Barrier` was invoked just before the all-reduce call.

Figure 3.8c compares the profile for the default case with an execution that has a barrier

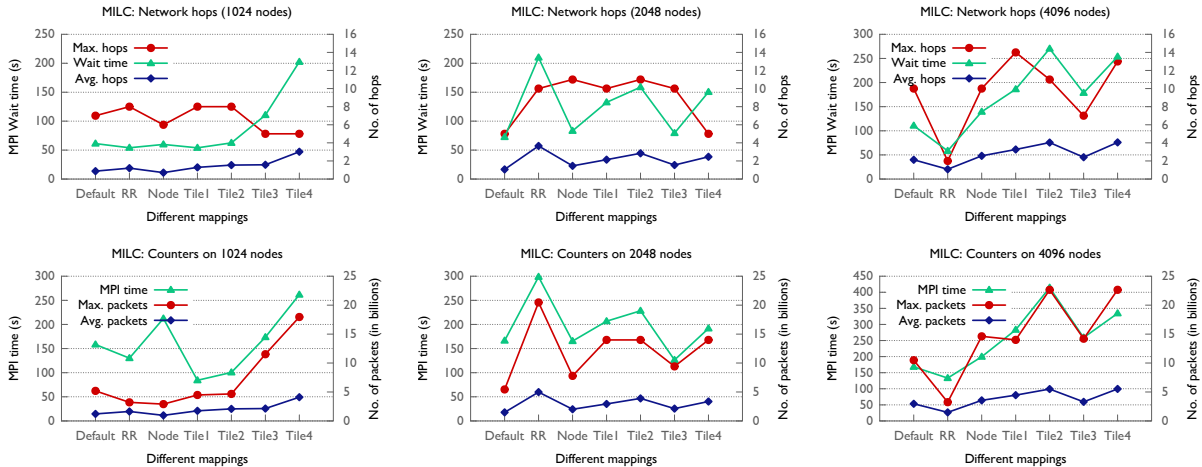


Figure 3.10: MILC plots comparing the time spent in point-to-point operations with average and maximum hops (top) and the MPI time with average and maximum load on network links (bottom) (Note: y-axis has a different range in each plot).

inserted before the all-reduce on 1,024 nodes. Most of the reported all-reduce time in the former case shifts to the barrier time in the latter version. This is also observed on other node counts, which supports our hypothesis. Further, since MILC does not have dynamic computational load imbalance and the all-reduce time of all processes is very high (as confirmed by per process profiling), we can confidently attribute the high all-reduce time to communication-induced variations and imbalance. Hence, it appears that the all-reduce time is a side-effect of the imbalanced or congested point-to-point communication, and can possibly be reduced by using task mapping.

Performance optimization: mapping techniques

As described in the previous section, MILC spends a significant fraction of its execution time in point-to-point communication which can possibly be reduced by mapping its tasks carefully. The simplest variation to try is the TABCDE mapping. Figure 3.9 presents the comparison of the communication time for the default ABCDET mapping (*Default*) and TABCDE (*RR*). Depending on the node count, we observe contrasting effects — for 1,024 and 4,096 nodes, TABCDE reduces the communication time by 20%, whereas for 2,048 nodes, it increases the communication time by 80%. The difference shows up in the time spent in wait and all-reduce, both of which can be attributed to the point-to-point communication (Section 3.1.4).

Another mapping which we called the *Node* mapping blocks communicating MPI processes as a sub-grid and places them on hardware nodes (with 64 MPI processes) This is a natural

choice for mapping of structured applications such as MILC. Surprisingly though, such blocking does not improve the performance; for most cases, *Node* mapping increases the communication time. As a result, we avoid blocking and scatter nearby ranks instead (as in TABCDE) when generating tile-based mappings with Rubik.

For the tile-based mappings, we attempt to map sub-grids from MILC to similar sub-grids on the BG/Q torus. For example, *Tile1* maps a $4 \times 4 \times 4 \times 4$ sub-grid of MILC to a $4 \times 4 \times 4 \times 4$ sub-grid of BG/Q along its first four dimensions (A, B, C, D). Other mappings (*Tile2*, *Tile3*, and *Tile4*) perform similar tilings on different symmetric and asymmetric sub-grid sizes. Most of these choices were guided by the observed performance and profiling information for these experiments that is summarized in Figure 3.9.

In a manner similar to *RR* and *Node* mappings, we observe significant variations in the impact of tile-based mappings on the communication time. *Tile1* and *Tile2* reduce the communication time significantly on 512 and 1,024 nodes, but increase the communication time on other node counts. Similar observations about other mappings can be made. None of the tile-based mapping we attempted were able to reduce communication time on 4,096 nodes.

Performance analysis: comparative evaluation

In this section, we attempt to find the cause for varying impact of different mappings on various node counts presented in Section 3.1.4. Figure 3.10 (top) shows the time spent in wait and the average and maximum hops traveled by point-to-point messages. The average and maximum number of 512-byte packets on the network links and the total MPI time is shown in Figure 3.10 (bottom). For MILC, since most of the communication volume is due to the point-to-point communication, the average hop curves are very similar to the overall average packet curves.

Overall, no correlation is observed between the wait times and maximum hops. This is expected since the message sizes in MILC are a few kilobytes and thus not latency bound. Moreover, per hop latency overhead on BG/Q is very low. However, the average hops and the maximum packets follow trends similar to the wait time and the MPI time respectively, with a few aberrations. These observations are in line with our previous results on correlating performance and metrics [82].

For the default mapping, similar values for average hops and maximum packets are observed on 1,024 and 2,048 nodes which translates into similar wait and MPI times. At 4,096 nodes, the average hops doubles which results in increased wait time. However, the MPI time remains the same due to reduced all-reduce time, indicating a better communication

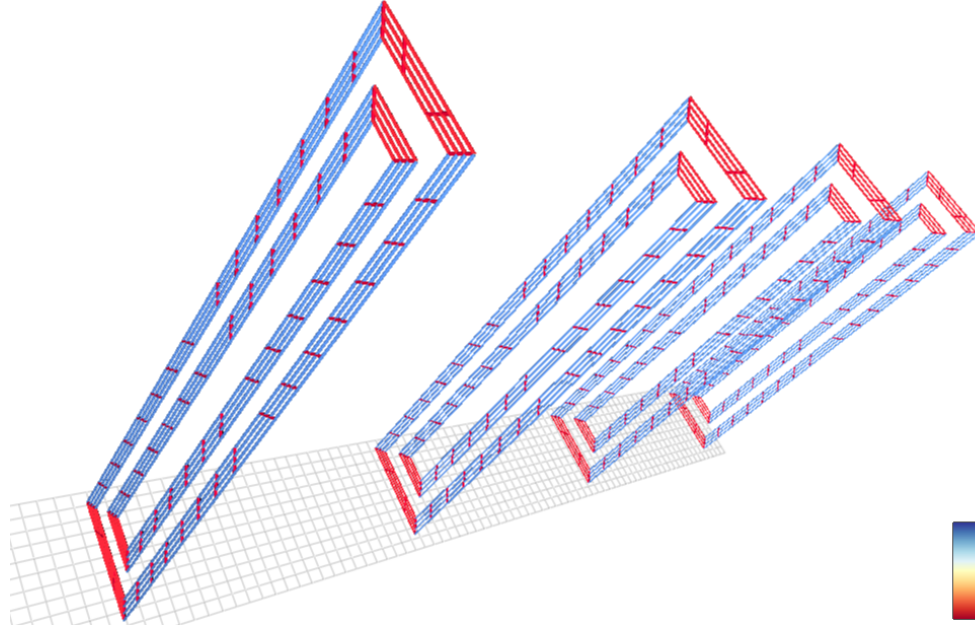


Figure 3.11: Four sub-tori showing the D (blue), C (red, long), and B (red, short, diagonal) links for the same E. The colors represent the number of packets passing through individual links.

balance. In contrast, for TABCDE, both the average hops and maximum packets are significantly higher on 2,048 nodes. As a result, TABCDE has a very high wait time and MPI time on 2,048 nodes. On other node counts, use of TABCDE halves the maximum packets in comparison to the default mapping, and hence also reduces communication time.

On 2,048 and 4,096 nodes, the *Node* mapping provides higher average and maximum packets in comparison to the default mapping. As a result, it also shows higher wait time and MPI time. At 1,024 nodes, we do not see similar trends. This needs a more detailed study (as is done by Jain et. al [82]). The four tile-based mappings follow similar trends: a mapping that provides lower average hops and maximum packets on a node count shows lower wait and MPI times.

Network visualization of packets

We explore network traffic in more detail using the BG/Q visualization module in Boxfish. This module extends Boxfish’s 3D torus visualization [84] which projects the links of the 3D torus into two-dimensional planes. For the 5D torus of BG/Q, multiple such planes are displayed to cover the two extra torus directions. By changing which torus directions compose the planes, the links of different directions can be examined. Figure 3.11 shows an example focusing on the C and D torus directions for the TABCDE mapping using 2,048

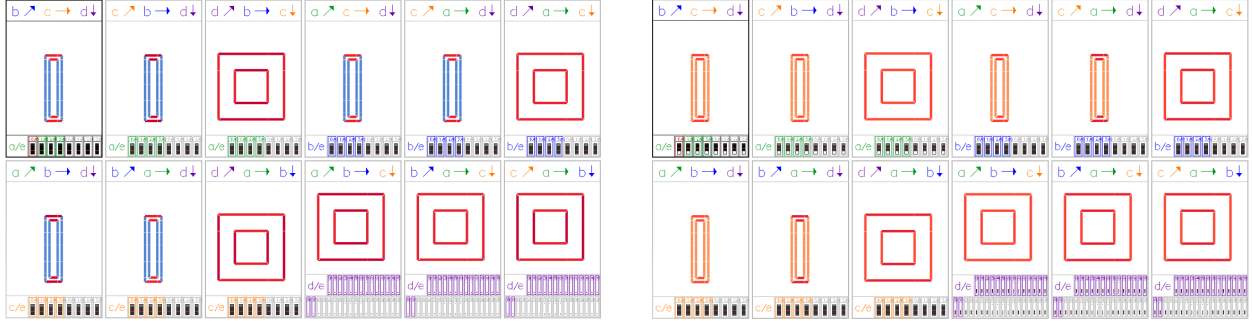


Figure 3.12: Minimaps showing aggregated network traffic along various directions for the TABCDE (left) and the *Tile3* mappings (right).

nodes. The displayed traffic is the network hardware counters data obtained from MILC runs.

We first look at previews of the planes (*‘minimaps’*) which are colored by aggregated packets taken across all but two torus dimensions. This provides an overview of link behavior in all directions. As each minimap shows two directions aggregated along a third (and the short E direction), they are twelve in total. Figure 3.12 shows the minimaps for the TABCDE and *Tile3* mappings of MILC run on 2,048 nodes. Traffic in the D direction for the TABCDE mapping is high while traffic in all other directions is low. This holds for all minimaps showing the D direction, indicating that this is true for all D links and is not affected by other directions. We also examined the individual links (Figure 3.11) and verified that there is no significant variation. In comparison, while the *Tile3* mapping has heavier traffic in the D direction, it is still relatively low. These observations are consistent with our findings in Figure 3.10, which shows higher maximum traffic for the TABCDE mapping than the *Tile3* mapping. Further, the visualization reveals that the maximum traffic is not due to outliers, but is caused by uniformly heavy use of a single torus direction.

3.1.5 Discussion and summary

In this section, we have presented a step-by-step methodology to optimize application performance through the technique of topology-aware task mapping. We have learned several lessons in the process of performance analysis and optimization of the applications, pF3D and MILC, on IBM Blue Gene/Q using this methodology. These are some interesting observations that might be useful to others optimizing their code on torus networks:

- The default mapping or blocking on the physical node may not yield the best performance even for near-neighbor codes.

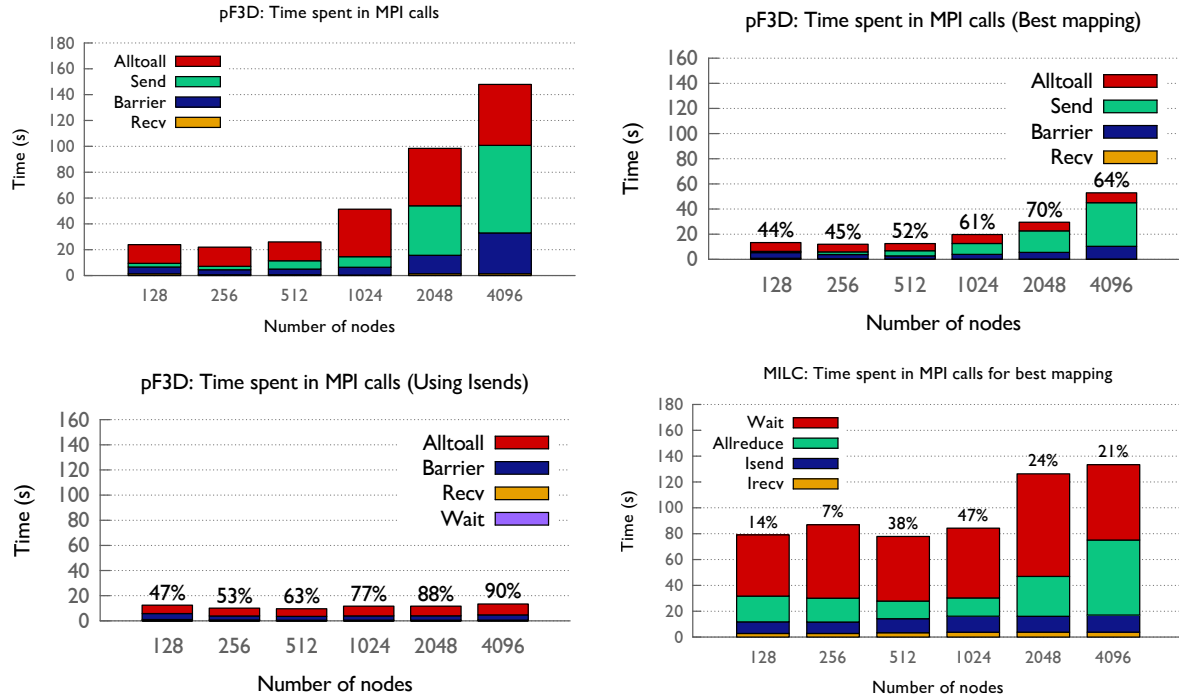


Figure 3.13: pF3D: A scaling comparison of the time spent in different MPI routines with the default mapping (top-left), best mapping discovered, and with the best mapping using the Isend optimization. A scaling comparison of the benefits of task mapping for MILC is also shown (bottom-right). The percentage values shown are improvement over the default mapping.

- Mappings that spread traffic all over the network may lead to better performance for some parallel applications.
- For certain communication patterns, using all dimensions to distribute network traffic may provide better performance rather than confining traffic to a few dimensions.
- Sometimes, computational or communication imbalance or delays due to network congestion can manifest themselves as wait time or time spent in a collective. Careful mappings can reduce this time.
- It takes several iterations to improve the performance of a parallel code. It may appear that intuitive mappings are not leading to expected performance gains. This can happen when the scaling bottleneck is elsewhere.

For the particular applications we study as part of this work, we were able to improve the performance of pF3D and MILC significantly. Figure 3.13 shows the time spent by pF3D in different MPI routines for the default mapping, the best mappings we found, and the

best mappings combined with the Isend optimization. The labels in the plots denote the percentage reduction in communication time compared to the default ABCDET mapping. The top right plot shows that the best mappings can reduce the time spent in all-to-all, while the bottom left plot shows that using Isends instead of Sends can improve the performance further. Tiled mappings improve the communication performance of pF3D by $2.8\times$ on 131,072 processes and the Isend modification improves it further by $3.9\times$. Figure 3.13 also shows the performance improvements obtained with the best mappings for a scaling run of MILC (21% reduction in MPI time at 262,144 processes).

3.2 Job placement on the dragonfly network ²

In the dragonfly topology, high-radix routers are used to organize the network into a two-level all-to-all or closely connected system. The presence of these multi-level hierarchies connected through network links opens up the possibilities for different routing strategies and job placement policies. Unlike the extensively studied torus network, the best choices of message routing and job placement policies are not well understood for the dragonfly topology. In this section, we propose a functional model and use it to compare various routing strategies and job placement combinations for different communication patterns executed on a dragonfly network. In particular, we attempt to find answers to the following questions:

- What is the best combination for single jobs with communication patterns such as unstructured mesh, 4D stencil, many-to-many, and random neighbors? These patterns represent production scientific applications routinely run on NERSC machines [86,87].
- What is the best combination for parallel job workloads in which several applications are using the network simultaneously?
- Is it beneficial for jobs in a workload to use different routing strategies that are more suitable for them in isolation? What is the best placement policy in this situation?

Three things distinguish this work from the previous communication and congestion modeling work on dragonfly networks. First, we consider different alternative routings with adaptivity and study their impact on network throughput. Second, we consider representative job workloads at supercomputing sites and simulate different routings and job placement

²Based on [85]

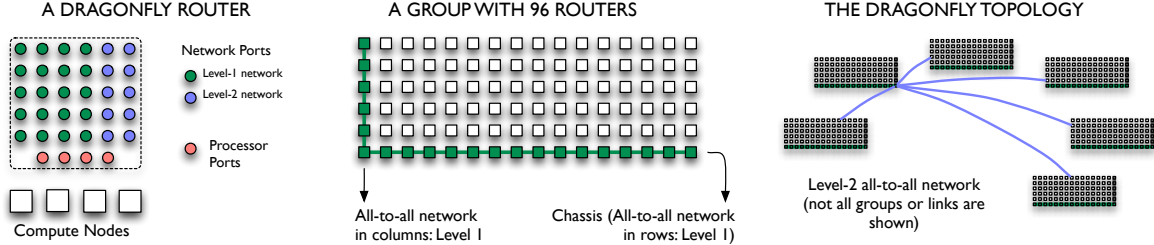


Figure 3.14: The structure of a dragonfly network.

strategies for these workloads. Third, we presents analysis for the dragonfly network at an unprecedented scale (8.8 million cores).

3.2.1 The dragonfly network

Multi-level direct networks have been proposed recently by several researchers as a scalable topology for connecting a large number of nodes together [15–17, 88]. The basic idea behind these networks is to have a topology that resembles an all-to-all at each level of the hierarchy which gives the impression of a highly connected network. Further analysis would show that the network is built using high-radix routers that only exist at the lowest level. The connections between these routers create an appearance of several all-to-all connected direct networks at multiple levels of the hierarchy.

Two prominent implementations of multi-level direct networks are the PERCS interconnect by IBM [17] and the Cascade system by Cray [16]. We focus on the Cascade system which is based on the dragonfly topology designed by Kim et al. [15]. The Cascade (Cray XC30) system uses the Aries router as its building block and has been used in supercomputers such as Edison at NERSC, Lawrence Berkeley National Laboratory and Piz Daint at the Swiss National Supercomputing Center.

We use the dragonfly topology to build a prospective 100+ Petaflop/s system. The parameters for this prototype machine are inspired by the Cray Cascade system. We have, however, simplified the router and link bandwidths for ease of modeling. The building block is a network router with 30 network ports and 4 processor ports (Figure 3.14). Each network router is connected to four compute nodes (of 24 cores each) through the processor ports. Sixteen such routers form a chassis and six chassis are combined together to form a group ($16 \times 6 = 96$ routers in total). Each network router is connected to all other routers in its chassis (15 ports) and to the corresponding routers in five other chassis (5 ports). These links along rows and columns in the group are called level 1 (L1) links in this section. The remaining 10 ports are used to connect to network routers in other groups. These inter-group

links form the second level (L2) of the network. L1 and L2 links together form a two-level direct network.

We take 960 such groups comprised of 96 routers (384 nodes) each to build a very large dragonfly system. This machine has 8,847,360 cores (8.8 million) and extrapolating the Edison system — a peak performance of 164.5 Petaflop/s. Two major differences between the prototype machine used in the section and the Cray Cascade system are: 1. There is only one L1 link between each pair of routers along the column whereas the Cascade machine has three such links leading to three times the bandwidth in that dimension, 2. Cray only allows for 240 groups which leads to 4 links connecting each pair of groups and hence much higher bandwidth.

3.2.2 Prediction methodology for link utilization

In order to compare the relative benefits of different job placement policies and routing strategies, we have developed a model that generates the traffic distribution for all network links given a parallel communication trace. Our hypothesis is that the traffic distribution is indicative of the network throughput we can expect for a given scenario [73, 82, 89]. The inputs to this model are:

- A network graph among dragonfly routers, $N = (V, E)$.
- An application communication graph for one time step or phase in terms of MPI ranks, $A^C = (V^C, E^C)$.
- A job placement/mapping of MPI ranks to physical cores.
- A routing strategy, \mathfrak{R} .

The model accounts for contention on network links and outputs the expected traffic on all network links for each phase of the application. All communication in one time step or phase is assumed to be occurring simultaneously on the network and all messages for the phase are considered to be in flight. For each phase, an *iterative solve* is performed to get the probabilistic traffic distribution on the links. Only one iteration may be needed for simple cases, such as the direct routing. The iterative solve in the model is described below.

Initialization: The input network graph N gives us the peak bandwidths on all network links. We define two other copies of this graph — $N^A = (V^A, E^A)$, which stores the bandwidths that have already been allocated to different messages; and $N^R = (V^R, E^R)$, which stores the remaining link bandwidths that can still be allocated in subsequent iterations. For edge l in these graphs, this relationship holds: $E_l = E_l^A + E_l^R$. At initialization, $E_l^A = 0$ and $E_l^R = E_l$ for all edges.

Iteration: The **do** loop below describes the iterative solve which is central to our traffic prediction model:

do *until no message is allocated any additional bandwidth*

1. For each edge (message), m in E^C , obtain a list of paths, $P(m)$ that it can send its packets on from the source to the destination router for a given routing \mathfrak{R} .
2. Derive the “request” count for each link using the $P(m)$ sets for all messages. The request count is the total number of messages that want to use a link; store the request counts for all links in another copy of the network graph, $N^{RC} = (V^{RC}, E^{RC})$.
3. For each path, p in $P(m)$ for each message m in E^C , calculate the “availability” of each link in p . Availability of a link l is its remaining bandwidth divided by its request count, E_l^R/E_l^{RC} . Each link on path p allocates additional bandwidth to message m which equals the minimum of the availabilities of all links on that path.
4. Decrement remaining bandwidth values in N^R and increment values in N^A based on the freshly allocated bandwidths on the links in the previous step.

end do

Post Processing: For each message, the model assumes that its packets will be divided among the paths on which it was allocated bandwidth during the iterative solve. Depending on the routing protocol \mathfrak{R} , the fraction of a message that is sent on different paths is computed differently. Thus, we obtain the traffic on a link l as,

$$traffic(l) = \sum_{\forall m \in E^C} f_p \text{ if } l \in p, \forall p \in P(m)$$

where f_p is the fraction of the message assigned to path p in the set $P(m)$.

This iterative model is generic and can be used for any routing by selecting appropriate schemes for finding $P(m)$ in Step 1, deciding the request counts N^{RC} in Step 2, finding the link availability in Step 3, and deciding the f_p in post processing. The specific schemes used for different routings are described in detail next.

Routing-specific enhancements to model: The model described so far has been implemented as a scalable MPI-based parallel program. For most parts, the parallelism is obtained by dividing the processing of the messages among the MPI processes. The implementations for different routing schemes build upon the generic model and customize it to improve the prediction capability and computation time. In the following description of the

routing schemes that are based on schemes proposed by Kim et al. [15], it is assumed that a message is sent from the source router s to the destination router d .

Static Direct (SD): In this scheme, a message from s to d is sent using the shortest path(s) between s and d . If multiple shortest paths are present, the message is evenly divided among the paths. For the dragonfly interconnect described in Section 3.2.1, the maximum number of hops for SD routing is 5 — two L1 hops in the source group, one L2 hop, and two L1 hops in the destination group.

For the evaluation of SD, only one iteration is needed to find all shortest paths that a message can take. Once those paths are determined, the message is divided equally among those paths during the post processing. Note that since this routing does not make use of the request count and availability computed in Step 2 and Step 3 respectively, our implementation skips those steps of the iteration.

Static Indirect (SI): In this scheme, for each packet created from a message, a random intermediate router i is selected. The packet is first sent to i using a shortest path between s and i . It is thereafter forwarded to d using a shortest path between i and d . For the given interconnect, use of an intermediate router results in the maximum number of hops for SI to be 10.

Ideally, for packet-level SI routing, only one iteration is needed to find all the indirect paths (like direct routing). However, storing all indirect routes requires very large amount of memory. To address the memory concern, our implementation goes over the packets in the message one by one, and assigns them to a randomly generated indirect path. Processing each packet individually leads to extremely high workload making this routing the most time consuming to evaluate.

Adaptive Direct (AD): The AD routing adds adaptivity to SD — if multiple shortest paths are present between s and d , the message is divided among the paths based on the contention on those paths. The iterative solve is suitable for adaptive routing given that it allows a message to request more bandwidth on resources that have leftover bandwidth iteratively. It also allows messages that can use multiple paths to get more bandwidth. In a typical run, we ran the iterative solve till convergence is reached, i.e. no message is able to obtain any more bandwidth for any of its paths.

Customization: In Step 2, instead of assigning equal weights to all requests of a message to the links of the paths it can use, the requests are weighted based on the minimum remaining bandwidth on any link of the paths. For example, if a message could be sent on two paths with 50 and 100 units of minimum remaining bandwidth on the links of those paths respectively, the requests to the links on those paths are given weights 0.33 and 0.66 respectively. Such weighted requests are helpful in adaptively selecting links that are less

congested. Also, the size of a message is considered while deciding the weights of the requests. This allows for favoring larger messages which may increase the overall throughput of the network as described next. In Step 3, on receiving several requests for a link from various messages, instead of equally dividing the remaining bandwidth to all requests, the division is weighted based on the weights of the requests. During post processing, the messages are divided among the paths in proportion to the bandwidth allocated on those paths so that the effective traffic on all links is equalized (as opposed to the static division done by SD).

Adaptive Indirect (AI): The AI routing is related to SI routing in a manner similar to the relation between SD and AD. For each packet sent using AI routing, the intermediate router, i , is selected from a randomly generated set of routers, based on the contention on the corresponding paths.

Customization: The implementation for this routing also uses the schemes described for adaptive direct routing. However, while adaptive direct routing uses the same set of paths in every iteration for a message, it is impractical to use thousands of paths in every iteration as required by the indirect routing. As a result, we used a set of 4 indirect paths selected afresh in every iteration. However, this may overload the links of the paths used in initial iterations since more bandwidth is typically available during the start. In order to overcome this bias, we added the concept of *incremental* bandwidth. In this method, at the very beginning, only a fraction of the maximum bandwidth of the links is available for allocation to the messages. In each iteration, more bandwidth is made available incrementally for allocation. This kind of increment of available bandwidth is continued until we have exposed all of the maximum bandwidth of the links. In our experiments, we exposed an additional fraction ($\frac{1}{f}$) of bandwidth in each of the first f iterations. Prediction results with varying f suggested that beyond $f = 50$, incremental exposure of bandwidth has no effect on the predictions.

Adaptive Hybrid (AH): A hybrid of AI and AD leads to the AH routing. In this scheme, for sending each packet, the least contended path is selected from a fixed size set of shortest paths and indirect paths. The indirect paths in the set are generated randomly for every packet of the message. AH is implemented using the same schemes as described for AI. To allow for use of direct paths in each iteration, the set of paths consists of 4 paths — up to two direct paths and the remaining indirect paths, instead of 4 indirect paths used for AI. This helps in biasing the routing towards direct paths if congestion on them is not high. In the current implementation of the model, we have assumed global knowledge of congestion (e.g. a router can estimate queue lengths on other routers). Hence, in terms of the original terminology used by Kim et al. [15], the model predicts link utilization for UGAL-G routing, which is an ideal implementation of Universal Globally-Adaptive Load-balanced (UGAL) routing.

3.2.3 Evaluation setup

For the routings described in the previous section, we study the dragonfly interconnect using the presented prediction framework for many job placement policies and communication patterns. In this section, we briefly describe these job placement policies, list the communication patterns, and explain the experimental setup.

Job placement

Job placement refers to the scheduling scheme used to assign a particular set of cores in a particular order for execution of a job. The ordering of the cores is important because it determines the mapping of MPI ranks to the physical cores. We explore the following schemes that have been chosen based on our previous work on two-tier direct networks [73] and the schemes that are currently in use at supercomputer centers that host Cray XC30, a dragonfly interconnect based machine.

Random Nodes (RDN): In this scheme, the job is allocated randomly selected nodes from the set of all available nodes in the system. The cores of a nodes are ordered consecutively, while the nodes are ordered randomly. Random placement may be helpful in spreading the communication uniformly in the system, thus resulting in higher utilization of the links.

Random Routers (RDR): The RDR scheme increases the level of blocking by allocating randomly selected routers (set of four nodes) to a job. The cores attached to a router are ordered consecutively, but the routers are ordered randomly. The additional blocking may help in restricting the communication leaving the router. It also avoids contention within a router among different jobs running on different nodes of the router.

Random Chassis (RDC): This scheme allocates randomly selected *chassis* to a job. The cores within a chassis are ordered, but the chassis are randomly arranged. The additional blocking may limit the number of hops to one L1 link for the messages of a job with communicating nearby MPI ranks.

Random Groups (RDG): The RDG scheme further increases the blocking to groups. This may be useful in reducing the average pressure on L2 links by restricting a significant fraction of communication to be intra-group. However, it may also overload a few L2 links if the groups connected by a L2 link contains nearby MPI ranks that communicate heavily.

Round Robin Nodes (RRN): In this scheme, a job is allocated nodes in a round robin manner across the groups. The cores of a nodes are ordered consecutively, while the nodes are ordered in a round robin manner. Such a distribution ensures uniform spreading of a job in the system.

Table 3.3: Details of communication patterns.

Communication Pattern	Number of Processes	Messages per Process (TDC)	Message Size (KB)
Unstructured Mesh	8,847,360	6 - 20	512
Structured Grid	$80 \times 48 \times 48 \times 48$	8	2,048
Many to many	$180 \times 128 \times 384$	127	100
Uniform Spread	8,847,360	6 - 20	512

Round Robin Routers (RRR): The RRR scheme is similar to the RRN scheme, but allocates routers instead of individual nodes to a job in a round robin manner.

Communication patterns

Kamil et al. [1] have defined topological degree of communication (TDC) of a processor as the number of its communication partners. They study a large set of important applications and show that the TDC of common applications vary from as low as 4 to as large as 255. In order to span a similar range of TDC and study a representative set of common communication patterns [86, 87], the patterns listed in Table 3.3 have been used. Each of the pattern is described in more detail as we analyze prediction results for it in Section 3.2.4.

The communication graphs for each of the pattern is generated either by executing them using AMPI [90], which allows us to execute more MPI processes than the physical cores, or by using a simple sequential program that replicates the communication structure of these patterns.

Prediction runs setup

The parallel code that implements the proposed model was executed on Vesta and Mira, IBM Blue Gene/Q installations at Argonne National Laboratory. For each run, three input parameters were provided: 1) communication pattern based on MPI ranks, 2) mapping of MPI ranks to physical cores, and 3) system configuration including the routing strategy. Depending on the communication pattern and the routing, different core counts were used for the parallel runs. Typically, for SD and AD routing schemes, 512 cores were used to complete the simulation in ~ 5 minutes. For the remaining routings, 2,048 cores were used to simulate the lighter communication patterns, such as structured grid, in ~ 30 minutes. For heavy communication patterns, e.g. many to many, 4096 – 8192 cores were required to finish the runs in up to two hours.

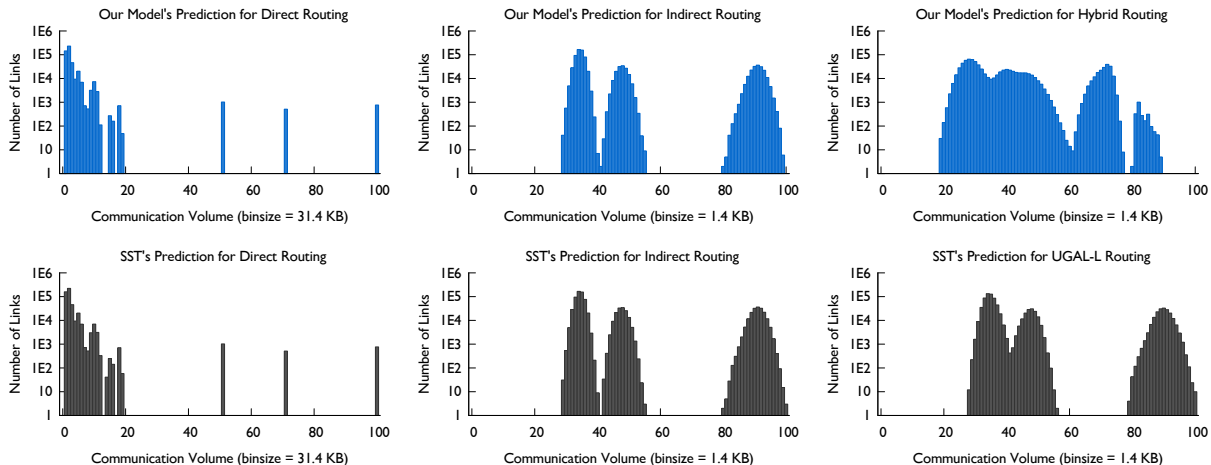


Figure 3.15: Comparison of the predictions by the presented model with predictions by SST/macro, a packet-level simulator, for a 4D Stencil simulated on a 36,864 router system.

Model validation

We validate our proposed model against SST/macro, a packet-level discrete-event simulator [50]. For these validation runs, we use the structured grid communication pattern from Table 3.3 at a smaller scale. The prototype system considered here is one-third the size of the full system (36,864 routers). However, we assume one active MPI rank per router to ensure that the predictions using SST/macro can be obtained in a reasonable time frame.

The left plots (top and bottom) in Figure 3.15 show the histograms of the predicted traffic distributions for direct routing using our model and SST/macro. The two histograms are very similar which attests that the proposed model closely resembles the predictions of a packet-level simulation for direct routing. Similar results are seen for indirect routing (center plots in Figure 3.15) which validates the model for indirect routing. For hybrid routing, we were not able to use SST/macro for a one-to-one validation because SST implements UGAL-L (a localized version of UGAL), while our model assumes global knowledge. Nevertheless, we present the predictions for SST’s UGAL-L and our model’s routing schemes in the right plots (top and bottom) in Figure 3.15. We observe that the predictions by SST’s UGAL-L routing are very similar to its predictions using indirect routing. This is possibly due to the localized view of the queues on a router; the queues for direct routes get filled quickly for large messages, hence diverting the traffic towards indirect routes. In contrast, the hybrid model is able to offload heavily used links (due to its global knowledge) and shift many links to left bins in comparison to indirect routing.

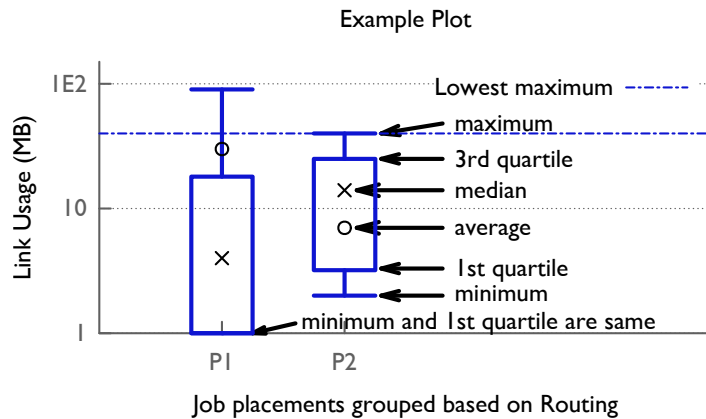


Figure 3.16: Example to explain the data displayed in the plots.

3.2.4 Predictions for single jobs

The first half of the experiments are focused on understanding network throughput for single job executions on a dragonfly network. We begin this section with a brief guide on how to analyze the box plots presented in the rest of the section. Following it, the four communication patterns are studied in detail. Finally, we present prediction results for the case in which the many-to-many pattern is executed in isolation on the system with variation in the number of cores used by it.

Description of the plots

Figure 3.16 shows a typical box plot used in this section. The x-axis contains combinations of routing strategies and job placement policies, which are grouped based on the routing strategy. The log scale based y-axis is the amount of traffic flowing on links in megabytes. For each combination of job placement and routing, six data points are shown — the minimum traffic on any link, the first quartile – 25% of links have lesser traffic than it, the median traffic, the average traffic on all the links, the third quartile – 75% of links have lesser traffic than it, and the maximum traffic on any link. The plot also shows a horizontal dotted blue line that indicates the lowest maximum traffic among all the combinations.

Very high value of maximum traffic relative to other data point indicates network hotspots . Hence, it is a good measure to identify scenarios whose throughput is impacted by bottleneck link(s). The average traffic is an indicator of the overall load on the interconnect. It is helpful in finding scenarios that reduce total traffic and hops taken by the messages. Comparing the average with median is valuable for estimating the distribution. If average is significantly

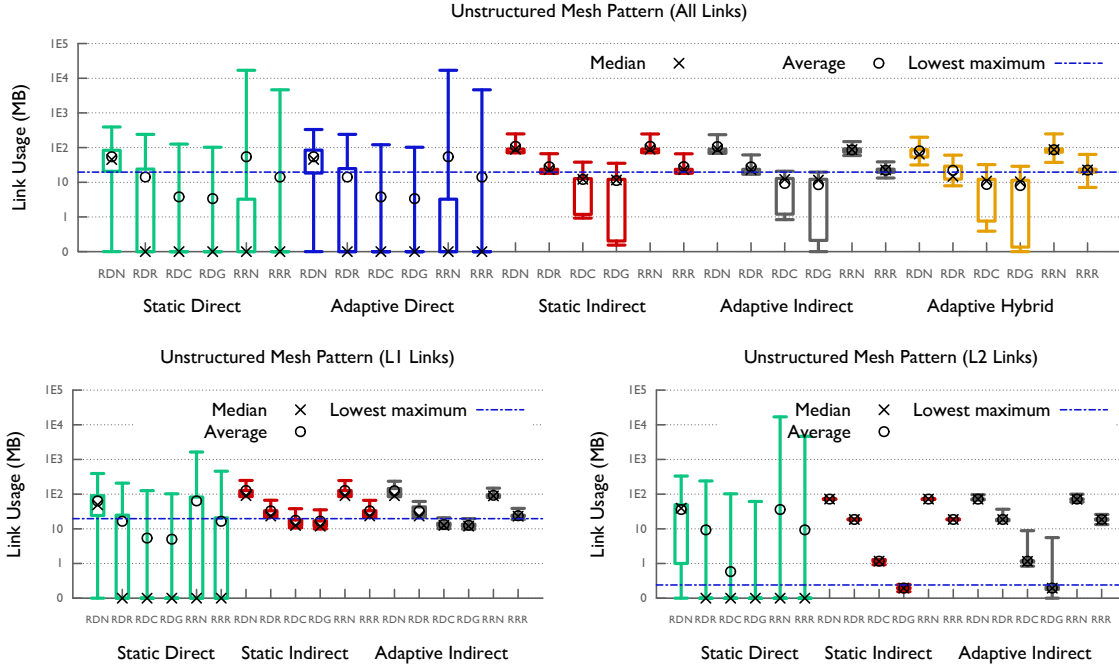


Figure 3.17: Unstructured Mesh Pattern (UMesh): blocking helps in improving the traffic distribution.

higher than the median (P1 in Figure 3.16), the distribution is skewed to the right — most of the links have relatively low traffic, but a long tail stretches to the right. In contrast, if median is higher than the average, the distribution is skewed to the left — most of the links have more traffic than the average, but a long tail stretches to the left. Finally, the quartiles can be used to find more information about how much fraction of the links had what volume of traffic flowing through them. Overall, we suggest that *a distribution with closer values of these data points is good for network throughput*. In case of similar distributions, lower values are better for throughput.

Unstructured Mesh Pattern (UMesh)

In this pattern, each MPI process r communicates with 6 – 20 other MPI processes in its neighborhood (within range $[r-30, r+30]$). Such a pattern is representative of unstructured mesh based and particle in cell (PIC) codes with space filling curve based mapping of MPI processes (e.g. Morton ordering).

Effect of Job Placement: Figure 3.17 (top) presents the expected link utilization when UMesh is executed on the full system. It can be seen that as we increase the blocking in job placement, the maximum, the average, and the quartiles decrease significantly. For UMesh

with many communicating nearby MPI ranks, this trend is observed because increasing blocking from nodes to router avoids network communication. Additionally, it may also decrease the number of hops traversed by messages, since it places most communicating MPI processes within a chassis or a group (as we move from RDR to RDC and RDG).

Effect of Indirect Routing: Comparison among routings shows that the use of any form of indirect routing leads to an increase in average traffic on the links, a trend that is seen in all results presented in this section. This is expected since indirect routing forces use of extra hops. However, indirect routing also leads to a more uniform distribution of loads on the links which is demonstrated by the closes values of the quartiles. Also, the median is closer to the average for indirect routing, in contrast with direct routing for which median is mostly zero (indicating a distribution skewed to the right). Note that although indirect routing increases the average, owing to a better distribution, the maximum is never worse than the direct routings for a given job placement. These characteristics indicate better network throughput for indirect routing in comparison to direct routing.

We also observe that for direct routing with RRN and RRR placements (shown for SD in Figure 3.17 (bottom)), only a few L2 links are being used heavily, thus increasing the overall maximum. These are the L2 links that connect the consecutive groups which are used by the communication among nearby MPI ranks mapped to the nodes and routers placed in a round-robin manner. Indirect routing offloads these L2 links by distributing the traffic to other unused L2 links.

Effect of Adaptivity: We observe that the expected traffic for adaptive versions of the routing schemes have very similar distribution to the static version with similar or lesser corresponding values for the data points of interest. In particular, for RDC and RDG, the AI routing scheme reduces the maximum traffic by 50% in comparison to its static counterpart, SI. We attribute this improvement to unloading of overloaded L1 links. As shown in Figure 3.17 (bottom), comparison of the average suggests that the L1 links are more loaded which is expected given the dominant nearby MPI rank communication in UMesh. For RDC and RDG, the AI routing is able to improve the distribution of traffic on L1 links, and thus reduces the maximum traffic.

Structured Grid Pattern (4D Stencil)

Based on a four-dimensional nine-point stencil, this pattern is representative of the communication pattern in MILC, a Lattice QCD code [81]. The MPI processes are arranged in a 4-D grid, with each process communicating with its 8 nearest neighbors in the four dimensions. As a result, this pattern has lesser MPI rank based communication locality in

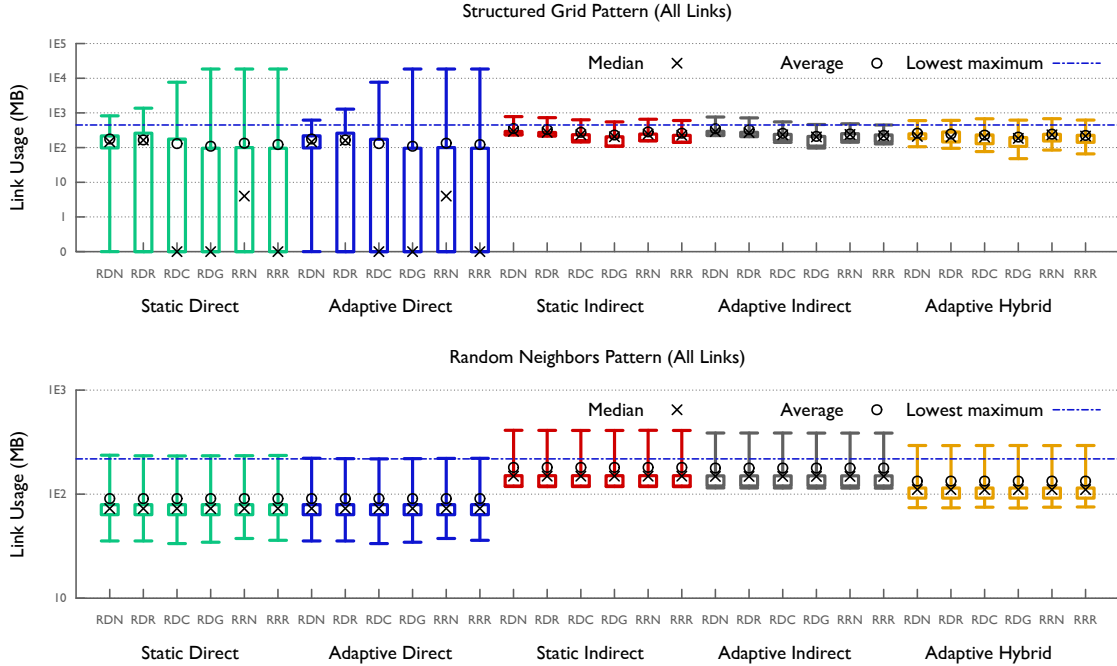


Figure 3.18: Structured Grid Pattern (4D Stencil) and Random Neighbors Pattern (Spread).

comparison to UMesh. For 4D Stencil, two of an MPI process’ communicating partners are its immediate MPI rank neighbors, but the remaining six neighbors are placed incrementally further away from it. For the configuration provided in Table 3.3, two of the neighbors are 48 MPI ranks away, the next pair is 2,304 ranks away, and the final two are 110,592 ranks away.

Effect of Job Placement: Figure 3.18 (top) shows the traffic distribution predictions for 4D Stencil. For direct routings, in a manner similar to UMesh, the average and the quartiles decrease as blocking is increased, although the decrease in average is significantly lesser when compared to UMesh. However, in contrast to UMesh, the maximum traffic increases as we increase the blocking. We suspect that the increase in the maximum is due to high traffic on a few L2 links — links that connect groups which contain many pairs of communicating MPI processes. Such scenarios may arise when blocking is performed at chassis and group levels. In this case, communication between corresponding consecutive MPI processes in two sets that are roughly 48, 2304, or 110,592 MPI ranks apart may result in large number of communicating pairs, thus overloading a few L2 links. To verify this assumption, we first studied the histogram for L2 link utilization (shown in Figure 3.19). It can be seen that while most of the L2 links are unused, a few are overloaded. Then, we identified these links using the raw link usage data and found them to be suspected links, hence verifying our assumption.

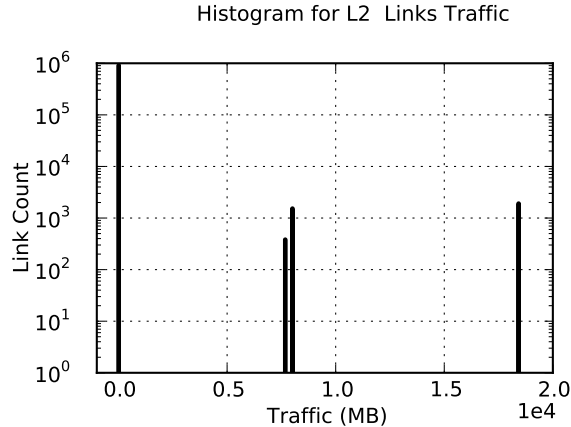


Figure 3.19: 4D Stencil: distribution of traffic on L2 links for RDG.

Effect of Indirect Routing: The skewness caused by the overloading of a few L2 links for direct routing is eliminated by the use of indirect routing. As shown in Figure 3.18 (top), indirect routing leads to a better distribution of traffic on the links. However, as we saw for UMesh, it also increases the average traffic on the links. These results are consistent with our past work on two-level direct networks in which 4D Stencil was also used as communication pattern [73].

Effect of Adaptivity: Use of AI further decreases the variation in traffic distribution. For many job placements (RDG, RRN, RRR), use of AI lowers the maximum traffic by up to 25%. Similar to UMesh, this gain is based on a better distribution of traffic on L1 links which leads to reduced maximum traffic. The adaptive hybrid routing provides a distribution that is similar to AI, but is marginally skewed by use of direct routes.

Many to Many Pattern (M2M)

In this pattern, the MPI processes are arranged in a 3-D grid with subsets being created along the Y axis. Within subsets of size 128, an all-to-all operation is performed. Such a pattern is representative of applications that perform many parallel Fast Fourier transform, e.g. pF3D [20], PARATEC, NAMD [91], and VASP [92]. Using the configuration presented in Table 3.3, an MPI process's communicating partners are separated by multiples of 384, i.e. a process r typically communicates with MPI ranks such as $r+384$, $r-384$, $r+2*384$, $r-2*384$ etc. Depending on the position of a process in the 3D grid of the processes, the number of partners that are to the left and to the right of an MPI process varies. Also, as was the case with 4D Stencil, each MPI process in a set of consecutive MPI processes typically communicates with the corresponding MPI process in another set if the two sets

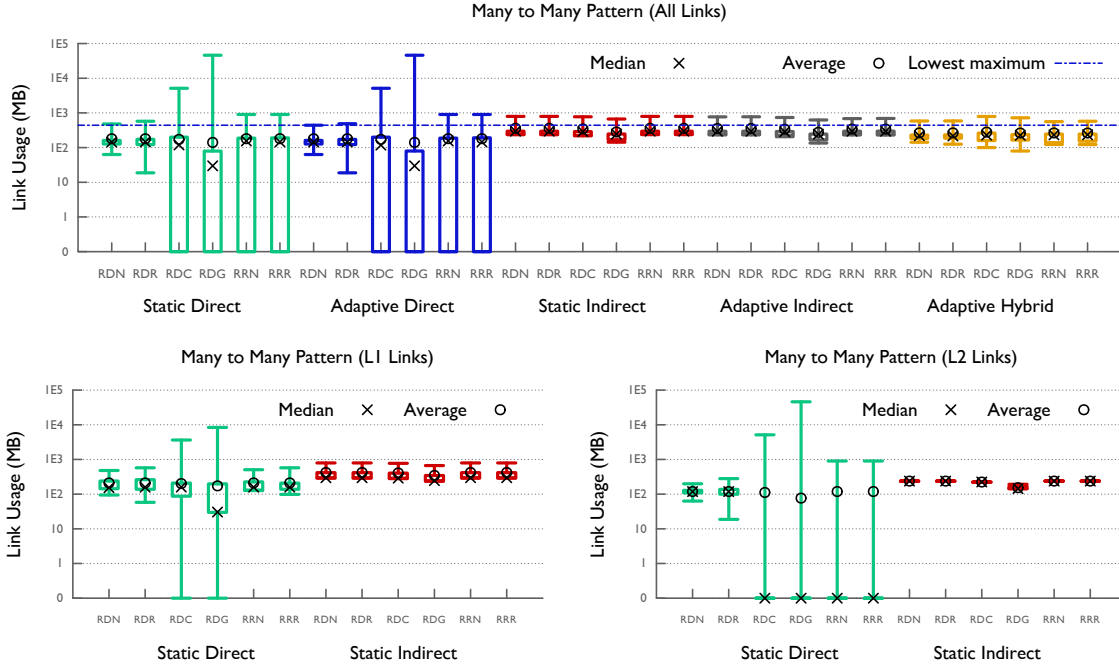


Figure 3.20: Many to many pattern (M2M): direct routing with randomized placement has lower average and maximum traffic.

are 384 ranks apart on an average.

Effect of Job Placement: Figure 3.20 shows the prediction results for M2M. In a manner similar to 4D Stencil, while the average and the median decreases on increasing the blocking for direct routing, albeit in lower proportions, the maximum traffic increases significantly. This increase is attributed to the overloading of certain L2 links as shown by the huge difference between the third quartile and the maximum in Figure 3.20 (bottom). This skewness is due to the non-uniform spread of communicating pairs described in the previous paragraph.

Effect of Indirect Routing: Use of indirect routing helps in offloading the overloaded L2 links, but it increases the load on L1 links (Figure 3.20 (bottom)). The extra load on L1 links is expected since indirect routing doubles the number of hops on an average. However, unlike the benchmarks we have seen so far, the maximum traffic is lower for direct routing with randomized placements and minimal blocking (RDN and RDR). We hypothesize that this is induced by a good distribution of traffic on links by randomized placement. The lower nearby values of the minimum, the median, and the quartiles for direct routing with randomized placement confirms this hypothesis. As a result, for M2M, direct routing is more likely to provide higher network throughput. We believe that such a distribution was not obtained for UMesh and 4D Stencil because of the fewer number of communicating partners

with better MPI rank locality.

Effect of Adaptivity: The adaptive versions of the static routings had a positive but limited impact on the distribution of traffic. This is in part due to the limited opportunity available for adaptivity in already uniform distribution (for randomized placements and indirect routing). For cases with skewed distribution, e.g. SD with RRN, the skewness is caused by a few L2 links that are the only path available for the messages to traverse from one group to other (Figure 3.20 (bottom)). As a result, adaptivity cannot improve the distribution. The adaptive hybrid yields a distribution that resembles AI, but unlike earlier, use of direct routes helps it improve upon AI.

Random Neighbors Pattern (Spread)

This pattern spreads the communication uniformly in the system by making each MPI process communicate with 6 – 20 neighbors selected randomly. In applications that perform computation aware load balancing, e.g. NAMD, or are not executed on near-by physical cores, such communication pattern arise. Figure 3.18 (bottom) shows the expected distribution of traffic for execution of Spread on the full system.

The first thing to observe is that almost all links are utilized irrespective of the job placement and the routing. This is a direct impact of the spread of the communicating pairs that the benchmark provides. Another effect of the spread is the minimal impact of the job placement on the load distribution. Next, we note that while the average quality of the distribution has improved, the gap between the maximum and other data points (average, median and quartiles) has increase significantly for indirect routings. Similar observation can be made for direct routing with randomized placement if we compare with the results for M2M. Further analysis of L1 and L2 links traffic distribution shows that such a skewness is caused by overloading of certain L1 links. We believe this is caused by non-uniformity in the communication pattern — randomization of communication patterns is probably not uniformly distributing them.

The next important observation from the Figure 3.18 (bottom) is the lower values of all data points (minimum, quartiles, average, and maximum) for direct routing in comparison to the indirect routing. This result is similar to what we described in M2M — given a sufficiently distributed communication pattern, indirect routing only adds extra traffic because of the extra hops it takes. Finally, we note that the adaptive versions of the routings reduce the maximum traffic by up to 10%. Other than that, they provide a very similar distribution. As we saw in M2M, the AH routing provides a distribution similar to AI with lower maximum traffic due to use of direct routes.

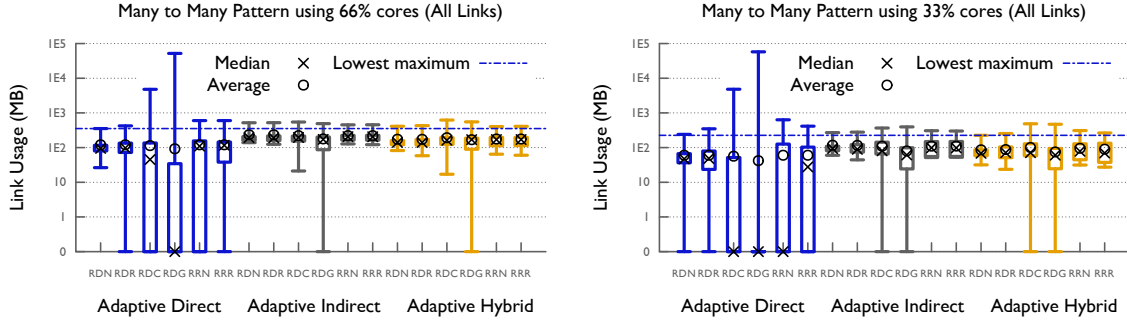


Figure 3.21: Traffic distribution for M2M on 66% and 33% cores.

Summary of full system predictions

Based on the analysis so far, we list the following summarizing points for single jobs executed on full systems:

- For patterns with many communicating nearby MPI processes, blocking may reduce the average and quartiles (UMesh).
- Direct routing may overload a few links, especially L2 links, if the communication is distribute evenly (4D Stencil, M2M).
- Randomized placement spreads traffic for patterns with non-uniform distribution of traffic (4D Stencil, M2M).
- Indirect routing is helpful in improving the distribution of traffic, but typically increases the average traffic (all patterns).
- If the communication pattern and job placement spreads the communication uniformly, indirect routing may increase the quartiles and the maximum traffic (M2M, Spread).
- Adaptive routing typically provides a similar traffic distribution, but may lower the maximum traffic significantly. Thus, in order to save space, we avoid showing results for static routings in the rest of the section.
- Adaptive hybrid provides a traffic distribution similar to AI, but may provide a higher or lower maximum traffic depending on the relative performance of AD and AI.

Variations in job size

We now present a case study in which one of the patterns, M2M, is executed in isolation on the full system, but occupies only a fraction of the cores. For comparison, we use M2M predictions on the full system from Figure 3.20 (top) and traffic distributions presented in Figure 3.21 for predictions using 66% and 33% of cores in isolation.

We observe very similar trends in traffic distribution across job placements and routings

as we move from predictions for 100% cores to predictions for 33% cores. As expected, the absolute values of most data points (maximum, average, quartiles) decrease steadily for the combinations that provide a good distribution. Direct routing with randomized placements consistently outperform indirect routings for critical data points including the maximum traffic.

Benefits of adaptive routing are significantly higher for job executions with smaller core counts. For the 100%, 66% and 33% cores executions, adaptive routing reduces the maximum traffic by up to 10.2%, 31.1% and 35% respectively. We attribute the increasing effect of the adaptivity to the non-uniformity that use of a fraction of cores in the system induces. Adaptive routing is able to observe these non-uniformities, and guides the traffic towards a better distribution.

Finally, we draw attention to the adaptive hybrid routing. For job placements that suit AD for this pattern (RDN and RDR), as we move from 100% to 33% cores, the critical data points (maximum, average, median) for AH are significantly lesser than those for AI. In fact, for the 33% cores case, the maximum traffic is least for AH among all the routings. This suggests that as non-uniformity in the system increases, AH is able to judiciously capitalize on good attributes of both AD and AI — use direct routes when they are not congested, else use indirect routes to offload traffic.

3.2.5 Predictions for parallel workloads

In this section, we focus on the more practical scenario in which multiple jobs with different patterns use the network simultaneously. Table 3.4 presents the representative workloads that we use for the experiments. These workloads represent capability jobs that use at least 10% of the total system size. For each workload, the system is divided among 5 single jobs that represent the following communication patterns: UMesh, 2D Stencil, 4D Stencil, M2M, and Spread. While four of these patterns are the ones described in Section 3.2.3, 2D Stencil is a new addition. It represents a two-dimensional stencil-based communication found in many applications such as WRF [93].

Comparing different parallel workloads

Figure 3.22 presents the predicted traffic distribution for workloads listed in Table 3.4. A common observation for all the workloads is the very high value for maximum traffic for AD with heavy blocking (RDC and RDG). Detailed histogram for the traffic on the links revealed that a few L2 links are heavily loaded. Initially, we suspected this to be caused by

Table 3.4: Percentage cores allocated to patterns in workloads.

Comm Pattern	Workload 1	Workload 2	Workload 3	Workload 4
UMesh	20	10	20	40
2D Stencil	10	10	40	10
4D Stencil	40	20	10	20
M2M	20	40	10	20
Spread	10	20	20	10

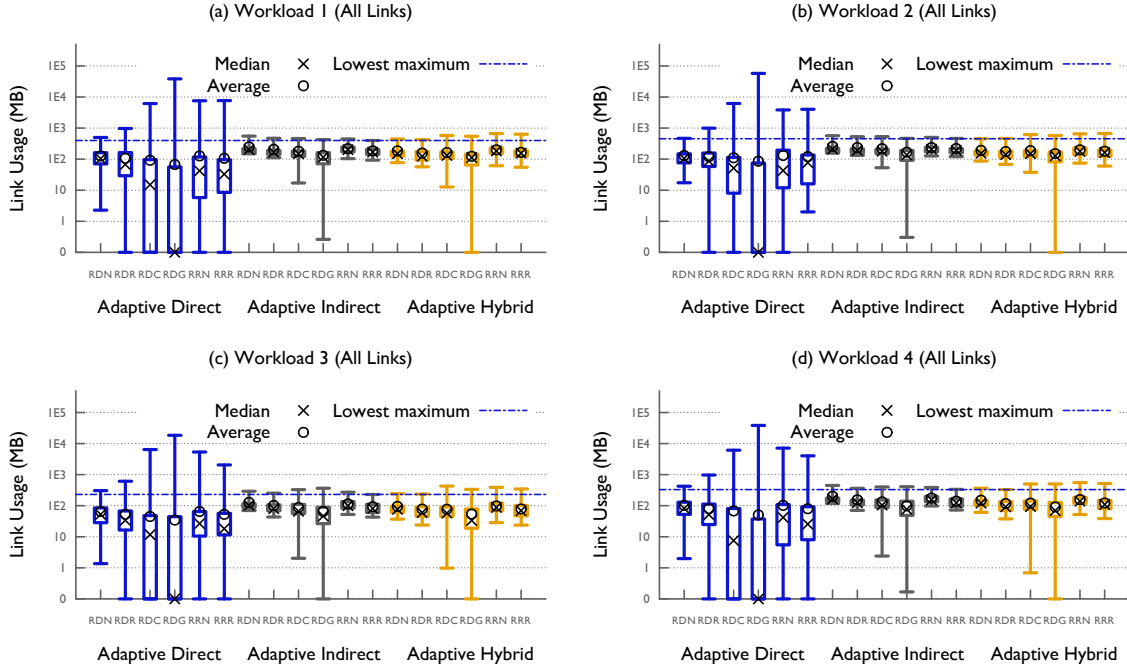


Figure 3.22: Parallel workloads traffic distribution.

overloading of a few L2 links by 4D Stencil in a similar manner as we saw in Section 3.2.4. In order to verify our assumption, we tried another workload with only four jobs: UMesh, Spread, M2M and 2D Stencil. However, for this fifth workload too, we observed similar overloading for AD with heavy blocking. Hence, we conclude that job placements with heavy blocking exposes any locality in communicating pairs of MPI ranks and leads to a few overloaded L2 links.

Figure 3.22 (a) presents the predicted traffic distribution for Workload 1, in which 40% of the cores are allocated to 4D Stencil; UMesh and M2M are assigned 20% cores each. For AD with blocked placement (RDC and RDG), we note that the average traffic is significantly higher than the median — a characteristic of 4D Stencil which occupies 40% of the cores in this workload. Use of randomized placement and indirect routing helps in reducing the skewness and maximum traffic. Among the combinations with similar distributions, the

maximum traffic is lowest for AI with RRR placement and AH with RDN/RDR placement. Adaptive routings reduce the maximum traffic by up to 35% in comparison to corresponding static routings.

In Workload 2, M2M is allocated the most number of cores (40%), while 4D Stencil and Spread are allocated 20% cores each. Other than the impact of locality in communicating pairs for AD with RDC and RDG described earlier, one can observe the impact of higher fraction of Spread and M2M in the closer values for average, median, and the quartiles. It also leads to AD with RRR and AH with RDN/RDR having the lowest value for the maximum traffic. Similar to Workload 1, adaptivity reduces the maximum traffic by up to 34.3%.

2D Stencil is assigned the largest number of cores (40%) in Workload 3, with UMesh and Spread being allocated 20% cores each. In 2D Stencil, four messages of size 64 KB are exchanged with its neighbors. For Workload 3, the traffic distribution shows mixed impact of Spread and 2D Stencil in Figure 3.22 (c). Contribution from Spread leads to a general increase in the maximum traffic for AI, while the gains obtained by randomized placements of 2D Stencil lower the maximum traffic for those combinations. Overall, the AH routing appears to take advantage of these effects and provides a nice distribution with the lowest value of maximum traffic for RDN and RDR. For Workload 4, predictions shown in Figure 3.22 (d) are very similar to Workload 3.

We make the following conclusions from these results: 1) Single capability jobs may have a significant impact on the traffic distribution of a workload, especially on its skewness as shown by the impact of 4D Stencil, 2) Similar traffic distributions are observed for workloads with the same set of jobs executing in different proportions, 3) The adaptive hybrid routing is able to combine positive features of AD and AI, thus providing a better traffic distribution.

Job-specific routing

Results presented in this section are for another interesting scenario in which each job in a workload is allowed to use a routing of its choice. This is currently not allowed on most systems but might become a useful option as system sizes increase further. We use Workload 2 and Workload 4 from Table 3.4 for these experiments. For each job, we select the routing that resulted in the lowest maximum traffic for a given job placement when the job was run by itself (Section 3.2.4).

Comparison of the traffic distribution for Workload 2, shown in Figure 3.23, with the results in Figure 3.22 (b) indicates that the distribution for job-specific routing is most similar to that of AH. However, for certain job placements, e.g. RDN and RDR, it has lower

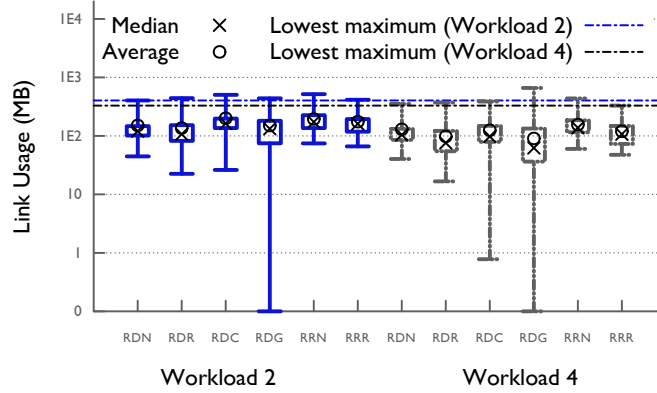


Figure 3.23: Job-specific routing traffic distribution (All Links).

values for minimum traffic and first quartiles — a characteristic shown by AD routing for Workload 2. This is not surprising because Workload 2 is dominated by M2M and Spread for which AD and AH were the best routings. An important observation to make is that the use of job-specific routing reduces the maximum traffic on any link for all job placements. Similarly, for Workload 4, the distribution of traffic for job-specific routing is similar to the load distribution for AI (Figure 3.22 (d)) which was the best performing routing for UMesh and 4D Stencil that dominate it. It also provides similar maximum traffic for best performing job placements.

3.2.6 Summary

We have presented a comparative analysis of various routing strategies and job placement policies w.r.t. network link throughput for the dragonfly topology. We have developed a congestion-aware model to determine the traffic distribution given a communication trace and a routing strategy. The output of this model is used to answer the questions we posed in the introduction. The answer to the first question is more nuanced than the other two because it depends heavily on the application communication patterns. The general observations are that a randomized placement at the granularity of nodes and routers and/or indirect routing can help spread the messaging traffic over the network and reduce hot-spots. If the communication pattern results in non-uniform distribution of traffic, adaptive routing may provide significantly better traffic distributions by reducing hot-spots.

For parallel job workloads (second question), adaptive hybrid routing is useful for combining good features of adaptive direct and adaptive indirect routings and may provide a good traffic distribution with lower maximum traffic. Adaptive routings also improve the

traffic distribution significantly in comparison to static routings. We also observed that allowing the users to choose a routing for their application can be beneficial in most cases on dragonfly networks (third question). Use of randomized placement at the granularity of nodes and routers is the suggested choice for such scenarios also. We believe that the model presented here will enable system administrators and end-users to try different scenarios w.r.t. optimizing network throughput for their use-cases.

Causes of Network Congestion ¹

In Chapter 3, we have seen that intelligent mapping of application tasks on nodes of a system can significantly improve the communication performance of the code. However, the process of finding the best mapping for an application may require executing a large number of runs at different scales (number of processors). This can consume a significant amount of resources including both man hours and machine allocation. Hence, it is desirable to predict the communication performance of an application on a system without performing real runs, given a communication graph and the mapping of tasks to nodes on the machine. However, in order to make accurate predictions, credible models or simulators are needed, which in turn require a detailed understanding of factors that impact communication performance.

Network congestion is widely recognized as one of the primary causes of performance degradation, performance variability, and poor scaling in communication-heavy applications running on supercomputers [31, 72, 95–97]. However, due to the complex nature of interconnection networks, as well as message injection and routing strategies, network congestion and its root causes in terms of network resources and hardware components are not well understood. Understanding network congestion requires a study of message flow on the network. When a message is sent from one node to another, it is split into packets that pass through many resources and hardware components on the network. A packet starts in an injection FIFO on the source. It then passes through multiple network links and receive buffers on intermediate nodes before it finally lands in the reception FIFO on the destination. When shared by multiple packets, any or all of these network components can slow down individual flits, packets and messages. This work aims to identify the hardware components that affect the performance of a message send the most.

Our approach is based on using supervised machine learning to build models that map from

¹Based on [82, 94]

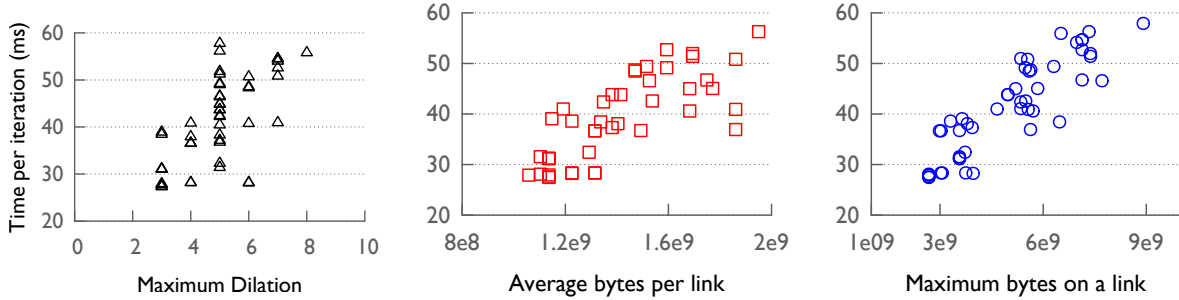


Figure 4.1: Performance variation with prior metrics for five-point halo exchange on 16,384 cores of Blue Gene/Q. Points represent observed performance with various task mappings. A large variation in performance is observed for the same value of the metric in all three cases.

independent variables, representing different network hardware components, to a dependent variable – the execution time of the application. We only consider computationally balanced, communication-heavy parallel applications and, hence, focus on the communication fraction of the total execution time. In order to generate multiple different executions of a parallel application, we vary the placement or layout of application processes/ tasks on the network. The different *task mappings* result in different message flows on the network and different execution times. This allows us to measure network hardware counters and execution times for the same executable under different configurations and network conditions.

Prior metrics

Traditionally, researchers have focused on the maximum dilation or average number of hops per byte for messages in an application as an indicator of its performance. These metrics make simplified assumptions about the cause of network congestion and do not provide accurate correlation with execution time. Mapping algorithms [27, 34, 98, 99] are generally designed to minimize these metrics which might be sub-optimal. We conducted a simple experiment with three of these metrics described in Section 2.4 – maximum dilation, average bytes-per-link and maximum bytes on a link to analyze their correlation with application performance. Figure 4.1 shows the communication time for one iteration of a two-dimensional halo exchange versus the three metrics in different plots. Each point in these plots is representative of a given task mapping on 16,384 cores of Blue Gene/Q. We observe that although the coefficient of determination values (R^2 , metric used for prediction success) are high, there is a significant variation in the y-values for different points with the same x-value. For example, in the maximum bytes plot (right), for $x = 6e9$, there are mappings with performance

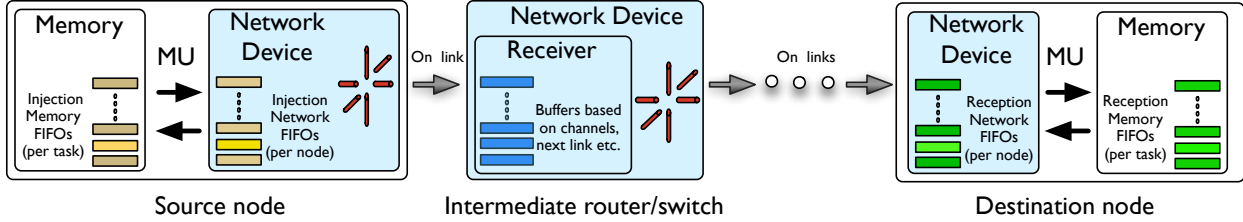


Figure 4.2: Message flow on Blue Gene/Q - a task initiates a message send by putting a descriptor in one of its memory injection FIFOs; the messaging unit (MU) processes these descriptors and injects packets into the injection network FIFOs from which the packets leave the node via links. On intermediate switches, the next link is decided based on the destination and the routing protocol; if the forward path is blocked, the message is stored in buffers. Finally on reaching the destination, packets are placed in network reception FIFOs from where the MU copies them to either the application memory or memory reception FIFOs.

varying from 20 to 50 ms. These variations make predicting performance using simple models with a reasonably high accuracy ($\pm 5\%$ error) difficult. This motivates us to find new metrics and ways to improve the correlation between metrics and application performance.

4.1 Contention on torus networks

Networks with n -dimensional torus topology are currently used in many supercomputers, such as IBM Blue Gene series, Cray’s XT/XE, and K computer. The IBM Blue Gene/Q (BG/Q) system is the third generation product in the Blue Gene line of massively parallel supercomputers. Each node on a BG/Q consists of 16 4-way SMT cores that run at 1.6 GHz. Nodes are connected by a five-dimensional (5D) torus interconnect with bidirectional links that can send and receive data at 2 GB/s. The BG/Q torus supports two shortest-path routing protocols – deterministic routing for short messages (<64 KB by default, configurable) and configurable dynamic routing for large messages.

4.1.1 Message flow and resource contention

Figure 4.2 presents the life cycle of a message on BG/Q. The tasks on a node send data on to the network through the Messaging Unit (MU) on the node. Injection **memory** FIFO (*imFifo*) is the data structure used to transfer information between the tasks and the MU. To initiate a message send, a task puts a descriptor of the message in one of its *imFifos*. Selection of which *imFifo* to inject a descriptor in is typically based on the difference in

coordinates of source and destination. The MU processes the descriptors in the *imFifos*, packetizes the message data it reads from the memory (packet size up to 512 B), and injects them into the injection **network** FIFOs. The descriptor of the message contains information of the binding that is used by the MU to inject into the appropriate injection network FIFO. In the default setting, there is a one-to-one mapping between *imFifos* and injection network FIFOs. This may lead to contention for injection network FIFOs if the distribution of source-destination pairs is such that a particular network injection FIFOs receives more traffic than others.

From the injection network FIFOs, packets are sent over the network based on the routing strategy and the destination. On the network, contention for hardware links is the most common source of performance degradation. When a packet injected on a link reaches an immediate neighbor, the network device decides the next link the packet needs to be forwarded to. If the next link is occupied, the packets are stored in buffers mapped to the incoming link. In the event of heavy contention for links, these buffers may get filled quickly, and prevent the use of the incoming link for data transfer. When packets eventually reach their destination, they are copied by the MU from the reception **network** FIFOs to either the reception **memory** FIFOs or the application memory. Limited memory bandwidth may prevent the MU from copying the data, and hence reception injection FIFOs and the buffers attached to the corresponding links may get filled. This may lead to blocking of the links for further communication.

4.1.2 Collecting hardware counters data

We use two methods to collect information that can indicate resource contention as described in Section 4.1.1:

Blue Gene/Q counters: The Hardware Performance Monitoring API (BGPM) provides a simple interface for the BG/Q Universal Performance Counter (UPC) hardware. The UPC hardware programs and counts performance events from multiple hardware units on a BG/Q node. Using BGPM to control the Network Unit of the UPC hardware, we implemented a PMPI-based profiling tool that records the following information:

- Sent chunks: count of 32-byte chunks sent on a link. Counters used for collecting this information:
PEVT_NW_USER_PP_SENT: user-level point-to-point 32-byte packet **chunks** sent (includes chunks in transit);
PEVT_NW_USER_ESC_PP_SENT: user-level **deterministic** point-to-point 32-byte packet

chunks sent (includes chunks in transit);

PEVT_NW_USER_DYN_PP_SENT: user-level **dynamic**

point-to-point 32-byte packet chunks sent (includes chunks in transit);

- Received packets: count of packets (up to 512 B) received on a link. Counter used for collecting this information:

PEVT_NW_USER_PP_RECV: user-level point-to-point **packets** received (includes packets in transit).

- Packets in buffers on incoming links: count of packets added across all buffers associated with an incoming link. Counter used for collecting this information:

PEVT_NW_USER_PP_RECV_FIFO: user-level point-to-point packets in buffers.

Analytical program: In order to derive information that is not available via counters, we implemented an analytical program to mimic important aspects of the BG/Q network, including the routing scheme and the injection network FIFO selection method. We use it to compute the following information:

- Dilation - number of hops (links) traversed by individual messages on the network.
- Messages in network FIFOs - number of messages injected in a particular injection **network** FIFO.

4.1.3 Indicators of resource contention

The information collected from hardware counters and the analytical program allows us to define several new metrics (Table 4.1). Bytes passing through links are used to compute average bytes and maximum bytes on links, which are indicators of link contention (these two are prior metrics). Buffer length, which increases as more packets get blocked during communication, is useful for measuring congestion on the intermediate switches. It may also indicate memory contention, since packets get buffered if available memory bandwidth to the MU is not sufficient to remove packets from the reception network FIFOs (at the destination). Ratio of the buffer length to the number of received packets indicates the average delay of packets passing through a link. FIFO length, which is also local to nodes, is an indicator of contention for injection network FIFOs, which may reduce the effective message rate.

The raw data we obtain for each execution is gathered *per link* in the network. To train our models, we require a single value for each feature aggregated over all the links. To achieve this, we use aggregates such as the average or maximum value of a feature over all

links. We also consider a smaller subset of links from the distribution, such as only those with a value greater than the mean (average outliers or AO), or those that are in the top 5% of the distribution (top outliers or TO). This helps us create several different aggregated features for each source or hardware component from which we obtained raw data. In the end, each execution (one sample) is represented by the nineteen features shown in Table 4.2 and a corresponding execution time.

4.2 Experimental setup

All the experiments were performed on two Blue Gene/Q systems – Mira and Vulcan. Mira is installed at the Argonne National Laboratory and Vulcan is hosted at the Lawrence Livermore National Laboratory. Table 4.3 presents the possible sizes of the torus dimensions (A, B, C, D and E) when partitions of 1024 and 4096 nodes are requested on BG/Q.

4.2.1 Communication kernels

We use three different communication kernels and two scalable, communication-heavy, production applications for the analysis in this work. A brief introduction to each is provided below:

Five-point 2D halo exchange: The *2D Halo* communication kernel uses a 2D grid of MPI processes to exchange four messages with two neighbors in each dimension.

15-point 3D halo exchange: The *3D Halo* communication kernel uses a 3D grid of MPI processes to exchange fourteen messages with its near-neighbors (six faces and eight corners).

All-to-all over sub-communicators: The *Sub A2A* communication kernel also uses a 3D process grid but performs all-to-alls on sub-communicators of size 64, formed from processes in one of the three dimensions.

MILC: MILC [19] is a Lattice Quantum Chromodynamics (QCD) application that does near-neighbor exchanges over a 4D process grid, similar to 2D and 3D Halo.

pF3D: pF3D [100] is a laser-plasma interaction code that performs all-to-alls over sub-communicators (similar to Sub A2A) and near-neighbor exchanges over a 3D process grid.

The communication kernels are executed with different message sizes – 8 B, 512 B, 16 KB and 4 MB to evaluate different MPI performance regimes. The computational load of both

Indicator	Source	Derived from
Bytes on links*	Counters	Sent chunks
Stalls or Buffer length [†]	Counters	#Packets in buffers
Stalls per packet or Delay per link [†]	Counters	#Packets in buffers divided by received packets
Dilation*	Analytical	Shortest-path routing between source and destination
FIFO length [†]	Analytical	Based on PAMI source

Table 4.1: *Prior and [†]new metrics that indicate contention for network resources.




















Feature name	Description
 avg dilation AO	Avg. dilation of average outliers (AO)
 max dilation	Maximum dilation
 sum dilation AO	Sum of dilation of AO
 avg bytes	Avg. bytes per link
 avg bytes AO	Avg. bytes per link for AO
 avg bytes TO	Avg. bytes per link for top outliers (TO)
 max bytes	Maximum bytes on a link
 #links AO bytes	No. of AO links w.r.t. bytes
 avg stalls	Avg. receive buffer length
 avg stalls AO	Avg. receive buffer length for AO
 avg stalls TO	Avg. receive buffer length for TO
 max stalls	Maximum receive buffer length
 #links AO stalls	No. of AO links w.r.t. rcv buffer length
 avg stallsp	Avg. number of stalls per rcv'd packet
 avg stallsp AO	Avg. no. of stalls per packet for AO
 avg stallsp TO	Avg. no. of stalls per packet for TO
 max stallsp	Maximum number of stalls per packet
 #links AO stallsp	No. of AO links w.r.t. stalls per packet
 max inj FIFO	Maximum injection FIFO length

Table 4.2: List of communication metrics (features) used as inputs to the machine learning model. The colors in this table correspond to different hardware components in Table 4.1

Nodes	A	B	C	D	E
1024	4	4	4	8	2
4096	4	4	8	16	2
4096	4	8	4	16	2

Table 4.3: Dimensions of the allocated job partitions on BG/Q.

#Nodes	2D Halo		3D Halo		Sub A2A		MILC	pF3D	Total
	16 KB	4 MB	16 KB	4 MB	16 KB	4 MB			
1024	84	84	84	84	84	84	208	94	806
4096	84	84	84	84	84	84	103	103	710
Total	168	168	168	168	168	168	311	197	1516

Table 4.4: Sizes of the input datasets in terms of the number of executions or samples for the different codes.

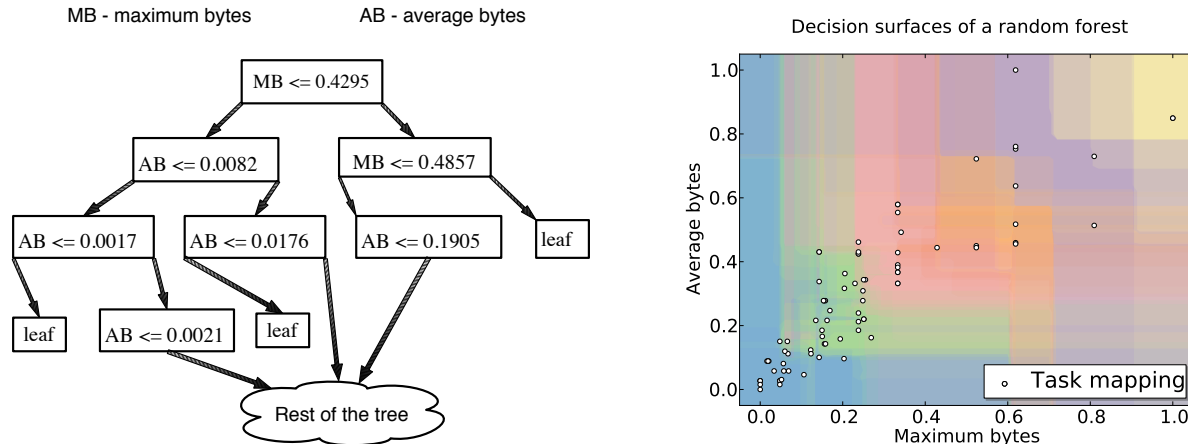
MILC and pF3D is almost perfectly balanced across MPI processes. This allows us to focus on their communication, which is a significant portion of their overall execution time. We ran all the codes on 1024 and 4096 nodes of Blue Gene/Q to study the congestion behavior on different torus sizes. Depending on the code, we placed between 16 and 64 processes per node.

Table 4.4 lists the number of task mappings that were generated for each kernel or application at each node count. For example, for 2D Halo, we created 84 different task mappings and ran them for the two message sizes – 16 KB and 4 MB (168 in total). Similar number of runs were performed for small message sizes also.

4.2.2 Prediction using ensemble methods

We employ supervised learning techniques used in statistics and machine learning to predict the performance (execution time) of an application for different mappings using metrics described in Section 4.1.3. The learning algorithm infers a model or function by working with a **training** set that consists of n *samples* (mappings), and one or more input *features* (raw and/or derived such as average bytes) per sample. Each sample has a desired output value (execution time), also known as the *target*. We then use the trained algorithm to predict the output for a **testing** set (new mappings for which we wish to predict execution time). The values of the features and the target are normalized to obtain the best results.

We tested several supervised learning techniques ranging from statistical techniques, such as linear regression, to machine learning methods, such as support vector machines and decision trees. The *scikit-learn* package provides several of these algorithms/estimators in Python [101]. The two steps, as described previously, are to first *fit* the estimator and then to use it to *predict* unseen samples. For the benchmarks presented in this chapter, ensemble learning provided the best fit. Ensemble methods use predictions from several, often weaker models, to produce a stronger model or estimator that gives better results than the individual



(a) Decision tree. Based on the training set and the learning scheme, conditions are computed to guide prediction based on features, e.g., maximum bytes and average bytes. To predict, beginning at the root, the tree is traversed based on the features of a test case until a leaf is reached. The leaf determines the predicted value.

(b) Random forests. A collection of decision trees is used to predict. Each color represent a leaf region in one of the decision trees; regions from different decision trees overlap. For a test case, all decisions trees are traversed to obtain local predictions, which are combined using weights to obtain the final prediction.

Figure 4.3: Example decision tree and random forests generated using scikit.

models.

Ensemble methods: The primary reasons for considering ensemble models are: (1) *Statistical*: different predictive models may perform similarly on the training data, when learned from a limited number of training samples. However, the performance of each of these models with test data can be poor. By averaging representations obtained from an ensemble, we may obtain an approximation closer to the true test data; (2) *Computational*: even with large training sets, the modeling technique might not reach the global optimum and using an ensemble of multiple locally optimal models can result in improved performance; (3) *Representational*: the hypothesis space assumed for learning the model cannot represent the test data, and this can happen when the data is corrupted.

Random forests are a type of ensemble learning method developed by Leo Breiman in 2001 [102]. The idea is to build several decision trees and to use averaging methods on these independently built models. In a decision tree, the goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. These trees are simple to understand and to interpret as they can be visualized easily. However, decision tree based learners create biased trees if some patterns dominate. Random forests attempts to avoid such a bias by adding randomness in the selection of

condition when splitting a node during the creation of a decision tree. Instead of choosing the best split among all the features, the split that is picked is one among a random subset of the features. Further, the averaging performed on the prediction from independently built decision trees leads to a reduction in the variance and hence an overall better model. We use the `ExtraTreesRegressor` class in scikit 0.13.1.

Figure 4.3 (left) shows an example decision tree that was used in one of our experiments. This tree was produced by training performed using two features, maximum bytes and average bytes, to predict the target, the execution time. The tree shows that at each level, based on the conditions derived from the training set on values of maximum bytes and average bytes, the prediction is guided to reach a leaf node, which determines the target value. Figure 4.3 (right) presents the overlay of a number of such decision trees over a 2D space spanned by the same two input features. Each color in the figure is a leaf region of one of the decision trees in the random forest generated by fitting the training set. As expected, regions from different decision trees overlap, and cover the entire space. The white circles are the test set being predicted - new mappings with known maximum bytes and average bytes. To predict, all the leaf regions, to which a test case belong, are computed. This provides a set of local predictions for the test case, which are combined to provide a unique target value.

Gradient boosted regression trees: The main idea of boosting is to add a new weak, base-learner model in each iteration, which is trained with respect to the error of the whole ensemble inferred so far. In gradient boosted regression trees (GBRT) [103], the new base-learners are designed to be maximally correlated with the negative gradient of the loss function associated with the whole ensemble. This technique is flexible enough to be used with different families of loss functions, and this choice is often influenced by the desired characteristics of the conditional distribution, such as robustness to outliers. Any arbitrary loss function can be plugged into the framework by specifying the loss function and the function to compute its negative gradient [104]. The squared ℓ_2 loss and the Laplacian ℓ_1 loss are common choices for regression tasks and these functions penalize large deviations from the target outputs, while ignoring smaller residuals. In addition, parameterized loss functions, such as Huber, can be adopted for robust regression. Given the input variable x , the target output y and the regression function f , the Huber loss is defined as

$$\Psi^H(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & |y - f(x)| \leq \delta, \\ \delta(|y - f(x)| - \delta/2) & |y - f(x)| > \delta. \end{cases} \quad (4.1)$$

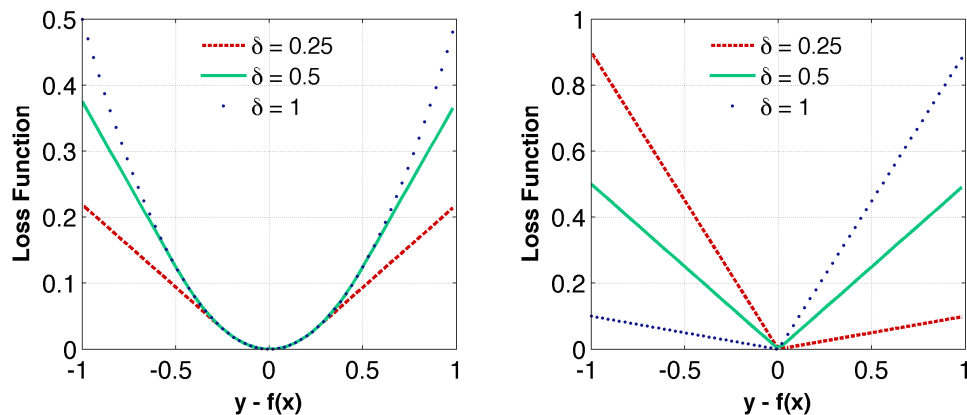


Figure 4.4: Parameterized loss functions for gradient tree boosting: Huber loss function with the cutting-edge parameter δ (left), quantile loss function (right).

As it can be observed, Huber loss combines ℓ_1 and ℓ_2 functions. The parameter δ is the cutting-edge parameter, and this specifies the maximum value of error beyond which the ℓ_1 function is applied. Alternatively, we can predict a conditional quantile of the target variable for robust regression. This can be achieved by considering the asymmetric quantile loss function:

$$\Psi^Q(y, f(x)) = \begin{cases} (1 - \alpha)|y - f(x)| & y - f(x) \leq 0, \\ \alpha|y - f(x)| & y - f(x) > 0. \end{cases} \quad (4.2)$$

The parameter α specifies the desired quantile of the conditional distribution. When $\alpha = 0.5$, the quantile loss function corresponds to the ℓ_1 loss. Figure 4.4 illustrates the Huber and quantile loss functions at different parameter values.

The following are the steps we follow for learning a model and predicting execution time for an individual dataset:

- Scale each feature in the dataset to have values between 0 and 1 based on the minimum and maximum values for that feature across all samples.
- Divide the n samples in the dataset into a training set and a testing set, roughly in a two-thirds and one-third split.
- Generate all possible combinations of the nineteen features that we would like to learn a model with.
- Do a parallel run where each process runs the ExtraTreesRegressor or the GBRT regressor on a subset of feature combinations and reports the prediction scores using the generated models.

- Based on the prediction scores, we pick the feature combinations that lead to the highest scores.

Metrics for prediction success

The goodness or success of the prediction function (also referred to as the *score*) can be evaluated using different metrics depending on the definition of success. Our main goal is to compare the performance of two mappings and determine the correct ordering between the mappings in terms of performance. Hence, we focus on a rank correlation metric for determining success; we also present results for a metric that compares absolute values for completeness.

Rank Correlation Coefficient (RCC): Let us assign ranks to mappings based on their position in two sorted sets (by execution time): observed and predicted performance. RCC is defined as the ratio of the number of pairs of task mappings whose ranks were in the same pairwise order in both the sets to the total number of pairs. In statistical parlance, RCC equals the ratio of the number of concordant pairs to that of all pairs (Kendall's Tau [105]). Formally speaking, if observed ranks of tasks mappings are given by $\{x_1, x_2, \dots, x_n\}$, and the predicted ranks by $\{y_1, y_2, \dots, y_n\}$, we define RCC as:

$$concord_{ij} = \begin{cases} 1, & \text{if } x_i \geq x_j \ \& \ y_i \geq y_j \\ 1, & \text{if } x_i < x_j \ \& \ y_i < y_j \\ 0, & \text{otherwise} \end{cases}$$

$$RCC = \left(\sum_{0 \leq i < n} \sum_{0 \leq j < i} concord_{ij} \right) / \left(\frac{n(n-1)}{2} \right)$$

Absolute Correlation (R^2): To predict the success for absolute predicted values, we use the coefficient of determination from statistics, **R-squared**,

$$R^2(y, \hat{y}) = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$$

where \hat{y}_i is the predicted value of the i^{th} sample, y_i is the corresponding true value, and

$$\bar{y} = \frac{1}{n_{samples}} \sum_i y_i$$

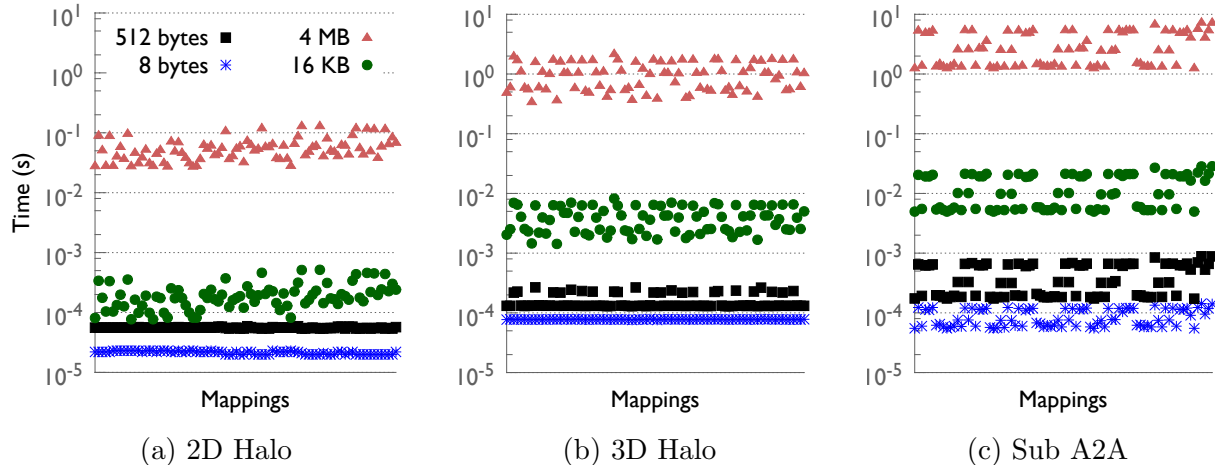


Figure 4.5: Performance variations with different task mappings on 16,384 cores of BG/Q. As benchmarks become more communication intensive, even for small message sizes, mapping impacts performance.

4.3 Performance prediction of communication kernels

In this section, we present results on the prediction of execution times of several communication kernels (Section 4.2.1) for different task mappings.

4.3.1 Performance variation with mapping

Figure 4.5 presents the execution times for the three benchmarks for four message sizes – 8 bytes, 512 bytes, 16 KB and 4 MB. These sizes represent the amount of data exchanged between a pair of MPI processes in each iteration. For example, for 2D Halo, this number is the size of a message sent by an MPI process to each of its four neighbors. For a particular message size, a point on the plot represents the execution time (on the y-axis) for a mapping (on the x-axis).

For 2D Halo, Figure 4.5a shows that for small messages such as 8 and 512 bytes, mapping has an insignificant impact. As the message size is increased to 16 KB, in addition to an increase in the runtime, we observe up to a $7\times$ difference in performance for the best mapping in comparison to the worst mapping (note the logarithmic scale on the y-axis). Similar variation is seen as we further increase the message size to 4 MB. For a more communication intensive benchmark, 3D Halo, we find that mapping impacts performance even for 512-byte messages (Figure 4.5b). As we further increase the communication in Sub A2A, the effect of task mapping is also seen for the 8-byte messages as shown in Figure 4.5c. In the following sections, we do not present results for the cases where the performance variations

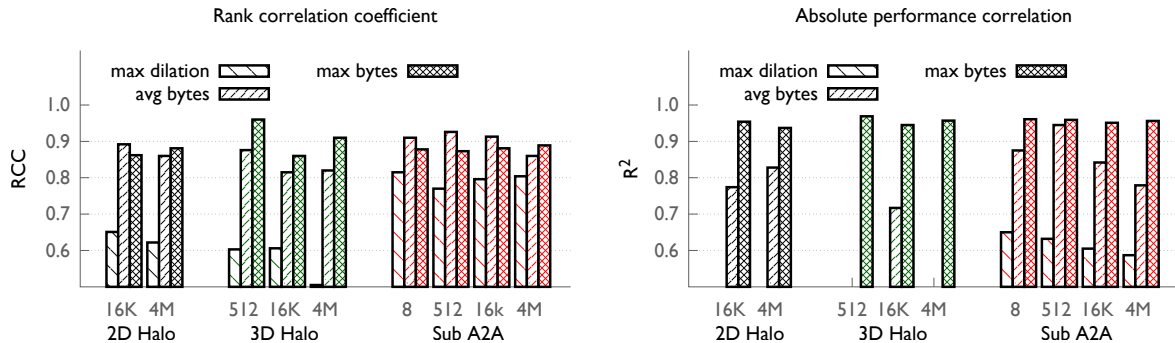


Figure 4.6: Prediction success based on prior features on 16,384 cores of BG/Q. The best RCC score is 0.91 for most cases - 38 mispredictions out of 378.

from mapping are statistically insignificant: 8- and 512-byte results in case of 2D Halo and 8-byte results in case of 3D Halo.

4.3.2 Prior features

We begin with showing prediction results using prior metrics/features and quantify the goodness of the fit or prediction using rank correlation coefficient (RCC) and R^2 . Figure 4.6 (top) presents the RCC values for predictions based on prior features (maximum dilation, average bytes per link and maximum bytes on a link). In most cases, we find that the highest value for RCC is 0.91, i.e., the pairwise ordering of 91% of mapping pairs was predicted correctly. For a testing set of 28 samples, an RCC of 0.91 implies incorrect prediction of the pairwise ordering of 38 mapping pairs. A notable exception is the 512-byte case for 3D Halo where the RCC is 0.96. In contrast, for 16 KB message size, the highest RCC is only 0.86.

In the case of 2D Halo and 3D Halo, prediction using maximum bytes on a link has the highest RCC while prediction using maximum dilation performs very poorly with an RCC close to 0.60. However, for Sub A2A, prediction using average bytes per link is better than prediction using maximum bytes on a link for small to medium message sizes (by 4-5%). The metric for absolute performance correlation, R^2 , is also shown in Figure 4.6. For all benchmarks and message sizes, maximum bytes on a link performs the best with a score of up to 0.95 for 3D Halo and Sub A2A. These results substantiate the use of maximum bytes and average bytes as simple metrics that are roughly correlated with performance.

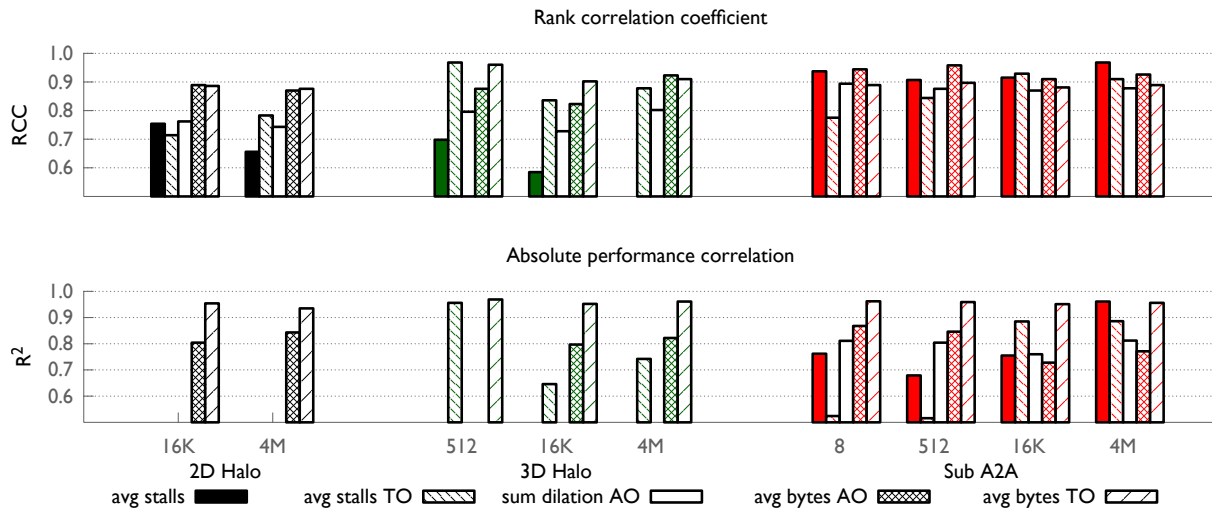


Figure 4.7: Prediction success based on new features on 16,384 cores of BG/Q. We observe a general increase in RCC, but R^2 values are low in most cases resulting in empty columns.

4.3.3 New features

We propose new metrics/features based on the buffer length, delay and FIFO lengths (see Table 4.1) and derive others by extracting counters and analytical data for outlier nodes and links:

Average Outliers (AO) We define a node or link as an *average outlier* if an associated value for it is greater than the average value of the entire data set. Selection of data points based on the average value helps eliminate low values that can skew derived features and hide information that may be useful.

Top Outliers (TO) Similar to the average outlier, we can define a node or link to be a *top outlier* if an associated value for it is within 5% of the maximum value across the entire data set.

We can use these two outlier selection criteria to define metrics that represent the features extracted from outliers. Among a large set of features that we explored using prior/new metrics in combination with known/new derivation methods, we focus on prediction using the following features that had the highest RCC: average buffer length (*avg stalls*), average buffer length of TO (*avg stalls TO*), sum of maximum dilation for AO (*sum dilation AO*), average bytes per link for AO (*avg bytes AO*), and the average bytes per link for TO (*avg bytes TO*).

The most important point to note as we transition from Figure 4.6 to Figure 4.7 is the

general increase in RCC. For Sub A2A in particular, we observe that RCC is consistently 0.95. The previous poor predictions in the case of 16 KB message size for 3D Halo improve from 0.86 to 0.90 (RCC value). For the low traffic 2D Halo, new network-related features such as those based on the buffer length exhibit low correlation. As traffic on the network is increased (larger messages sizes) in 3D Halo and Sub A2A, the RCC of these new network-related features increases, and occasionally surpasses the RCC of other features.

We note that the R^2 value is consistently high only for the *avg bytes TO* feature. For a number of features, the R^2 values are either low or zero. There are two reasons that can explain the low R^2 values: 1) the features did not correlate well with the performance, e.g. *avg stalls* for 2D Halo and 3D Halo, or 2) the predicted performance followed the same trend as the observed performance but was offset by a factor, e.g., *avg stalls* for large messages in 3D Halo.

4.3.4 Hybrid features

The previous sections have shown that up to 94% prediction accuracy can be achieved using a single feature. Depending on the benchmark and the message size, the feature that provides the best prediction may vary. This motivated us to use several features together to improve the correlation and enable better prediction.

In order to derive hybrid features that improve RCC, we performed a comprehensive search by combining the features that had high RCC values. In addition, the combinations of *good* features were also augmented with features that exhibited low to moderate correlation with performance. We made two important discoveries with these experiments: 1) combining multiple *good* features may result in a lower accuracy of prediction, and 2) the addition of features that had limited success on their own to good features can boost prediction accuracy significantly.

Hybrid	Features combined
H1	avg bytes, max bytes, max FIFO
H2	avg bytes, max bytes, sum dilation AO, max FIFO
H3	avg bytes, max bytes, avg stalls, max FIFO
H4	avg bytes, max bytes, avg stalls TO
H5	avg bytes TO, avg stalls TO, avg stallspp AO, sum hops AO, max FIFO
H6	avg bytes TO, avg stalls AO, avg stallspp TO, avg stallspp AO, sum hops A0, max FIFO

Table 4.5: List of hybrid features that achieve strong correlations.

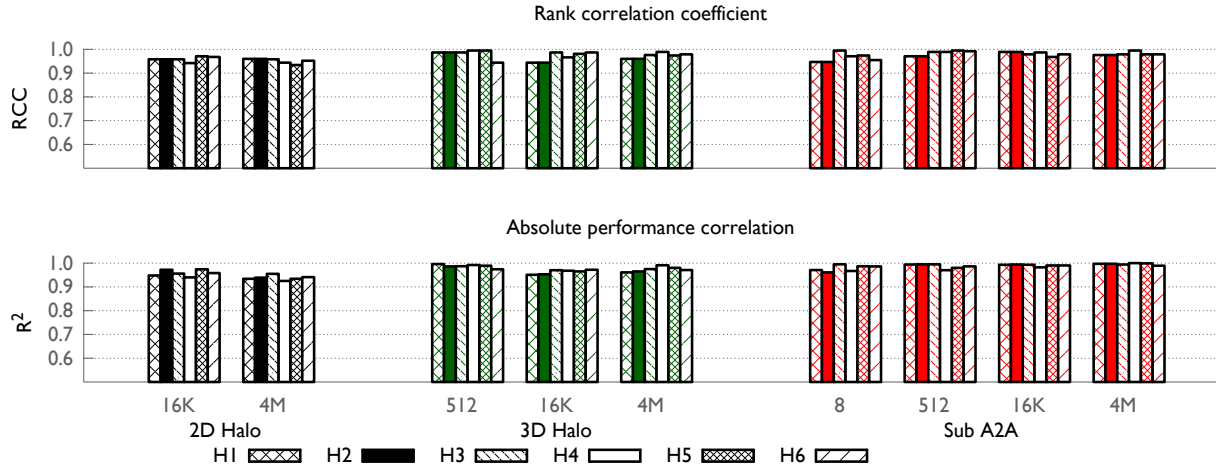


Figure 4.8: Prediction success based on hybrid features from Table 4.5 on 16,384 cores of BG/Q. We obtain RCC and R^2 values exceeding 0.99 for 3D Halo and Sub A2A. Prediction success improves significantly for 2D Halo also.

Figure 4.8 presents results for the hybrid features that consistently provided high prediction success with different benchmarks and message sizes. Table 4.5 lists the features that were combined to create these hybrid features. In our experiments, we found that combining the two most commonly used features, *max bytes* and *avg bytes* improves the prediction accuracy in all cases. The gain in RCC was highest for the 4 MB message size, where RCC increased from 0.91 (individual best) to 0.94 for all benchmarks. The addition of *max FIFO*, which did not show a high RCC score as a stand alone feature, further increased the prediction accuracy to 0.96. We denote this set as H1.

To H1, we added another low performing feature, *avg stalls* to obtain H3. This improved the RCC further in all cases with the RCC score now in the range of 0.98 – 0.99 for 3D Halo and Sub A2A (Figure 4.8). Replacing *avg stalls* in this set with *avg stalls TO* improved the RCC for Sub A2A, but reduced the RCC for 2D Halo. Using a number of such combinations, we consistently achieved RCC up to 0.995 for Sub A2A. Given a testing set of size 28, this implies that the pairwise order of only 2 pairs was mispredicted for Sub A2A; in the worst case for 2D Halo, pairwise order of 16 pairs was mispredicted.

Prediction using hybrid features also results in high R^2 values as shown in Figure 4.8. For 2D Halo, the scores go up from 0.95 and 0.93 to 0.975 and 0.955 for the 16 KB and 4 MB message sizes respectively. For the more communication intensive benchmarks, we obtained R^2 values as high as 0.99 in general. Hence, the use of hybrid features not only predicts the correct pairwise ordering of mapping pairs but also does so with high accuracy in predicting their absolute performance.

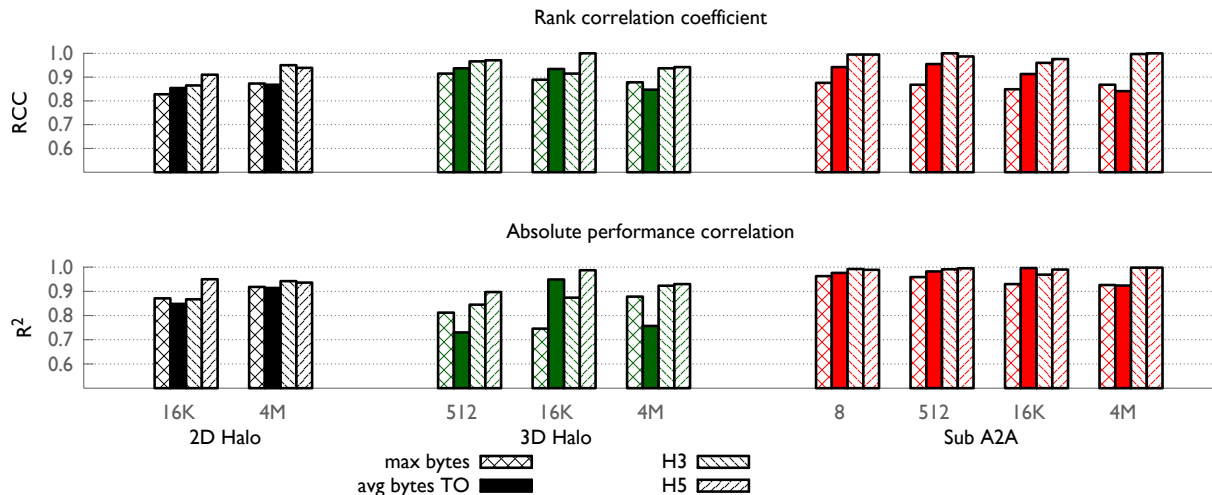


Figure 4.9: Prediction success: summary for all benchmarks on 65,536 cores of BG/Q. Hybrid metrics show high correlation with application performance.

4.3.5 Results on 65,536 cores

Figure 4.9 shows the prediction success for the three benchmarks on 65,536 cores of BG/Q. From all the previously presented features (prior, new and hybrid), we selected the ones with the highest RCC scores for 16,384 cores, and present only those in this figure. Similar to results on 16,384 cores, we obtain significant improvements in the prediction scores using hybrid features in comparison to individual features such as *max bytes* and *avg bytes TO*. For Sub A2A, RCC improved by 14% from 0.86 to 0.98, with a RCC value of 1.00 for both 512 bytes and 4 MB message sizes. For 2D Halo and 3D Halo, an improvement of up to 8% was observed in the prediction success. Similar trends were observed for R^2 values.

Figure 4.10 presents the scatter-plot of predicted performance for the three benchmarks for the 4 MB message size. On the x-axis are the task mappings sorted by observed performance, while the y-axis is the predicted performance. The feature set *H5: avg bytes TO, avg stalls TO, avg stallspp AO, sum hops AO, max FIFO* was used for these predictions. It is evident from the figure that an almost perfect performance based ordering can be achieved using prediction for all three benchmarks, as expected based on the high RCC values. In addition, as shown in Figure 4.10, absolute performance values can also be predicted accurately using the proposed hybrid metric. In particular, for Sub A2A with large communication volume, the predicted value curve completely overlaps with the observed value curve. These results suggest that the same set of features correlate with the performance irrespective of the system size being used.

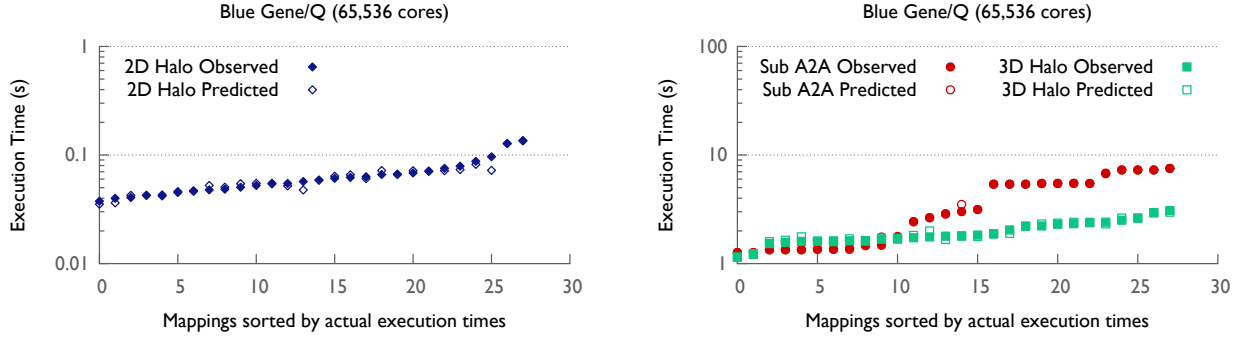


Figure 4.10: Summary of prediction results on 65,536 cores using 4 MB messages. For all benchmarks, prediction is highly accurate both in terms of ordering and absolute values.

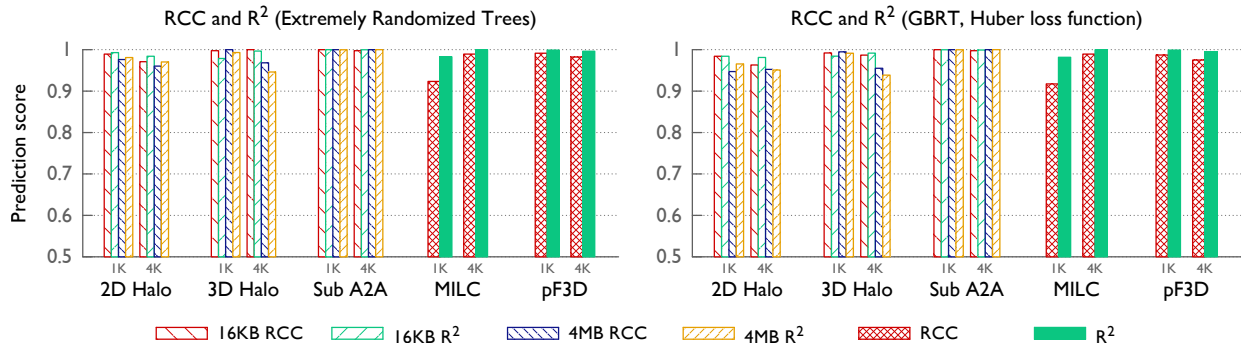


Figure 4.11: Highest prediction scores obtained for the individual datasets using Extremely Randomized Trees (left) and Gradient Boosted Regression Tree (right). Adjoining pairs of vertical bars represent the RCC and R^2 values for each of the sixteen datasets.

4.4 GBRT and production applications

Figure 4.11 shows the highest prediction scores (both RCC and R^2) obtained for any feature combination for each of the datasets. Adjoining pairs of vertical bars represent the RCC and R^2 values for each of the sixteen datasets. The left plot illustrates results obtained using the extremely randomized trees algorithm, and the right plot shows similar results using GBRT with the Huber loss function. Though either of the methods can be used for subsequent analysis, we choose GBRT for the results in the rest of this chapter because of its flexibility in allowing parameterized loss functions.

As we discussed in the previous sections, on an average, the prediction scores are very high. When predicting the execution time of 2D and 3D Halo, we obtain RCCs in the range 0.95–1.0 and R^2 in the range 0.94–0.996. As we increase the amount of communication being performed (from 2D Halo to 3D Halo to Sub A2A), the predictions become stronger. For Sub A2A, the RCC and R^2 values are between 0.997 and 1.0. This is not unexpected – the more

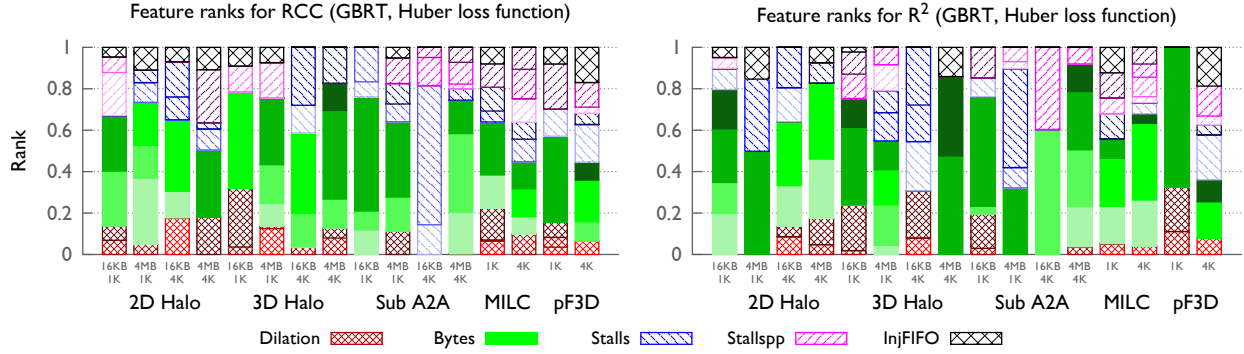


Figure 4.12: Ranks of different features in the models that yield the highest RCC (left plot) and R^2 scores (right plot) for individual datasets using Gradient Tree Boosting (loss function = ‘Huber’). Each stacked bar represents the ranks of the nineteen features (colored by categories) for one of the sixteen datasets.

a parallel code stresses the network, higher is the correlation between the communication features that represent congestion and execution time.

For production applications, which have more complex communication patterns, we observe very high prediction scores. MILC, which performs a 4D halo, is communication-heavy and task mapping sensitive. Other than the RCC scores on 1K nodes, the prediction scores for MILC are very high (R^2 between 0.98 and 0.997). pF3D has communication patterns similar to Sub A2A along with a near-neighbor communication, which results in high RCC values between 0.975 and 0.991. This can be attributed to the structured and communication-intensive all-to-all operations whose execution time is heavily dependent on network congestion. The prediction scores for pF3D are also good. On 1K nodes, the R^2 values are close to 0.995, while on 4K nodes, both RCC scores R^2 scores are in the range 0.975–0.996.

As we compare the prediction quality of the supervised learning models for different codes, a natural question that comes up is – which features are important in predicting the execution time for different kernels and applications? Figure 4.12 presents the relative importance or ranks of different features in the models that yield the highest RCC (left plot) and R^2 values (right plot). Each stacked bar represents the ranks of the nineteen features (colored by categories) for one of the sixteen datasets. As we can see, the relative importance of features changes depending on the code and on whether RCC or R^2 is more important. The only conclusive observation that can be drawn from these plots is that the number of bytes flowing over the network has a significant impact on execution time, which is to be expected. Ideally, we would like to identify a smaller subset of features that can predict the execution time well for a range of applications, message sizes and node counts. We discuss this in detail in

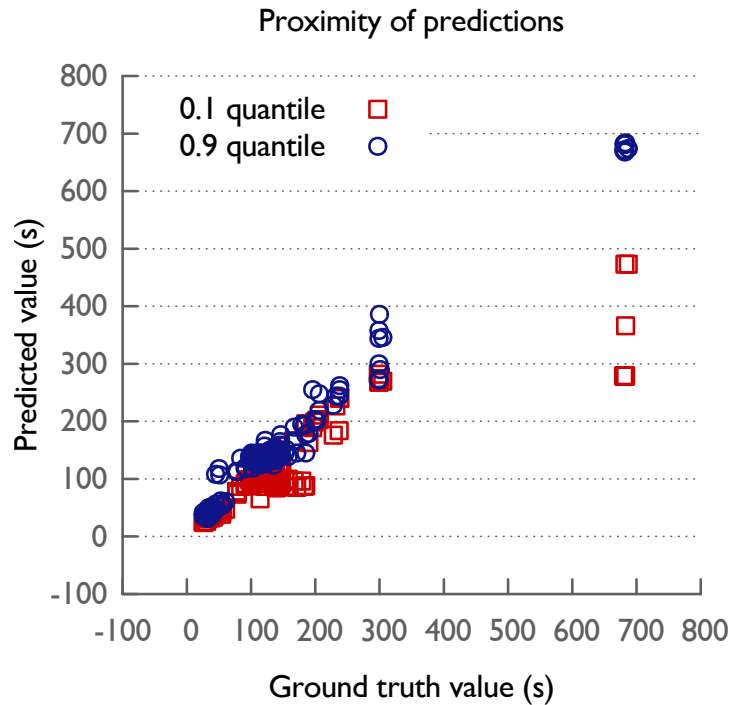


Figure 4.13: GBRT regression on the Apps dataset using different quantile loss functions. The lower quantile regression function underpredicts for those samples with high execution time, while predicting effectively for those with low execution times.

the next section.

4.5 Identifying relevant feature subsets

The variability in the importance (rank) of different features in the regression models learned for different parallel codes makes it challenging to identify a common set of factors that contribute the most to network congestion. Furthermore, some of the features considered in our analysis might be strongly correlated to one another, thereby introducing instabilities in the model selection process across multiple datasets. In order to overcome these challenges, we propose to infer regression models under different quantiles, and analyze them to identify the most relevant features in a stable manner (irrespective of our choice of training sets).

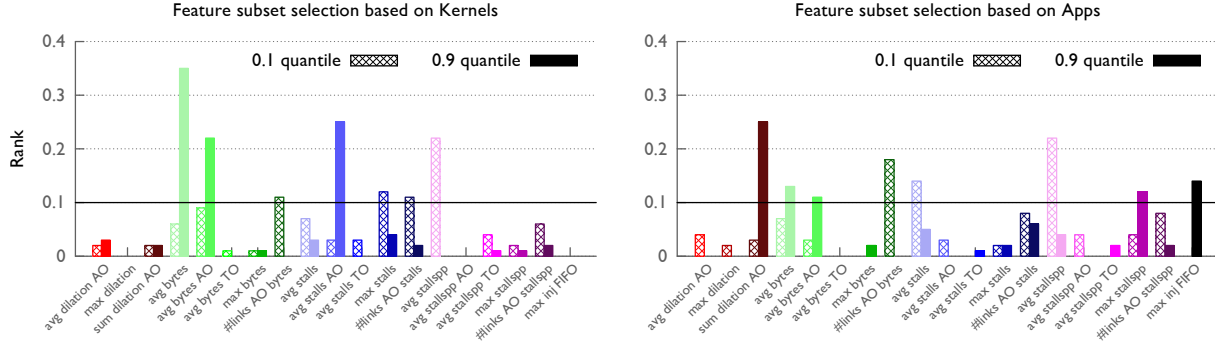


Figure 4.14: Ranks of different features obtained using GBRT with quantile loss functions at $\alpha = 0.1$ and $\alpha = 0.9$ respectively: left plot is for a combined set of the three communication kernels (twelve datasets) and the right plot is for a combined set of the two applications (four datasets).

4.5.1 Feature selection from extreme quantiles

For the analysis presented in this section, we use GBRT with the quantile loss function defined in equation (4.2) in Section 4.2.2. In order to identify the most relevant features for predicting the execution time, we propose to analyze the regression models at lower ($\alpha = 0.1$) and higher ($\alpha = 0.9$) conditional quantiles. In particular, we consider the ranks of the different features at the extreme quantiles. Instead of inferring a single regression function that minimizes the average or median error for all data samples, the quantile loss weights different regions in the function space asymmetrically (see Figure 4.4). For example, in Figure 4.13, the lower quantile model provides an accurate prediction for samples with low execution times (bottom left corner), while making large errors on samples with high execution times.

It turns out that for the datasets used in this chapter, optimizing for the conditional quantiles inherently promotes sparsity in the inferred model (Figure 4.14). This means that only a few features show significant importance for prediction, and the ranks for different features in the lower and higher quantiles case vary considerably. This also results in different features being more important for the two quantiles. Figure 4.14 shows the feature importance for the extreme quantiles for all the kernel datasets combined together (left plot) and all the application datasets combined together (right plot). In the left plot, we see that the feature *avg bytes* has a high rank in predicting at the higher quantiles only. On the other hand, the feature *avg stallpp* is prominent in predicting at the lower quantiles and not used by the regression function optimized for the higher quantile. We can observe similar things about *sum dilation AO* and *max inj FIFO* in the right plot.

We exploit these observations by selecting the most relevant features from the models at different quantiles, and using this subset of features to predict the execution time for different applications. The steps involved in this proposed technique for feature selection for a dataset are as follows:

- Create random splits of the dataset into training and testing sets (70% for training and the rest for testing). We repeat this over 50 iterations to avoid overfitting.
- Learn regression models using GBRT with quantile loss functions at $\alpha = 0.1$ and $\alpha = 0.9$. We denote the feature ranks in the two cases by $\tau_{0.1}$ and $\tau_{0.9}$ respectively.
- Compute the average feature ranks for the extreme quantiles from the 50 iterations.
- Identify the relevant features as those with either $\tau_{0.1}$ or $\tau_{0.9}$ greater than a pre-defined threshold t . In our experiments, we fixed t at 0.1.

4.5.2 Results and discussion

We employ the proposed feature selection technique explained above on the following larger datasets formed by combining the individual datasets in Table 4.4:

1. 2D Halo (4 datasets)
2. 3D Halo (4 datasets)
3. Sub A2A (4 datasets)
4. Kernels (combination of (a), (b), and (c), 12 datasets)
5. MILC (2 datasets)
6. pF3D (2 datasets)
7. Apps (combination of (e) and (f), 4 datasets)
8. All (all 16 datasets added together)

The goal is to identify a common set of features that might be relevant across multiple datasets. Figure 4.15 presents the feature ranks obtained using the technique described above for each of the larger datasets. Note that the importance/rank of each feature is obtained by first identifying the smallest subset for each dataset and then performing another cycle of training and testing to obtain the relative importance of the features in this new subset.

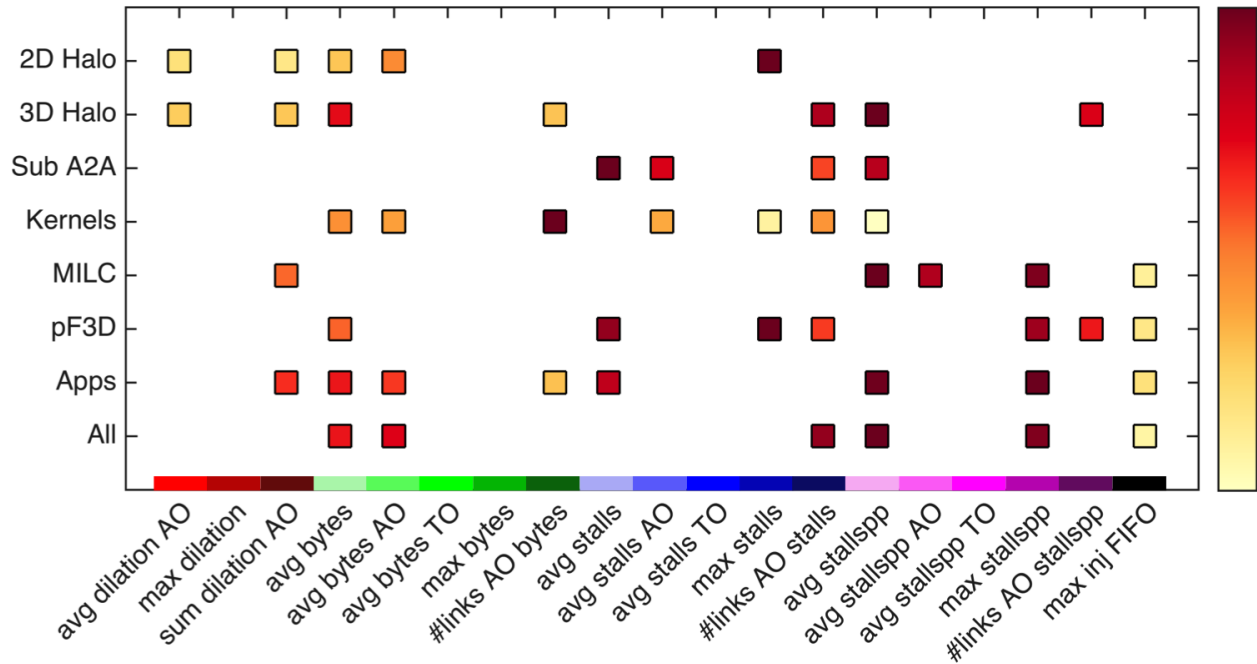


Figure 4.15: Comparison of the feature ranks obtained using the feature selection technique applied to the eight larger datasets. Note that the marker colors for each row/dataset are scaled independently (red is high and blue is low).

The marker colors for each row/dataset are scaled independently (red is high and blue is low).

We can make a few important observations from Figure 4.15. The markers in the row for the *All* dataset show that the “stalls” features are the most important. The stalls group indicates scenarios in which a network packet has to wait in the receive buffer. The wait could either be on an intermediate node because the next link is busy or on the destination node because the node is not able to consume the received packets at the same rate as they arrive. *Stallspp* refers to the number of stalls encountered on a link per packet. The high ranks of these features suggest that the receive buffers on the nodes are one of the most important causes of network congestion.

The other important feature in the *All* dataset of Figure 4.15 is *avg bytes*. This refers to the average number of bytes passing through a network link and is an indicator of the average traffic on the network. A high rank for this feature suggests that to mitigate congestion, algorithms and task mappings should aim to keep the average load per link low. It also indicates that *max bytes* or the most overloaded link (often referred to as a hot-spot) is not a strong determinant of the execution time. Finally, we note that *max inj FIFO* or the maximum injection FIFO length also plays a small part in predicting the execution time, especially for the production applications.

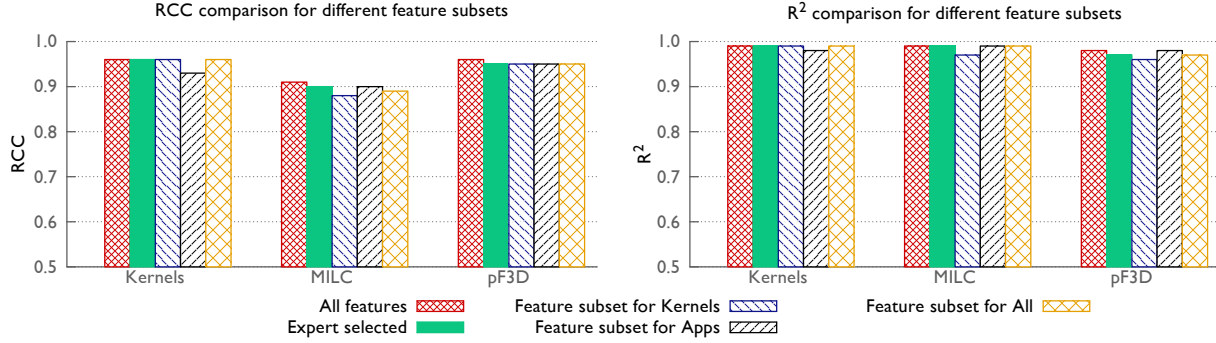


Figure 4.16: Prediction performance of the features selected using the proposed quantile analysis on different datasets.

We can also observe which features are important for a particular larger dataset using Figure 4.15. For example, *max inj FIFO* is only important for the production applications (MILC and pF3D) and the Apps and All datasets. *Avg stallpp* is important across most datasets but *max stallpp* is only important for the production applications and their combinations. *Avg bytes* is also important in almost all datasets except Sub A2A and MILC. Further, it is important to note that a feature might not show up as important for a dataset in this figure for one of the following two reasons – either it was less important than the top 5-7 features or another feature that highly correlates with this feature was in the top list.

In order to evaluate the performance of the relevant feature subsets obtained using feature selection, we learn regression functions using the subsets on the following datasets: Kernels, MILC, and pF3D. In addition, we compare the performance for these feature subsets with that obtained by using all nineteen features and an expert selected subset of twelve features. For the expert selection, we only pick those features that we believe represent some unique information about the dataset. We pick one of two features if they are known to be highly correlated. In each case, we run GBRT regression with 70% of the data for training and the rest for testing. The results reported in Figure 4.16 were obtained by averaging the RCC and R^2 values over 50 random splits of training and testing sets.

In Figure 4.16, we observe that if we use the feature subset from the communication kernels to predict MILC or pF3D, there is a performance drop. This is also true if we use the feature subset from the *Apps* to predict the communication kernels. This suggests that the communication kernels dataset has some characteristics that are not well modeled by the features extracted from the *Apps* dataset and vice versa.

Nonetheless, the main result in Figure 4.16 is that using a feature subset derived from all the datasets, we can do reasonably good predictions for communication kernels and produc-

tion applications. These predictions are close to predictions performed using all nineteen features. This subset of features is: *avg bytes*, *avg bytes AO*, *#links AO stall*, *avg stallpp*, *max stallpp* and *max inj FIFO*. From these results, we conclude that the features that are among the primary root causes of network congestion are (in decreasing order of importance): the average and maximum length of receive buffers, average load on the network links, and the maximum length of injection FIFOs on the source node. We also observe that the maximum load on a link (network hot-spots) and the dilation or hops a message travels are lesser indicators of network congestion.

4.6 Summary

The ability to predict the performance of communication-heavy parallel applications without actual execution can be very useful. This requires understanding which network hardware components affect communication and in turn, performance on different interconnection architectures. A better understanding of the network behavior and congestion can help in performance tuning through the development of congestion-avoiding and congestion-minimizing algorithms.

This chapter presented a machine learning approach to understand network congestion on supercomputer networks. We used regression analysis on communication data and execution time to find correlations between the two and learn models for predicting execution time of new samples. Using the technique of feature subset selection, we were also able to extract the relative importance of different features and the corresponding hardware components in predicting execution time. This helped us identify the primary root causes of network congestion which is a difficult challenge.

Using our methodology, we showed prediction scores close to 1.0 for individual datasets. We were also able to reasonably predict the execution time on higher node counts using training data for smaller node counts. We also obtained reasonable ranking predictions for new applications using datasets based on communication kernels only. Finally, we identified the hardware components that are primarily responsible for predicting the execution time. These are – receive buffers on intermediate nodes, network links and injection FIFOs in decreasing order of importance. We also observed that network hot-spots and the dilation or hops a message travels are lesser indicators of network congestion. This knowledge gives us a real insight into network congestion on torus interconnects and can be very useful to network designers and application developers.

TraceR: PDES Simulator

The design and deployment of large supercomputers with hundreds of thousands of cores is a daunting task. Both at the design stage and after the machine is installed, several decisions about the node architecture and the interconnection network need to be made. Application developers and end users are often interested in studying the effects of these decisions on their codes' performance for existing and future machines. Hence, tools that can predict these effects are important. So far, we have presented two such prediction and analysis methodologies (Section 3.2 and Chapter 4). However, each of these methods has deficiencies that limit its use. The learning method is more suited for port-mortem analysis and requires input data to characterize the executions. The functional model for dragonfly (Section 3.2) is suitable for throughput comparison based on indirect indicators of performance. However, it does not capture flow dependencies of complex applications and does not predict execution time.

In this chapter, we present a detailed simulation based method for prediction and analysis of communication performance. Discrete-event simulation (DES) based frameworks are often used to simulate interconnection networks. The usability and performance of these frameworks depend on many factors: sequential versus parallel (PDES) simulation, the level of detail at which the communication is simulated (e.g., flit-level or packet-level), whether the PDES uses conservative or optimistic parallelism methods, etc. Existing state-of-the-art DES-based network simulators suffer from two major problems. First, sequential simulators have large memory footprints and long execution times when simulating large execution traces. Second, some simulators can only simulate synthetic communication patterns that do not accurately represent production high-performance computing applications. These shortcomings can be eliminated by using a scalable PDES engine, which improves performance and reduces the memory footprint per node, and by replaying execution traces generated

from production HPC codes. To achieve this, we have developed a trace replay tool called TRACER for simulating messaging on HPC networks.

TRACER is designed as an application on top of the CODES simulation framework [106]. It uses traces generated by BigSim’s emulation framework [107] to simulate an application’s communication behavior by leveraging the network API exposed by CODES. Under the hood, CODES uses the Rensselaer Optimistic Simulation System (ROSS) as the PDES engine to drive the simulation [108]. Our major contributions w.r.t detailed simulation of communication are two fold. First, we have developed TRACER, a trace-driven simulator that executes under an optimistic parallel discrete-event paradigm using reversible computing for real HPC codes. As a result, it can be used for simulating production MPI and Charm++ applications by generating traces using Charm++’s emulation framework. The time taken to perform these simulations is also low, given the good scalability of TRACER and its underlying PDES engine ¹. Second, in order to perform highly accurate and diverse simulations, we have made significant changes to CODES as described later in this chapter. Heavy modifications to the torus and dragonfly models, and addition of a new fat-tree model are among the most important contributions to CODES.

5.1 Background

TRACER is built upon several existing tools which are introduced briefly below.

BigSim’s emulation framework: The first requirement of simulating a parallel execution is the ability to record the control flow and communication pattern of an application. The BigSim emulation framework [107] exploits the concept of virtualization in Charm++ [110] to execute a large number of processes on a smaller number of physical cores and generate traces. This enables trace generation for networks of sizes that have not been built yet. Using AMPI [90], this feature enables trace generation for production MPI applications as well.

ROSS PDES engine: ROSS [108] is a general purpose, massively parallel, discrete-event simulator. ROSS allows users to define logical processes (LPs) distributed among processors and to schedule time-stamped events either locally or remotely. The ROSS core ensures that all the posted events are delivered and scheduled on the respective targets ordered by their time-stamps. The user is free to choose a static mapping of the LPs to the physical

¹The initial implementation of TRACER was done by Bilge Acun under the guidance of Nikhil Jain and Abhinav Bhatele [109]. Various features such as simulation of multiple concurrent jobs, simulation of collectives, task-aware mapping, online-modification of traces, etc. were added by Nikhil Jain.

processors they are executed on. ROSS provides two execution modes that differ in the mechanisms used to ensure the ordering of events according to their time-stamps. The *conservative* mode executes an event for an LP only when it is guaranteed to be the next lowest time-stamped event for it. Ensuring such a requirement requires synchronization among all processors and may result in low performance for parallel ROSS. On the other hand, the *optimistic* mode aggressively executes events that have the lowest time-stamps among the current set of events. If an event with time-stamp lower than the last executed event is encountered for an LP, *reverse handlers* are executed for the events executed out of order to undo their effects. Optimistic scheduling with reverse handlers has been shown to have better scaling characteristics for many cases [111], including the full Sequoia run that yielded the 504 billion events per second record using PHOLD benchmark [112].

CODES: The CODES framework is built on top of ROSS to facilitate studies of HPC storage and network systems [106]. The network component of CODES, Model-net, provides an API to simulate the flow of messages on HPC networks using either detailed congestion models or theoretical models such as LogP. Model-net allows users to instantiate a prototype network based on one of these models. Such instantiations are controlled by parameters such as network type, dimensions, link bandwidth, link latency, packet size, buffer size, etc. CODES has been used to study the behavior of HPC networks for a few traffic patterns [106]. These traffic patterns have been implemented as one-off applications that use Model-net as a network driver. Recently, an application to replay DUMPI [51] traces has also been added to CODES.

5.2 Design and implementation of TRACER

TRACER is designed as an application on top of the CODES simulation framework. Figure 5.1 (left) provides an overview of TRACER’s integration with BigSim and CODES. The two primary inputs to TRACER are the traces generated by BigSim and the configuration parameters describing the interconnection network to be simulated. The meta-data in the traces is used to initialize the simulated processes. The network configuration parameters are passed to CODES to initialize the prototype network design.

We define the following terminology to describe the working of TRACER:

PE: Each simulated process (called PE) is a logical process (LP) visible to ROSS. It stores virtual time, logical state, and the status of tasks to be executed by it.

Task: The trace for a PE is a collection of tasks, each of which represents a sequential

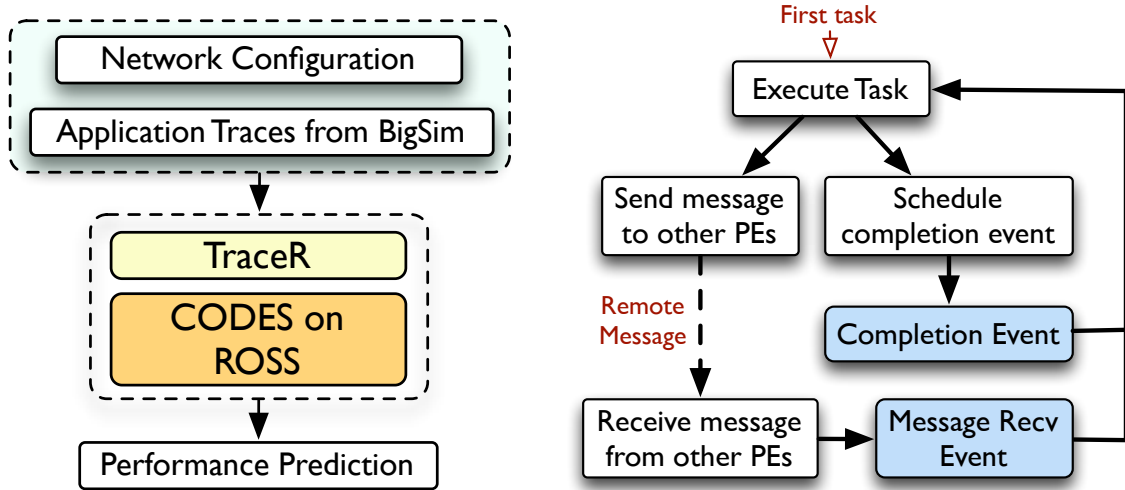


Figure 5.1: Integration of TRACER with BigSim emulation and CODES (left). Forward path control flow of trace-driven simulation (right).

execution block (SEB). A task may have multiple backward dependencies to other tasks or to message arrivals. At startup, all tasks are marked *undone*. If a task has an *undone* backward dependency, it can not be executed.

Event: A unit entity that represents an action with a time-stamp in the PDES. We implement three types of events in TRACER:

- *Kickoff* event starts the simulation of a PE.
- *Message Recv* event is triggered when a message is received for a PE. The network message transmission and reception is performed by CODES.
- *Completion* event is generated when a task execution is completed.

Reverse Handler: Another unit entity which is responsible for reversing the effect of an event. It is needed only for the optimistic simulation mode.

Let us consider an MPI application that performs an iterative 5-point stencil computation on a structured 2D grid to understand the simulation process. In each iteration, every MPI process sends boundary elements to its four neighbors and waits for ghost elements from those neighbors. When the data arrives, the MPI process performs the 5-point stencil followed by a global reduction to determine if another iteration is required. From TRACER’s perspective, every MPI process is a PE. Tasks are work performed by these MPI processes locally: initial setup, sending boundary elements, the 5-point stencil computation, etc. The Kickoff event triggers the initial setup task. Whenever an MPI process receives ghost elements, a Message Recv event is generated. The dependence of the stencil computation on the receives of ghost elements is an example of a backward dependency. Similarly, posting of a receive by an MPI

Algorithm 1 Event handler implementation for PEs: code lines that begin with an asterisk (*) are required only in the optimistic mode.

pe_busy: A boolean; set to true if the PE is executing a task at the current virtual time.
ready_tasks: List(FIFO) of tasks that are ready to be executed on a PE if *pe_busy* is false.
trigger_task: Map that stores the task executed at the completion of a given task.

<p>(a) Execute_Task(task_id)</p> <ol style="list-style-type: none"> 1: Get the current virtual time of the PE, t_s. 2: Mark the PE busy, $pe_busy = true$. 3: Send out messages of the task with their offsets from t_s. 4: Get the execution time of the task, t_e. 5: Schedule a Completion event for the PE at time $t_s + t_e$ for this task. <hr/> <p>(b) Receive_Msg_Event(msg)</p> <ol style="list-style-type: none"> 1: Find the task T that depends on the message. 2: If T does not have any undone backward dependencies, add T to <i>ready_tasks</i>. 3: if $pe_busy == false$ then <li style="padding-left: 20px;">4: Get the next task, T', from <i>ready_tasks</i>. <li style="padding-left: 20px;">5: *Store T' and pe_busy for possible use in the reverse handler; $trigger_task[T] = T'$, $busy_state[T] = pe_busy$. <li style="padding-left: 20px;">6: Call <code>Execute_Task(T')</code>. 7: end if 	<p>(c) Completion_Event(msg)</p> <ol style="list-style-type: none"> 1: Get the completed task, T, from the <i>msg</i>. 2: Mark T as <i>done</i>. 3: Set $pe_busy = false$. 4: for every task, f, that depends on T do <li style="padding-left: 20px;">5: if f does not have any undone backward dependency then <li style="padding-left: 40px;">6: Add f to <i>ready_tasks</i>. <li style="padding-left: 20px;">7: end if 8: end for 9: Get the next task, T' from <i>ready_tasks</i>. 10: *Store T' for possible use in the reverse handler, $trigger_task[T] = T'$. 11: Call <code>Execute_Task(T')</code>.
--	--

process is a prerequisite for TRACER to execute the Message Recv event.

Figure 5.1 (right) presents the forward path control flow of a typical simulation in TRACER. Application traces are initially read and stored in memory. When the system setup is complete, the Kickoff event for every PE is executed, wherein the PEs execute their first task. In the 2D stencil example, this leads to execution of the initial setup task. What happens next depends on the content of the task being executed, and the virtual time, t_s , at which the task is executed.

Every task T has an execution time t_e , which represents the virtual time T takes for executing the SEB it represents. When a task is executed, TRACER marks the PE busy and schedules a Completion event for T at $t_s + t_e$ (Algorithm 1(a)). During the execution of a task, messages for other PEs may be generated. These actions are representative of what happens in real execution. When the MPI process is executing a SEB, e.g. to send boundary elements, the process is busy and no other SEB can be executed till the sending of boundary is complete. The generated messages are handed over to CODES for delivery. Note that the execution of a task in our framework only amounts to fast-forwarding of the PE's virtual time and delegation of messages to CODES; the actual computation performed by the SEB is not repeated.

When a Completion event is executed, the task T is marked done and the PE is marked available (Algorithm 1(c)). Next, some of the tasks whose backward dependencies included

T may now be ready to execute. Thus, those tasks are added to a list of pending tasks, *ready_tasks*. Finally, the task at the top of *ready_tasks* list is selected and *Execute_task* function is called (Figure 5.1 (right)).

As the simulation progresses, a PE may receive messages from other PEs. When a message is received, if the task dependent on the incoming message has no other undone backward dependency, it is added to the *ready_tasks* list (Algorithm 1(b)). If the PE is marked available when a message is received, the next task from the *ready_tasks* list is executed. After the initial tasks are executed, more tasks become eligible for execution. Eventually, all tasks are marked done, and simulation is terminated.

5.2.1 Running TRACER in optimistic mode

When executing in the optimistic mode, TRACER speculatively executes available events on a PE. When all the messages finally arrive, ROSS may discover that some events were executed out of order and *rolls back* the PE to rectify the error. In order to exploit the speculative event scheduling, TRACER does two things. First, during the forward execution of an event, extra information required to undo the effect of a speculatively executed event is stored. In Algorithm 1, these actions are marked with an asterisk. For both Message Recv and Completion events, the data stored includes the task whose execution is triggered by these events. For the Message Recv event, whether the PE was executing an SEB when the message was received is also stored. If this information is not stored by TRACER, it will get lost and hence the rollback will not be possible.

Second, as shown in Algorithm 2, reverse handlers for each of the events are implemented. These handlers are responsible for reversing the effect of forward execution using the information stored for them. For example, in the stencil code, reverse handler for a Message Recv event reverts the MPI process back to a state where it was still waiting for the ghost elements. In general, for a Message Recv event, the reverse handler marks the message as *not received*, while the reverse handler of a Completion event marks the task as *undone*. In addition, the tasks that were added to the *ready_tasks* list are removed from the list. Both the reverse handlers also add the task triggered by the event to the *ready_tasks* list.

5.3 Network models in CODES

The CODES framework provides generic infrastructure for simulating various network models. Two types of LPs are used to simulate the network: one type to simulate the functioning

Algorithm 2 Reverse handler implementations: they use extra information stored by event handlers to undo their effect.

<p>(a) Message_Recv_Rev_Handler(msg)</p> <ol style="list-style-type: none"> 1: Find the task T that depends on the message. 2: Recover the busy state of the PE, $pe_busy = busy_state[T]$. 3: if $pe_busy == false$ then <li style="padding-left: 20px;">4: Add $trigger_task[T]$ to the front of the $ready_tasks$. 5: end if 6: Remove T from the $ready_tasks$. 	<p>(b) Completion_Rev_Handler(msg)</p> <ol style="list-style-type: none"> 1: Get the completed task, T, from the msg. 2: Mark T as undone. 3: Remove the tasks that depends on T from the bottom of $ready_tasks$. 4: Add $trigger_task[T]$ to the front of $ready_tasks$.
--	---

of a NIC and another to simulate routers/switches. The implementation of the NIC LP is common to most networks. It includes support for queuing the messages that needs to be sent out, different scheduling methods to issue packets, and receiving the messages. Currently, three type of scheduling policies are available: first-come-first-serve (FCFS) i.e. all packets of a message are issued before the next message is packetized, round-robin where one packet is issued for each enqueued message in a cyclic fashion, and a priority-based ordering. Each network model is required to provide a method that creates the next packet for the message selected based on the scheduling policy.

At the time of writing this thesis, the production version of CODES provides torus and dragonfly network models among others, which are of current interest to the HPC community. We first describe the fat-tree model that we have added to CODES, and then discuss the major improvements that have been made to other models.

5.3.1 Fat-tree model

We have added a full-bisection fat-tree network model with up to three levels of switches/routers to CODES. Instead of using high-radix fat switches at the upper layers, our construction uses multiple switches of same radix (as the lower layers) to form logical high radix switches at upper layers. This mode of building fat-tree networks resembles a folded Clos network [5], and is often used for practical deployment (e.g. Cab at LLNL) and in simulations [50, 52].

Figure 5.2 shows an example three level fat-tree built with switches of radix 4. In the construction described next, this fat-tree is used as a running example. Let the input be radix k switches with n levels desired in the fat-tree. At the lowest level (L0), the leaf level switches are placed (shown as blue boxes in Figure 5.2). For a full fat-tree constructed using

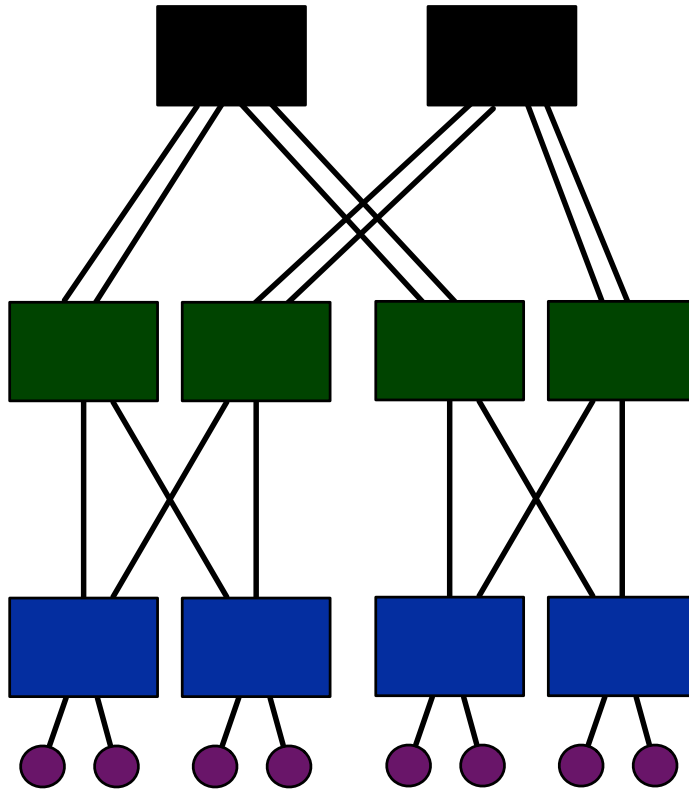


Figure 5.2: Fat-tree construction using switches of same radix.

radix k switches and n levels, $(\frac{k}{2})^{n-1}$ leaf switches are needed. Each of these switches can connect to $\frac{k}{2}$ nodes, which leads to $(\frac{k}{2})^n$ nodes being the maximum size of a system built with these parameters. The L0 switches are divided into sets of $\frac{k}{2}$ switches to form L0 groups. Switches at the next level (L1) are also divided in a similar fashion. Each of the L0 groups is mapped to a L1 group (and vice-versa); all the switches in the mapped groups are connected to form complete bipartite graphs. In Figure 5.2, L0 and L1 switches shown as blue and green boxes are divided into 2 groups each of size 2. It can be seen that switches in these groups are connected to form complete bipartite graphs.

As more levels are added, similar grouping and mapping of switches can be performed. However, due to the limited radix of switches, complete bipartite graphs are not created. Instead, each group at a given level is subdivided into smaller groups; these subgroups are then connected to other subgroups of a paired group in the adjacent level. At the highest level (L2 in our construction), the number of switches is half the number of switches at other levels. This is because L2 switches can use all their ports to connect to switches at lower levels. As a result, in this case, two connections are added between every pair of switches that are connected. This is illustrated by the connections between the black and green boxes

in Figure 5.2. Note that, if the fat-tree has only two levels, the number of switches at L1 will be half the number of switches at L0, and each subgroup at L1 will connect to two subgroups at L0.

5.3.2 Adaptive routing

The existing network model for torus in CODES simulates dimension-ordered static routing [111]. However, it has been shown that adaptive or dynamic routing improves communication performance, especially for applications with large messages [67, 113, 114]. Hence, we have added a fully-adaptive minimal path routing scheme for the torus network model in CODES. Based on the routing scheme used in IBM’s Blue Gene series [114], the implemented adaptive routing attempts to move the packets towards their destination by making locally optimal decision. When a packet arrives at an intermediate router, all ports through which the packet can be forwarded are computed. Among these ports, the one with minimum buffer length is selected to enqueue the packet. In order to preserve the accuracy of simulations with adaptive routing, we also added the functionality to handle possibly out-of-order arrival of packets at the destination nodes.

For the fat-tree network, several pattern specific static routing schemes have been proposed [115, 116]. However, given our immediate focus on comparison of different networks using a diverse range of benchmarks, we have implemented a pattern-oblivious adaptive routing scheme for the fat-tree network. Like the torus network, when a packet arrives at a switch, all ports through which the packet can be forwarded are computed. If the packet is going *up* and is to be forwarded to the next layer of switches, many ports (half the radix of the switch) are eligible. In contrast, if the packet is going *down*, the number of eligible port depends on the level of the switch: lower the level of current switch, fewer the number of eligible ports. In either case, the port with minimum buffer length is selected to enqueue the packet. We acknowledge that this routing scheme may not be the best routing scheme for certain communication patterns for fat-tree networks. However, it provides a fair comparison of fat-tree networks with other networks for the general case.

5.3.3 Deadlock avoidance

Various resources in the network have limited capacity. For example, the length of buffers at the intermediate routers which store the packets to be sent out is typically large enough to store only tens of packets. Hence, use of different routing schemes and adaptivity may lead

to deadlock, which can stall the progress of communication on the network [117]. Several deadlock avoidance and resolution mechanisms have been proposed in the past and are used in practical deployments of various networks [118–120]. Many of these schemes prevent deadlock by restricting formation of cycles during routing of packets. Next, we briefly describe two of these schemes that we have used to avoid deadlock in the network models in CODES.

- Torus: For the torus network, we have added the Puente et al.’s *bubble* routing scheme which avoids deadlock by use of an *escape* channel [120]. For dimension-ordered static routing, this scheme requires a packet to be forwarded to the next port only when there is enough space for a full packet in the next port’s buffer. If the packet makes a turn from one dimension to another or is newly injected into the network, bubble routing requires enough space for at least two full packets in destination buffer. These rules are applied only to a special channel, called escape channel.

In addition to the escape channel, the network can use other virtual channels to forward packets without fulfilling the given constraints. Typically, all packets are initially inserted into these *other* virtual channels. While forwarding a packet, if other virtual channels are blocked (due to cycle formation or overflow of resources), packets can be placed in the escape channel only when there is space for at least two full packets in the buffer. The routing used in the other channels can be selected independently of the escape channel.

- Fat-tree: In a minimally routed fat-tree, cycles cannot form. Hence, we do not need to add any deadlock prevention mechanism to the fat-tree network model.
- Dragonfly: Like torus, cycles can form when packets are routed on dragonfly networks for both minimal static routing and non-minimal adaptive routing. Kim et al. [15] proposed the use multiple virtual channels to avoid deadlock. We have implemented their scheme using minimal number of virtual channels. In the original scheme, deadlock is avoided by using as many virtual channels per connection as the maximum number of hops, *max_hops*, a packet may take in a given network. These virtual channels are numbered from 0 to *max_hops* - 1 locally on every router for every connection. When a packet is forwarded at a router, it is inserted to a virtual channel whose number is one more than the current virtual channel the packet occupied.

For non-minimal routing, the maximum number of hops is 5. However, any given connection is either local (connects routers within a group) or remote (connects routers across groups). The maximum number of local and global hops a packet may take are

3 and 2, respectively. Hence, only 3 virtual channels are needed per connection to implemented deadlock free routing on dragonfly networks. A separate count is kept for the last local and global virtual channels a packet was enqueued in, and this count is used to determine the number of the channel the packet is buffered in at the next router.

5.4 Simulation configuration

The complexity involved in simulating real codes over a PDES engine manifests itself in a number of design and parameter choices. The first choice is the type of PDES engine: conservative versus optimistic. While the optimistic mode provides an opportunity for exploiting parallelism by speculative scheduling of events, the benefits of speculative scheduling may be offset by the repeated rollbacks for scenarios with tight coupling of LPs. Conservative mode does not pose such a risk, but spends a large amount of time on global synchronization.

Another option that is available through the BigSim emulation is defining regions of interest in the emulation traces. TRACER can exploit this functionality to skip unimportant events such as program startup. For some applications, this can speed the simulation significantly. Next, we briefly describe some other important configuration parameters:

Event granularity: This parameter decides the number of tasks to execute when an event is scheduled. We can either execute only the immediate task dependent on this event or all the tasks in the *ready_tasks* list. The former leads to one completion event per task, while in the latter method, a single completion event is scheduled for all the tasks executed as a set. The second option reduces the number of completion events to the minimum required.

Execution of one task per event may lead to a larger number of events and hence results in overheads related to scheduling and maintaining the event state. However, it simplifies the storage of information for reverse computation since only one task needs to be stored per event. In contrast, when we execute multiple tasks per event, a variable length array of all executed tasks needs to be stored. This leads to inefficiency in terms of memory usage and memory access. However, the number of total events is fewer thus reducing the PDES engine overheads. These modes are referred to as TRACER-single and TRACER-multi in Figure 5.6.

Parameters for optimistic mode: There are three important parameters that are available in the optimistic mode only: batch size, global virtual time (GVT) interval and number of LPs per kernel process (KP).

- *Batch size* defines the maximum number of events executed between consecutive checks on the rollback queue to see if rollbacks are required.
- *GVT interval* is the number of batches of events executed between consecutive rounds of GVT computation and garbage collection. GVT is the minimum virtual time across all LPs and its computation leads to global synchronization.
- *Number of LPs per KP*: ROSS groups LPs into kernel processes (KPs), which is the granularity at which rollbacks are performed.

Multi-job simulation: The common scenarios for using large scale systems involve concurrent execution of many jobs. Depending on the network being used and the distribution of the jobs on the network, the execution and performance of these jobs may be affected by other jobs. For example, if two jobs share communication links, the effective bandwidth observed by these two jobs may be less than the maximum available bandwidth. In order to facilitate understanding of such effects, TRACER supports concurrent simulation of multiple jobs traces.

Conducting multi-job simulations with TRACER is straightforward and uses the same traces as the single job simulation. These traces for individual jobs are generated independently and thus can be reused across various studies. When TRACER is executed, a list of individual jobs and the location of their traces is provided in TRACER's configuration file. These jobs are attached to different cores and nodes in the network using a mapping of global rank (across the entire system) to the pair of job ID and local rank (within the job). Thereafter, all the jobs are simulated simultaneously. TRACER handles the task of converting the pair of job ID and local rank to global rank before the messages are offloaded to CODES for transmission. Similarly, on receiving a message from CODES, TRACER performs the reverse mapping from global rank to the pair of job ID and local rank.

Job placement and task mapping: In addition to supporting multi-job simulations, TRACER also provides functionality for the users to select placement of the jobs being executed and mapping of tasks within a job. Both these configurations are specified via a binary file that contains mapping of global rank to the pair of job ID and local rank. By letting users specify these two configurations, TRACER enables comparison study of various mapping schemes with minimal effort from the users.

5.5 Impact of simulation configuration

In this section, we evaluate the performance of TRACER as a simulation tool. Results based on application of TRACER, e.g. to compare various networks, will be discussed in the following chapters.

5.5.1 Experimental setup and configuration parameters

Proxy applications: We use two proxy applications for evaluating TRACER. *3D Stencil* is an MPI code that performs Jacobi relaxation on a 3D process grid. In each iteration of 3D Stencil, every MPI process exchanges its boundary elements with six neighbors, two in each direction. Then, the temperature of every grid point is updated using a 7-point stencil. In our experiments, we allocate $128 \times 128 \times 128$ grid points on each MPI process, and hence have 128 KB messages. *LeanMD* is a Charm++ proxy application for the short-range force calculations in NAMD [121]. We use a molecular system of 1.2 million atoms as input to LeanMD. We simulate three iterations of each benchmark.

Simulated networks: We simulate 3D tori of sizes 512 to 524, 288 nodes to measure and compare simulator performance. For these studies, 3D torus has been used because it is the only common topology available in all simulators used in this section. For validation, we simulate a 5D torus because isolated allocations on IBM Blue Gene/Q (which has a 5D torus) allow us to collect valid performance data. The fat-tree network model is validated by comparing simulation results of 3-level fat-trees with radix 8 switches from TRACER and Booksim [52], a cycle accurate simulator. Dimensions of the 3D tori are chosen to be as cubic as possible and those of the 5D tori mimic the real allocations on IBM Blue Gene/Q. For the 3D torus, we have experimented with two types of congestion models: a packet-level congestion model based on IBM’s Blue Gene/P system (TorusNet) [111] and a topology-oblivious packet-level $\alpha - \beta$ model (SimpleNet). The simulation runs were performed on Blue Waters, a Cray XE6 at NCSA, while the emulation (trace generation) and validation were performed on Vulcan, an IBM Blue Gene/Q system at LLNL.

Evaluation metrics: We use three metrics to compare and analyze the performance of different simulators:

- Execution time: time spent in performing the simulation (excluding startup).
- Event rate: number of committed events executed per second
- Event efficiency: represents the “rollback efficiency” and is defined as:

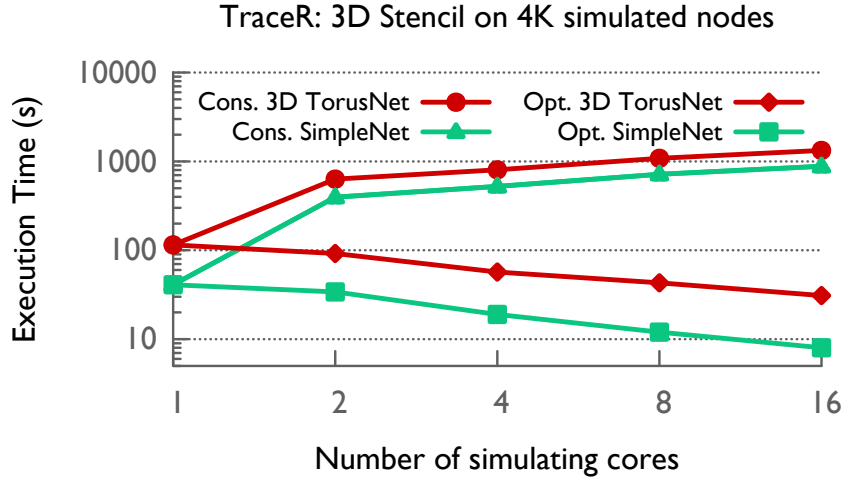


Figure 5.3: Optimistic vs. conservative DES

$$\text{Event efficiency (\%)} = \left(1 - \frac{\#rolled\ back\ events}{\#committed\ events} \right) \times 100$$

Based on the equation above, when the number of events rolled back is greater than the number of committed events (events that are not rolled back, which equals the number of events executed in a sequential simulation), the efficiency is negative. A parallel simulator may be scalable even if its event efficiency is negative. This is because while using more cores may not improve event efficiency, it may reduce the execution time due to additional parallelism.

5.5.2 Conservative versus optimistic simulation

We begin with comparing the conservative and optimistic modes in TRACER. In these experiments, we simulate the execution of 3D Stencil on 4K nodes of a 3D Torus using 1 to 16 cores of Blue Waters. As shown in Figure 5.3, the execution time for the conservative mode increases with the number of cores, but decreases for the optimistic mode (for both TorusNet and SimpleNet). Detailed profiles of these executions show that the conservative mode performs global synchronization 43 million times which accounts for 31% of the execution time. Overall, 60% of the total execution time is spent in communication.

In contrast, the optimistic mode synchronizes only 1,239 times for GVT calculation with communication accounting for 10% of the execution time. This is in part due to the overlap of communication with useful computation and in part due to the lazy nature of global synchronization in the optimistic mode. Based on these results, we conclude that the optimistic

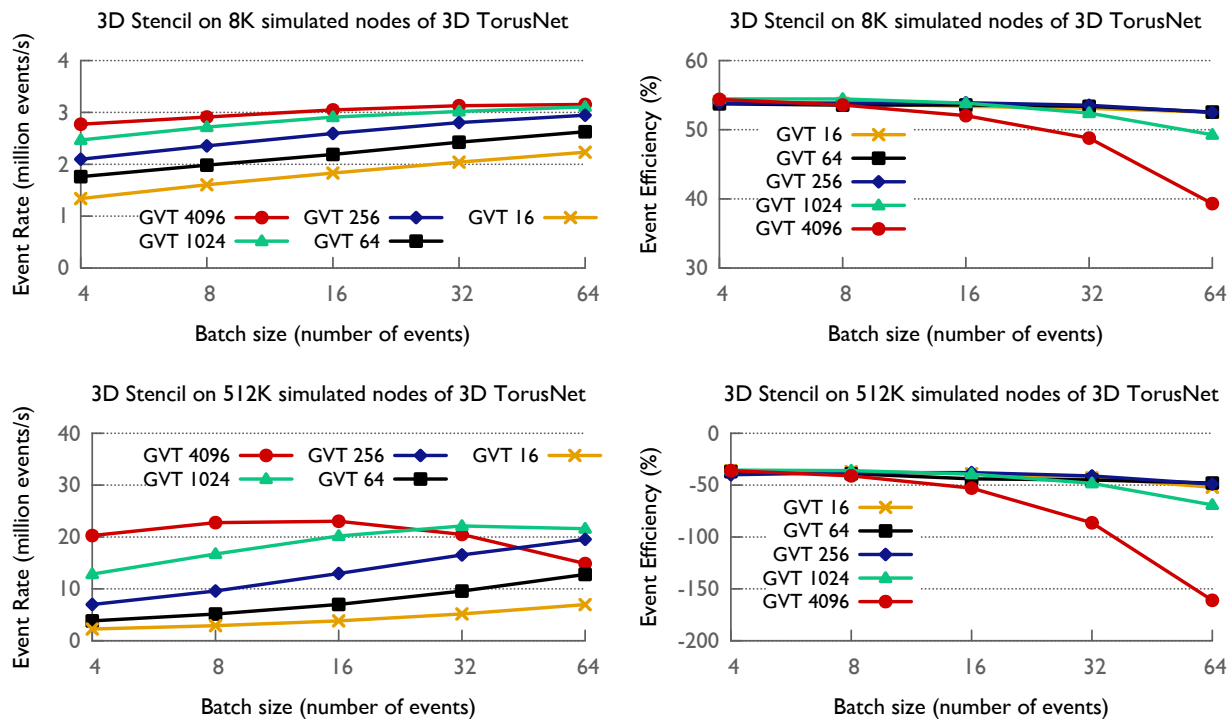


Figure 5.4: Effect of batch size and GVT interval on performance: 8K simulated nodes are simulated using 8 cores (top 2 plots), and 512K using 256 cores (bottom 2 plots).

mode is suitable for performing large simulations using TRACER and we use it for the results in the rest of this thesis.

5.5.3 Effect of batch size and GVT interval

Figure 5.4 shows the impact of batch size and GVT interval on performance when simulating 3D Stencil on a 3D torus with 8K nodes and 512K nodes using 8 and 256 cores, respectively. Note that the choice of the number of simulated nodes and that of simulating cores affects the results minimally except at the limits of strong scaling. These results are for the TorusNet model; similar results were obtained for SimpleNet model also. The first observation from Figure 5.4 (left) is the diminishing increase in the event rate as the batch size is increased. The improvement in the event rate is because of two reasons: positive impact of spatial and temporal locality in consecutive event executions and overlap of communication with computation. However, as the batch size becomes very large, the communication engine is progressed infrequently which reduces the overlap of communication with computation. At the same time, the number of rollbacks increases due to the delay in communication and execution of pending events to be rolled back. These effects become more prominent on

#LPs/KP	Efficiency (%)	Time(s)
1	51	82
2	38	92
16	2	119
128	-87	189

Figure 5.5: Impact of #LPs per KP.

larger core counts as shown by the event efficiency plots in Figure 5.4 (right).

Next, we observe that the event rate typically improves when a large GVT interval is used. This is because as the GVT interval is increased, the time spent in performing global synchronization is reduced. Infrequent synchronization also reduces idle time since LPs with variation in load do not need to wait for one another. These results are *in contrast* to past findings that were performed on PDES with uniform loads on the LPs [108]. When a sufficiently large GVT interval is used with a large batch size, memory limits force certain LPs to idle wait till the next garbage collection. As a result, the rollback efficiency and event rates drop as shown in the Figure 5.4. Based on these findings, we use a batch size of 16 and GVT interval of 1024 for all simulations in the rest of this chapter.

5.5.4 Impact of number of LPs per KP

ROSS groups LPs together to form kernel processes (KPs) to optimize garbage collection. This causes all LPs in a KP to rollback if any one of the LPs has to rollback. In [108], it was shown that although smaller values of LPs per KP reduce the number of rolled back events, they do not have a significant impact on the execution time. Our findings, shown by a representative set in Figure 5.5 (obtained by simulating 3D Stencil on 8K nodes of 3D Torus with TorusNet model on 8 cores), differ – smaller values of LPs per KP *reduce* the execution time also. As we reduce the number of LPs per KP from 128 to 1, the execution time decreases by 57%.

The primary reason for the difference in impact of LPs per KP is the varying event efficiency. For synthetic benchmarks used in [108], the event efficiency is greater than 95% in all cases. As a result, any further increase caused by decreasing LPs per KP is marginal. In contrast, for real application simulations, the event efficiency is much lower. Thus, a reduction in the number of rollbacks can significantly impact the overall execution time.

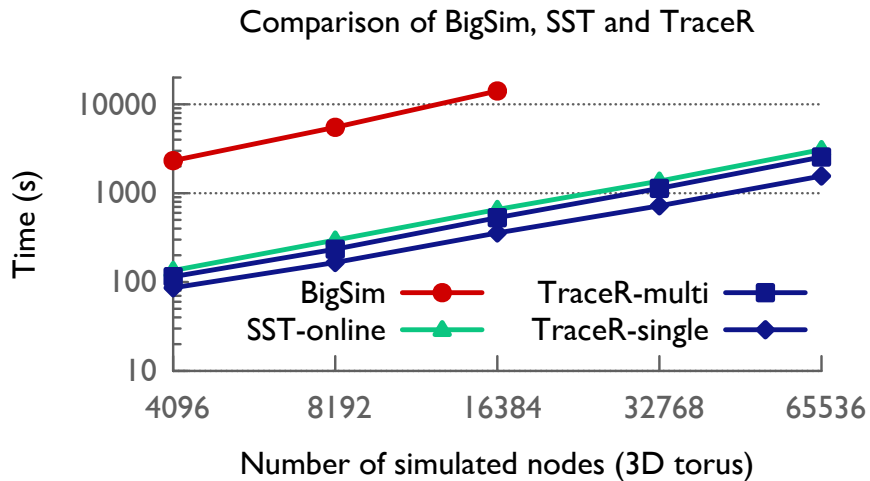


Figure 5.6: Sequential simulation time.

5.6 Performance comparison

We now compare the performance of TRACER with other simulators and analyze its scaling performance and prediction accuracy using the packet-level model (TorusNet). Here, TRACER is executed in optimistic mode with batch size = 16, and GVT interval = 2048. The simulated network topology is 3D torus, which is the only topology available in TRACER, BigSim, and SST.

5.6.1 Comparison with sequential executions

We first compare the sequential performance of BigSim, SST (online mode), TRACER-single, and TRACER-multi for simulating 3D Stencil’s execution on various node counts. Figure 5.6 shows that TRACER is an order of magnitude faster than BigSim. This is primarily because of the inefficiencies in BigSim’s torus model and its PDES engine. Compared to SST, the execution time of

The simulation time with TRACER-single is lower by 50% in comparison to SST, which we believe is due to ROSS’s high performing DES engine. The performance of TRACER-single is also better than TRACER-multi. This is because while TRACER-single increases number of events visible to ROSS, it makes better use of cache by storing the triggered event along with the message. Hence, for the remaining experiments in this thesis, we use and report the performance of TRACER-single as TRACER’s performance.

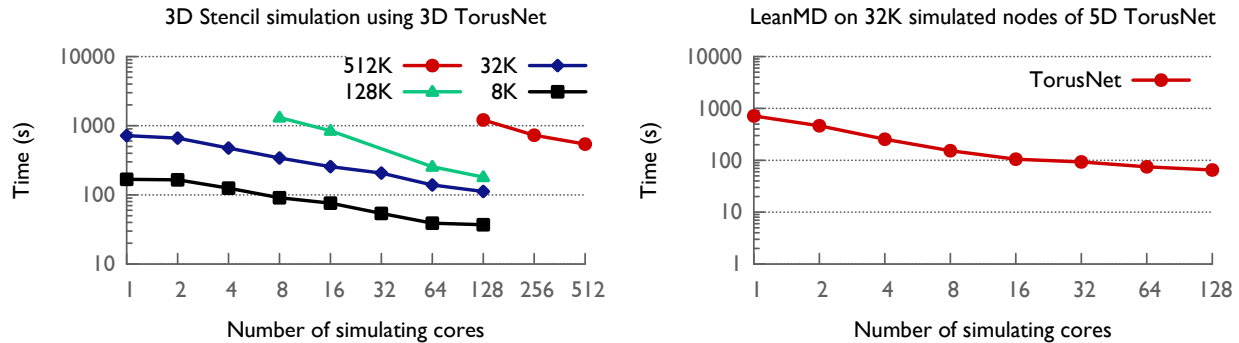


Figure 5.7: Scalability of TRACER when simulating networks of various sizes.

5.6.2 Parallel scaling and validation of TRACER

Next, we present scaling results for TRACER using packet-level TorusNet model. The comparison with other simulators was not possible due to the following reasons: 1) The parallel execution of BigSim, on 2-16 cores, is an order of magnitude slower than its sequential version. This is due to high overheads introduced by its PDES engine. 2) The parallel version of SST does not work with packet-level models, and hence is not available for a fair comparison.

Figure 5.7(left) presents the execution time for simulating 3D Stencil on various node counts of 3D torus. It is clear that TRACER scales well for all simulated system sizes. For the simulations with half a million (512K) nodes, the execution time is only 542s using TorusNet. For smaller systems, the execution time is reduced to less than 100s.

Figure 5.7 (right) shows the scaling behavior of TRACER when simulating LeanMD on 32K nodes of a 5D torus. The simulation takes only 65s on 128 cores using TorusNet. However, the speed up for simulation of LeanMD is lower in comparison to the speed up for simulating 3D Stencil. This is due to LeanMD’s relatively more complicated interaction pattern and dependency graph, which causes more rollbacks on large core counts.

Validation: To validate various network models, we have conducted the following experiments:

- Dragonfly: We have made minimal modifications to the dragonfly network model in CODES which has been validated in the past [122].
- Fat-tree: To validate the fat-tree model, we first modified the routing scheme inside Booksim to match our implementation. Alternatively, we also modified the routing scheme implemented in our network model to match with the schemes available in Booksim. Next, we implemented a benchmark that mimics the communication pattern generated by the *uniform* pattern available in Booksim. Finally, since Booksim does

not support software-related overheads, we set those values to be zero in TRACER for these validation experiments. Using a fat-tree constructed with radix 8 switches and three levels, both Booksim and TRACER are executed to predict the injected and accepted flit rate. We find that the predictions of TRACER are within 3% of the predictions made by Booksim, thus validating the fat-tree network model.

- Torus: As described earlier, the Blue Gene/Q systems allow isolated execution of jobs and hence are ideal for validating the torus network model. For both static and adaptive routing, the following benchmarks were first executed on the real machine and then simulated using TRACER to obtain real and predicted execution time.
 1. Ping pong between various pairs of nodes of different message sizes.
 2. Simultaneously communicating multiple pairs of nodes, with different placements to cover various overlap scenarios.
 3. Permutation communication pattern in which each MPI rank communicates with a randomly selected partner for an allocated of size 512 nodes with 16 cores each.

For each of these benchmarks, we find that the predicted time to execute is within 5% of the real execution time. These results combined with validation presented in earlier papers [111] demonstrate that the torus network model used by TRACER is accurate. These results also show that the simulation components added on top of CODES's network model are correct and predict the performance correctly.

5.7 Summary

In this chapter, we have presented a trace-driven simulator, TRACER, for studying communication performance of HPC applications on current and future interconnection networks. TRACER shows how one can leverage optimistic parallel discrete-event simulation with reversible computing to provide scalable performance. We have also shown that TRACER outperforms state-of-the-art simulators such as BigSim and SST in serial mode and significantly lowers the simulation time on large core counts. In the following chapters, we show two example applications of TRACER. Chapter 6 uses TRACER to compare three commonly used network topologies at the scale of the next generation supercomputers: torus, fat-tree, dragonfly. Following that, Chapter 7 deploys TRACER to study the impact of various network configuration parameters on the performance of two mini-applications executed on the torus and dragonfly networks..

Comparison of Networks

A primary motivation for development of TRACER (presented in Chapter 5) was to build a simulator that can be used to compare performance of different network topologies for different types of communication patterns in a practical time frame. In this chapter, we use TRACER to achieve that goal by first designing system prototypes with different network topologies, and then comparing communication performance obtained on these systems. For the comparison, we use a set of five communication patterns induced by mini-applications which are representative of common HPC workload. We also develop a model to estimate the cost of networks for the prototype systems, and combine the cost metrics with performance metrics to compare the cost-effectiveness of various network topologies.

6.1 Network prototypes

Three interconnect topologies are widely used to build HPC networks: fat-tree, torus, and dragonfly. Hence, we focus on comparing prototype systems of similar capabilities built using these topologies. We measure the capability by the total number of nodes in the system; this combined with the assumption that nodes with similar capacity are used in every system, leads to prototype systems with similar peak performance (FLOPS) built using different network topologies. In addition to the choice of topology, many design choices need to be made to fully define a prototype system. In order to primarily focus on the effects of network topology, we keep the following design choices constant across different prototypes of a given capability:

- Link bandwidth: Bidirectional bandwidth of links connecting a pair of routers/switches in the system.

- Injection bandwidth: Bidirectional bandwidth of bus/link that connects a node's NIC to its router/switch.
- Floating point to injection bandwidth ratio (F/B): This is the ratio of node's computation capability to the injection capability of its NIC.
- Ranks per node: Number of control flows (MPI ranks, Charm++ objects, etc.) that are executed on a given node.
- Injection policy: The policy used by the NIC to select message that should be packetized to generate the next packet transmitted to the router/switch.
- Packet size: Size of individual packets that messages are broken down to by the NIC. This is the smallest transmission entity visible to the simulator.
- Router buffer: Capacity of each of the channel of a router port that holds packets in transit; defined as a multiple of the packet size.
- Software delay: The delay caused by software stack for initiating a message send or processing a received message.

While the above mentioned design choices are kept constant across different prototypes, most of the other components/characteristics are decided based on the network topology being constructed. The fundamental design principle that guides our topology specific choices is that the network should be balanced. For example, given the large number of hops on torus topologies, the number of nodes attached to a torus router is chosen to be one [123] (more on this later in this section). Additionally, practicality of design choices is also considered. Design choices that lead to unrealistic systems have been omitted. Here is a list of components/characteristics that are determined based on the topology being used:

- Router connectivity: Fundamentally, the choice of topology determines how various pairs of routers/switches are connected.
- Number of nodes per router: In order to build a balanced network, the number of injection ports should be such that the outgoing links should be able to transfer the injected traffic at full speed, while also serving the traffic from other routers at full speed. Given that we want the same F/B ratio (floating point capacity to injection bandwidth ratio) across network topologies, this imply that the number of nodes connected to a router depends on the network topology being used.

- Routing scheme: The choice of topology also determines the routing schemes that can be used to send messages from one node to another.
- Network latency/delay: When packets are injected onto the network, they encounter delays of several types, e.g. time to compute route when packet arrive at a router, or latency to traverse from one router to another, etc. These parameters are typically dependent on the network topology and the size of the system being studied.
- Job placement: Based on the network and the communication pattern, the placement of communicating pairs that provides the best performance may change. We rely on past results, personal experience, and simulation results to find the task mapping that gives the best performance on each of the topology.

Table 6.1 summarizes the design choices made in the prototype systems that are simulated in this chapter; these choices are described in detail next.

6.1.1 Torus

Multi-dimensional torus networks have been used in many HPC systems, e.g. Cray XE's 3-D torus, IBM Blue Gene/Q's 5D torus, and K-computer's 6D-torus. In a typical torus network, messages travel many hops when sent from a source node to destination node. The average number of hops depends on the application being executed. Thus, the best one can do to create a balanced torus-based system is to attach only one node per router [123]. We also adopt this choice when designing prototype systems based on torus.

As the number of dimensions increase in a torus, the bisection bandwidth also increases for a given router (or node) count. Additionally, the number of one-hop neighbors also increases. Thus, our prototype systems use 6D-torus, which is the largest dimensionality that has been used in a production torus-based system. In terms of routing, most torus systems use minimal path routing which are either static or adaptive in nature. Our prototype systems also provides these routing options. Finally, given the low radix and localized nature of router connections, we use lowest values for the latency on torus-based prototypes.

6.1.2 Dragonfly

Dragonfly topology is being used in some of the largest next generation supercomputers being built (e.g. Aurora at ALCF, Cori at NERSC). In their introductory paper, Kim et al. [15] proposed that for a dragonfly network built using routers with radix k , $\frac{k}{4}$ ports should

be used for injection, $\frac{k}{2}$ ports should be used for local connections within a group, and $\frac{k}{4}$ connections should be used for global connections across groups. These values are suitable for creating a balanced system which uses direct routing, i.e., for every packet injected onto the network, two local and one global links are used in the worst case. However, recent studies and practical deployments [16, 85, 89] have shown that indirect or hybrid routing is essential to obtain good performance on dragonfly networks. In indirect routing, for a given packet, three local links and two global links may be used. Thus, in a balanced system which uses indirect routing, $\frac{k}{6}$ ports should be used for injection, $\frac{k}{2}$ ports should be used for local connections, and $\frac{k}{3}$ ports should be used for global connections. These are the values used in our prototype systems and simulations.

Among many routing schemes that have been proposed for dragonfly networks, three have been deployed in production systems: direct, indirect, and hybrid. Our experiments make use of these schemes for evaluating dragonfly networks. The router radix required to construct dragonfly networks is typically larger than the torus. Moreover, the length of wires that connects the groups is also longer. Hence, the latency used for dragonfly networks is higher than the latency used for torus-based system.

6.1.3 Fat-tree

Over the last 10 years or so, the number of systems with fat-tree topology, using either infiniband or ethernet, has grown significantly. These systems now account for as many as 50% of the total number of machines. Moreover, two of the largest next generation supercomputers, Summit at ORNL and Sierra at LLNL, will also be based on the fat-tree network topology. To create a full-bisection fat-tree, for every switch at level i , the number of ports connected to switches at levels $i + 1$ and $i - 1$ should be equal, i.e. half the ports should connect to switches at level $i + 1$, and the other half should connect to switches at level $i - 1$. For the leaf switches, the switches at lower levels are replaced by the nodes. This construction is used in our prototype systems to build 3-level fat-tree networks. The number of levels is chosen to be three for two reasons: 1) We want to limit the radix of the switches being used to a reasonable value. For large systems, a two-level fat-tree may require switches with very large radix (>100). 2) Use of 3-level fat-tree also limits the maximum number of hops traversed by any message to 5 switches. Adding more level increases this value further, which may impact the performance significantly.

Most routing schemes for fat-tree networks use shortest path, e.g. Up/Down routing. Many of these routing schemes are designed to optimize performance for specific communi-

Design Option	Torus	Dragonfly	Fat-tree
Number of nodes (n) and ranks per node	Kept constant (~ 46656)		
Router radix (r)	13	$6 * \text{ceil}(\left(\frac{n}{18}\right)^{\frac{1}{4}})$ (48)	$2 * \text{ceil}(n^{\frac{1}{3}})$ (72)
Nodes per router	1	$\frac{r}{6}$ (8)	$\frac{r}{2}$ (36)
Routing scheme	Static and adaptive with minimal paths	Adaptive with minimal paths, non-minimal paths, and hybrid	Adaptive with minimal path
Network latency/delay	Lowest (~ 30 ns)	Medium (~ 50 ns)	High (~ 60 ns)
Job placement	Blocked mapping	Random at node-level	Linear
Link and injection bandwidth per node	Kept constant (12.5 GB/s)		
Packet size	Kept constant (1,024)		
Router buffer	Kept constant (65,536 bytes)		
Software delay	Kept constant (1 microsecond)		

Table 6.1: Design choices for prototype systems. Specific values shown are for the systems compared in the next section. The job placement choices have been made after comparing different types of placement schemes for each of the network.

cation pattern. Given our evaluation of significantly different communication patterns, we avoid such customizations and deploy full adaptive Up/Down routing with minimal paths for the prototype systems. Finally, the latency used for fat-tree based system is highest among the three networks, due to their largest router radix and typically long wire length.

6.2 Communication performance comparison

Even when high-level decisions have been made with regard to various design choices, comparison of different network topologies requires us to fully specify prototype systems. In addition to summarizing the high-level design choices, Table 6.1 also presents these specifications. Here, our goal is to compare systems of size comparable to the largest supercomputers in the next generation of HPC systems, viz. systems with 150 – 200 PetaFlops capability. Current trends suggests that the capability of individual nodes in the next generation systems will be approximately 4 TF. Thus, we focus on systems with node count in the range 45,000 to 50,000.

In the given node range, a symmetric 6D-torus can be created with length of each dimension being 6, i.e. a $6 \times 6 \times 6 \times 6 \times 6 \times 6$ torus. The number of nodes in this prototype system is 46,656. These 46,656 nodes can also be connected as a fully-formed 3-level fat-tree using switches with 72 ports. The closest port count for a dragonfly network in this node range

is 48. However, a fully-formed dragonfly network with 48 port router consists of $> 70K$ nodes; a similar network constructed using 42 port routers can only support $\sim 43K$ nodes at its maximum. Thus, in order to construct a balanced dragonfly-based system with $\sim 46K$ nodes, we make two modifications to the dragonfly network constructed using 48 routers: 1) Only 10 global connections are attached to every router; this leads to a system with 240 groups each with 24 routers. 2) The link bandwidth of each of the global link is increased by 60%. In this way, using ten global links only, we can balance the traffic from 24 local links. This modified dragonfly system has 46,080 nodes, which is only 1.2% less than the prototype system constructed using torus and fat-tree topologies.

The link bandwidth and the injection bandwidth per node has been chosen to be 12.5 GB/s, which matches with the corresponding values for the next generation systems (to the best of our knowledge). The packet size for running the simulations has been set to 1,024 bytes for two reasons: 1) Similar values have been used in torus and dragonfly networks (BG/Q, PERCS). 2) Efficiency and accuracy of the simulator are reasonable for this value of packet size. The size of router buffer, which is typically not known in public domain, is chosen to fit 64 packets based on private conversations with researchers in industry. Finally, the network latency and software delay values have been appropriately chosen to resemble the current systems.

Job placement or task mapping: Placement of tasks relative to each other can have a significant impact on the communication performance of an executing pattern. Thus, it is important that the task mapping which provides the best performance is used when comparing different network topologies. In our experiments, we have tested various such mappings and used the one which provides the best performance as listed below:

- Torus - Mappings tested: random, linear, blocked. Best mapping: blocked mapping.
- Dragonfly - Mappings tested: random at node level, random at router level, linear, blocked, round-robin at node level, round-robin at router level. Best mapping: random at node level mapping.
- Fat-tree - Mappings tested: random, linear, blocked. Best mapping: linear mapping.

Using the prototype systems and job placement policies described above, we perform simulation of five representative mini-applications. Given the high capacity nodes, it is expected that multiple control flows (MPI ranks, Charm++ objects) should execute on each of the nodes. It is expected that the number of cores/threads available on each node will be greater than 100, possibly a few hundreds. However, it is also expected that not all these

resources will execute as independent ranks; rather within node parallelism will be deployed to reduce the number of communicating ranks per node. Based on recent trends, we choose 64 to be the number of ranks executing per node. Thus the total number of ranks simulated are around three million (64 on each of $\sim 46K$ nodes).

6.2.1 Stencil

As stated in Section 2.2, one of the common decomposition in scientific applications is to partition the problem domain in a structured manner among ranks. For example, a 3D domain can be divided among ranks arranged as a 3D grid, wherein each rank owns a sub-cuboid. In a typical relaxation-based method, this decomposition leads to each rank communicating with its neighbors in the grid. Note that the grid neighbors may be far apart in terms of MPI ranks, and thus blocked job placement is often performed to improve the communication performance.

We use 4D Stencil, in which ranks are arranged in a four-dimensional grid, as a representative mini-application for this type of decomposition and its communication pattern. The dimensions of the process grid used in these experiments are $160 \times 128 \times 192 \times 192$. Within this grid, for all the networks, the ranks within a node are blocked together as a sub-grid of dimension $4 \times 4 \times 2 \times 2$. This is done to minimize the amount of traffic that is injected onto the network. For each of the three prototype systems, execution of the 4D Stencil kernel is simulated for six different message sizes: 5 KB, 50 KB, 250 KB, 500 KB, 1 MB, 2 MB. For the torus-based system and dragonfly-based system, we found that the adaptive routing and hybrid routing provide the best performance, respectively; hence, we use them for comparison with the fat-tree based system, which used adaptive Up/Down routing.

Figure 6.1 presents a comparison of 4D Stencil’s performance when executed for different message sizes on the three networks. For the smallest message size (5 KB) the run time for all the networks is very similar (around 400 microseconds). However as the message size is increased to 50 KB and 250 KB, the increase in execution time for torus is much smaller in comparison to the other networks. For torus, the execution time increases by 6x as the message size is increased by 10x, while the increase is 9.5x and 11.5x for dragonfly and fat-tree, respectively. A further increase in message size of 5x (to 250 KB) leads to relatively similar increase in execution time on all the network (5.5x on torus, 5.9x on dragonfly, and 6.5x on fat-tree).

As shown in Figure 6.1 (right), for large messages, the increase in execution time for both torus and dragonfly network is proportional to the increase in the message size. However,

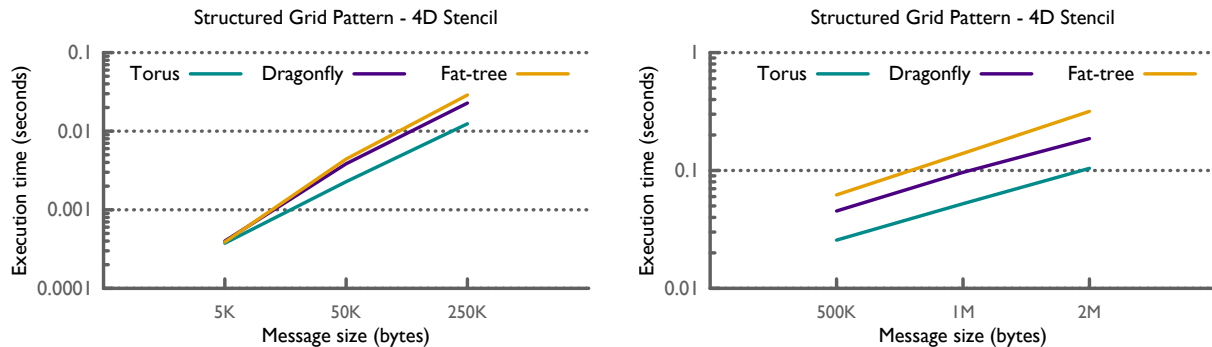


Figure 6.1: Communication performance of different networks for 4D Stencil. Torus outperforms dragonfly, which in turn performs better than fat-tree for large message sizes. For smallest message size, all networks show similar performance.

the increase in execution time on fat-tree is higher, ~ 2.25 for every 2x increase in the message size. As a result, for 2 MB messages, the execution time for fat-tree is three times the execution time on torus, and 1.7 times the time for dragonfly.

6.2.2 Neighborhood communication

The next communication pattern we study examines the effect of communication restricted to a process' rank neighborhood. Such communication frequently arise in two scenarios: 1) An application is written in such a way that a given rank communicates only with other ranks that are in its rank neighborhood (e.g. in bubble sort). 2) The user performs topology-aware mapping to ensure that all the processes that communicate heavily are placed on nearby nodes. Our benchmark, called Near-Neighbor (NN), emulates these patterns by selecting 20 – 30 communicating partners per rank within a rank neighborhood of 2000. We compared performance for various topology aware mappings and show results for the placement that provides the least execution time.

Figure 6.2 (left) compares the performance of torus, dragonfly, and fat-tree for running NN with small message sizes. At 5 KB message size, torus shows 1.7x speed up over both dragonfly and fat-tree. On all other message sizes, torus is 2x faster than dragonfly and fat-tree, which show similar performance. This is due to the high suitability of NN pattern to the torus network: by carefully mapping clusters of nearby ranks to smaller blocks (cuboids in the torus network), we can reduce the number of hops while minimizing the interference among communication induced by the clusters. Attempts to perform similar mapping in dragonfly and fat-tree leads to more interference, while not affecting the number of hops (which are already very low).

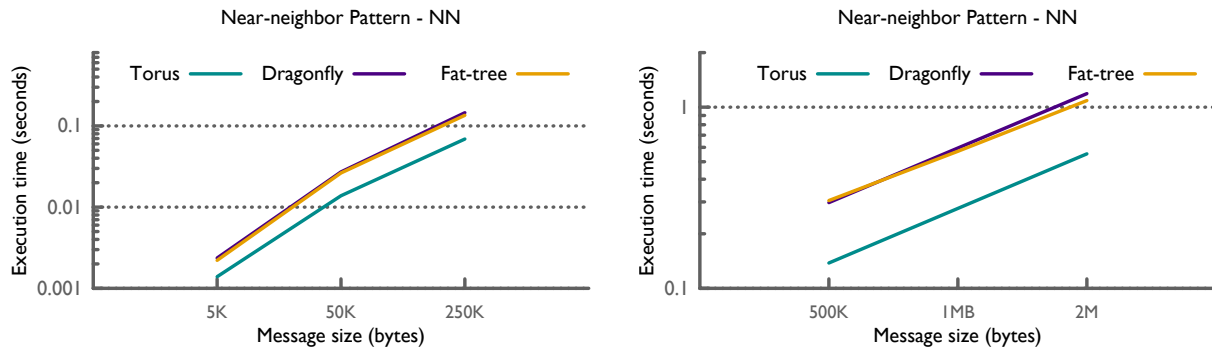


Figure 6.2: Communication performance of different networks for Near-Neighbor (NN). Irrespective of the message size, torus is faster by 2x in comparison to dragonfly and fat-tree.

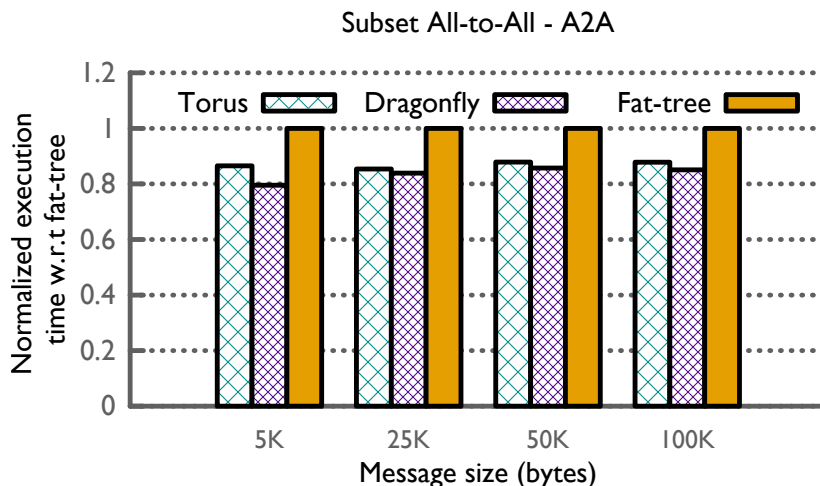


Figure 6.3: Communication performance of different networks for Subset All-to-All (A2A). With careful mapping, the execution time is similar for all the networks, with dragonfly being marginally better and fat-tree being marginally worse than torus.

6.2.3 Subset All-to-All

Multi-dimensional Fast-Fourier Transforms are one of the most common operations performed in scientific applications. Both 1D and 2D decomposition of data to perform FFT lead to a communication pattern in which subsets of ranks perform all-to-all operations within the subsets. The A2A benchmark represents this scenario in our experiments. The rank space is arranged as a 3D grid of dimensions $64 \times 144 \times 320$. The all-to-all operation is performed along the first and the second dimension, i.e. 144×320 1D FFTs are performed among subsets of size 64, and 64×320 1D FFTs are performed among subsets of size 144. In order to optimize communication, the first dimension is wrapped entirely within a node.

Figure 6.3 compares the execution time of A2A for four different message sizes: 5 KB,

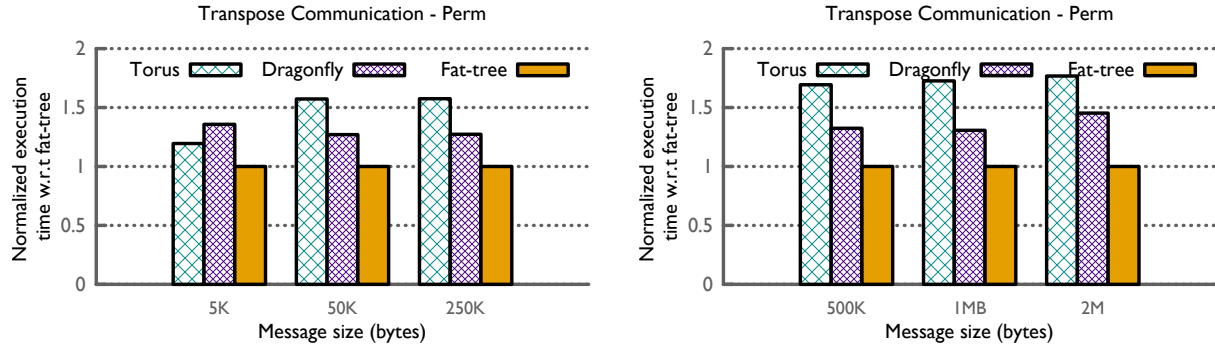


Figure 6.4: Communication performance of different networks for Perm. Fat-tree provides the best performance for all message sizes, followed by dragonfly which is better than torus by $\sim 25\%$.

25 KB, 50 KB, 100 KB. It can be seen that for all the message sizes, the performance of the three networks is comparable. The fat-tree provides marginally slower execution time, with torus and dragonfly being 10 – 20% faster. Note that this similarity of performance is obtained by finding the best mapping for each of the network. If instead of blocked mapping, the default mapping is used in torus, its execution is almost doubled. Similarly, if linear mapping is used instead of randomized mapping for the dragonfly network, its performance goes down by more than 50%. Finally, if randomized mapping is used instead of linear mapping for fat-tree, it takes almost twice the time to execute A2A for any given message size.

6.2.4 Transpose communication

The next communication pattern tests communication performance of communicating partners situated far-away, both logically and physically. In the given benchmark, Perm, a rank communicates with another rank that is placed diagonally opposite to it assuming linear rank space. Topology aware mapping is not performed for any of the networks since the purpose of this benchmark is to test performance for such as unfavorable case. These type of patterns can be obtained for various reasons, two of which are: 1) Applications that perform transpose operation on large matrices need such far-away communication. 2) On many systems, job placement policies can allocate nodes of a given job far away in the system; as a result, a near-neighbor pattern manifests as a transpose pattern in this case.

In Figure 6.4 (left), we compare the execution time for three message size: 5 KB, 50 KB, and 250 KB for simulating Perm. At the smallest size, 5 KB, fat-tree provides the minimum execution time of 123 microseconds, followed by torus with 147 microseconds, and then by

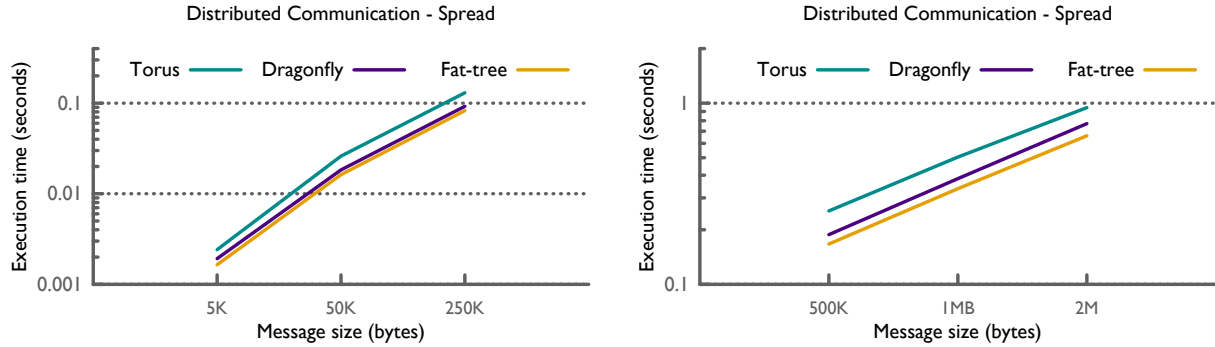


Figure 6.5: Communication performance of different networks for Spread. Fat-tree outperforms dragonfly by a small margin, while both of them are significantly faster than torus for all message sizes.

dragonfly at 167 microseconds. However, as the message size is increased by ten times to 50 KB, the execution time for torus increases proportionately by 10.5x, while dragonfly and fat-tree increases by 7.5x and 8x only. Hence, while fat-tree provides the minimum execution time of ~ 1 ms, the dragonfly outperforms torus by 20%. Thereafter, the increase in execution time is proportional to the increase in message size for all networks, with minor variations. At the largest message size (2 MB shown in Figure 6.4 (right)), the execution time on fat-tree is 45% less than torus, with dragonfly in between.

6.2.5 Distributed communication

The last communication pattern we study is a distributed communication pattern. In this pattern, each rank communicates with an arbitrary set of partners in such a way that obtaining good topology aware mapping is not feasible. Among other scenarios, this communication pattern provides an approximation for performance of a complete all-to-all communication pattern, which is often difficult to simulate. Applications that are load balanced based on compute load, or implement some form of distributed hash-tables, or use approximations for force computations from distant objects exhibit such a communication pattern. In our benchmark, called Spread, we arbitrarily select 15 – 20 partners for every rank to mimic the distributed communication pattern.

Figure 6.5 presents a comparison of the performance obtained when Spread is simulated on the three prototype systems using various message sizes. For all message sizes, fat-tree is the best performing network. It outperforms the dragonfly network by a small margin which varies from 10% to 15%. The growth in execution time is proportional to the increase in the message size for both these networks. Both these networks provide better performance

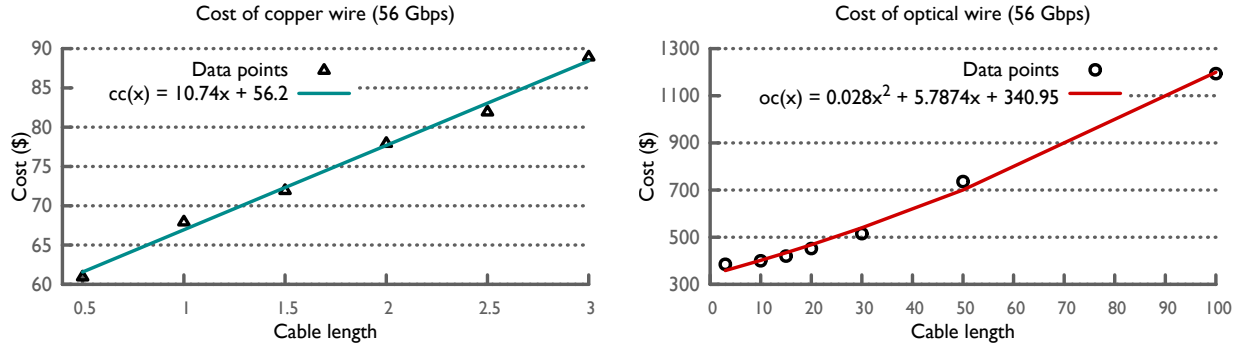


Figure 6.6: Cost model for copper and optical cables.

than torus, which also shows proportional increase in the execution time. For the smallest message size of 5 KB, the fat-tree and the dragonfly are better than torus by 32% and 20%, respectively. At the largest message size of 2 MB, the improvements for the former are 30%, while the latter provides 19% lower execution time in comparison to torus.

6.3 Network cost comparison

In addition to performance, cost of networks being built is also critical to their deployment. In this section, we create models to estimate and compare the cost of building networks for prototype systems of various sizes. To create these models, we first find the price of cables and routers available in the market¹. By performing regression on these values, appropriate models are obtained for estimating price of cables and routers based on different independent variables.

6.3.1 Cost models

Three types of connections are commonly used in building HPC networks: backplane-based, copper cables and optical cables. Since the cost of connections performed via backplane is extremely low, we ignore them in our models. Among others, copper cables are used for short distance connections (up to six meters or so), while optical are used for the rest. Other than the material used to build cables, their length plays an important role in determining their cost. Figure 6.6 shows change in price of 56 Gbps copper and optical cables based on their length. By using linear regression on the data shown on the left side of the figure, we

¹All prices obtained from Colfax as of Dec, 2015.

obtain the following function that estimates the cost of copper wire with an R^2 value higher than 0.99:

$$cost_{copper}(x) = 10.743x + 56.2 \quad (6.1)$$

where x is the length of cable.

Similarly, the right side graph in Figure 6.6 shows how the price of 56 Gbps optical cable changes with the length of the cable. In this case, very high value of R^2 is obtained by using a quadratic polynomial as shown below:

$$cost_{optical}(x) = 0.028x^2 + 5.787x + 340.95 \quad (6.2)$$

where x is the length of cable.

The simulation results presented in the previous section are based on 100 Gbps link bandwidth, which is expected to be used in the next generation systems. If the price of cables with 56 Gbps bandwidth is compared with the cost of cables with 100 Gbps bandwidth, we observe a 1.35x increase in the price of a given length. In our cost computation, we use this ratio to estimate the cost of cables with 100 Gbps bandwidth. Note that the model used here is developed based on cables with 56 Gbps because more data points are available for these cables; in contrast only 2-3 data points are available for 100 Gbps cables.

Next, we look at the price of routers, where the number of ports in the router affect the cost significantly. Figure 6.7 shows the variation in cost of a QDR router (with 40 Gbps bandwidth per port) and a FDR router (with 56 Gbps bandwidth per port) as the number of ports is varied. It can be seen that for both QDR and FDR routers, the price of the router is a linear function of the number of ports. In our study, we use the formula obtained by performing linear regression on the data for QDR router since the number of ports for these data points is closer to the values we have used in the prototype systems:

$$cost_{router}(x) = 140.11x + 646.11 \quad (6.3)$$

where x is the number of ports.

Again, a comparison of the cost of router with 40 Gbps per port capacity with a router with 100 Gbps per port capacity suggests a 2x increase in the price as the bandwidth is increased. Hence, we use this value to estimate the cost of routers in our analysis. It is to be noted though that we found a large variation in the cost of routers across different vendors and different per port bandwidths. As a result, the absolute values presented here may vary significantly based on the vendor and bandwidth chosen as base values.

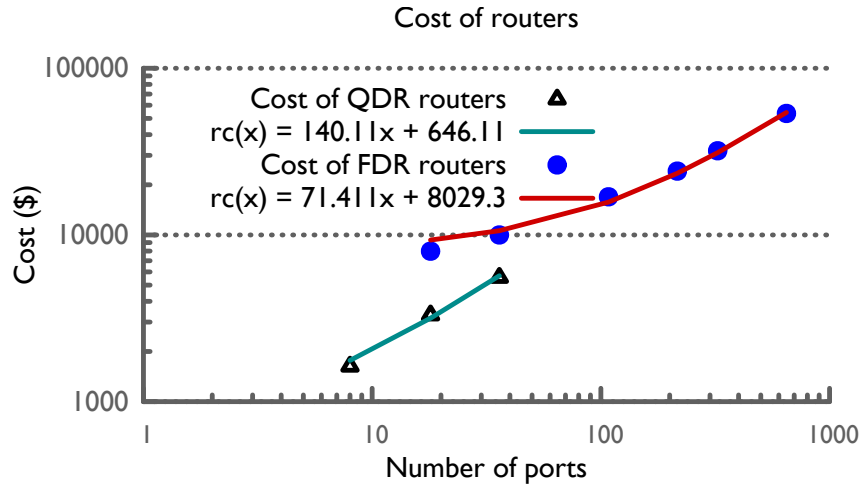


Figure 6.7: Cost model for routers.

6.3.2 Router and cable cost

Estimating the cost of router is straightforward. Based on the formula defined in Table 6.1, the number of routers and their radix can be computed. Cost of each of these routers is then computed using the model defined in the previous section. This value multiplied with the number of routers provide the estimated cost of routers. In Figure 6.8, this cost is shown in black as a bar chart. It can be seen that for all the networks, the router cost is a large value; it is in fact approximately 80%, 75%, and 85% of the total network cost for torus, dragonfly, and fat-tree topologies, respectively. In terms of absolute values, the total number of routers required to build torus networks is very large, and hence the total router cost is also very large. The number of routers required to build a dragonfly network is approximately double the number of routers needed by the fat-tree network, though the radix of the routers needed by dragonfly network is smaller. Hence the total router cost of a dragonfly network is only 25 – 30% higher than the total router cost of the corresponding fat-tree network. An interesting result to note is that as the number of nodes is increased by 10x, from 10 K to 100 K, the router cost for all the network also increases by a factor of ten.

Estimating total cable cost is more complex than estimating total router cost since cables of different lengths are used for connecting routers and nodes placed at different locations. We briefly describe the layout used for each of the networks based on which cable costs are computed.

Torus: For torus networks, we assume that each cabinet/rack consists of 128 nodes arranged

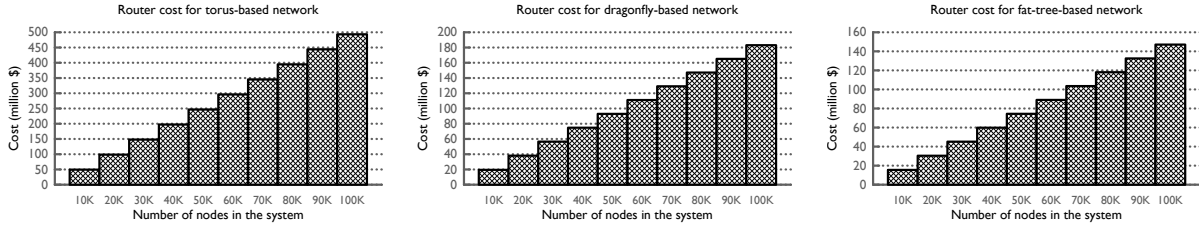


Figure 6.8: Estimated router cost for building networks for prototype systems based on different interconnect topologies.

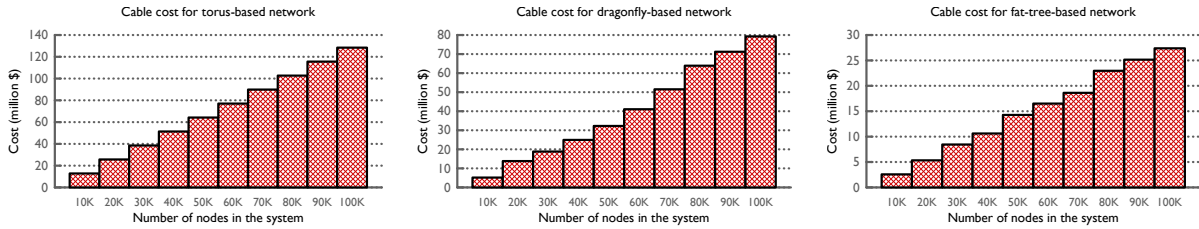


Figure 6.9: Estimated cable cost for building networks for prototype systems based on different interconnect topologies.

as a $4 \times 2 \times 2 \times 2 \times 2$ block. The size of the cabinet is estimated to be two meters high, with one meter depth and width. All the connections within this block are electrical, made using the backplane. At each of the 5-dimensional surfaces of this block, end points of the connections for the sixth dimension are added. Based on the total number of nodes, the number of these cabinets are computed; these cabinets are then arranged as a 2D-grid. The cabinets in the rows are placed next to each other without any gap, while the rows are placed at the distance of 1 m. Three out of six dimensions are connected within a row with an increasing cabinet distance, while the other three connections are made across rows along the column. This leads to the following connection set up in a row of cabinets: positive direction of first dimension is made to the adjacent cabinet (distance one), while the negative direction is made to the cabinet at distance two. The positive direction of the second dimension is made to the cabinet at distance three, while the negative direction is at distance six. Finally, the positive direction of third dimension is at distance seven, while the negative dimension is at distance 14. Similar set up is done along the column for the other three dimensions.

Dragonfly: Each group of the dragonfly network is placed in two cabinets that are situated right next to each other. This is done since it is hard to accommodate all the nodes that are part of a group in one cabinet. The cabinets are same size as the torus, and are arranged in exactly the same manner as the torus. Intra-group connections within a cabinet are supported via backplane, while connection across cabinets are copper based. All the inter-

group connections are made using optical cables that are connected via a 2D-grid set up on top of the cabinets.

Fat-tree: Using similar cabinets as for torus and dragonfly, multiple rows of cabinets are laid out. Each cabinet hosts three level 0 and level 1 routers, and the nodes connected to the level 0 routers. Each row consists of cabinets that contain routers which belong to a common level 0 cluster, i.e. we have as many rows as the number of level 0 clusters. The routers are rack mounted, and are connected to the nodes via copper cables. Level 0 to level 1 connections among routers are restricted to individual rows, while level 1 to level 2 connections can be restricted to individual columns if the level 2 routers are placed in the central row in a relatively sparse set up.

Based on the layouts described above, the computed cable cost is presented in Figure 6.9. The large number of router in the torus network also leads to a large number of cables. Hence the total cable cost for torus is very high. In contrast, for the fat-tree, although the router radix is high, the small number of routers leads to low total cable cost. For both these networks, the increase in the cable cost is proportional to the increase in the number of nodes. In contrast, for the dragonfly, the cable cost increases by 16x for a 10x increase in node count. This is partially due to increased radix as the systems become large, and partially due to the requirement for longer wires for the inter-group connections.

6.3.3 Total cost comparison

Figure 6.10 compares the estimated cost for building networks based on the torus, dragonfly, and fat-tree topologies. It is easy to see that the cost of building a torus network is much higher (2 – 3x) in comparison to the cost of dragonfly and fat-tree networks. As discussed earlier, this is primarily due to the high router count, which also leads to higher cable count in the torus network. The comparison among dragonfly and fat-tree is more interesting. At the lowest node count (10 K), the cost of building a dragonfly is 30% higher than the fat-tree. However, as the node count is increased, this difference increases to 50%. Within this 50%, only 20% increase is due to the router cost, which accounts for 70% of the total cost of the dragonfly. The cable cost increases much faster for the dragonfly, and accounts for the remaining 30% of the cost difference with the fat-tree.

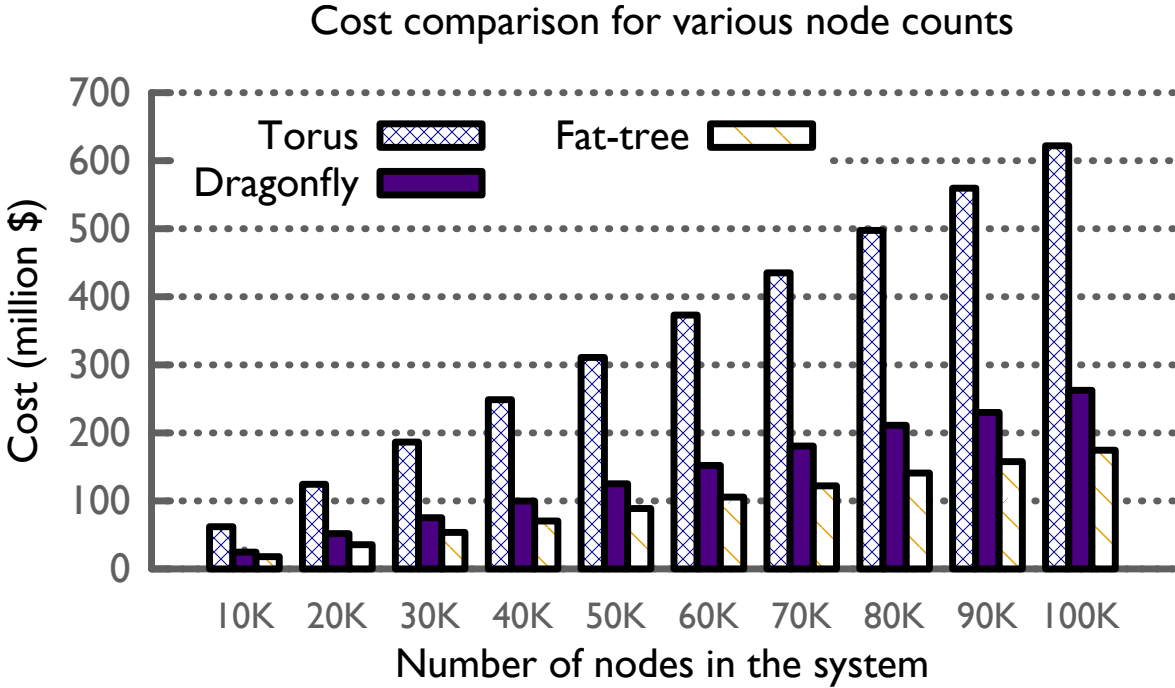


Figure 6.10: Comparison of estimated cost for building different networks for a given node count. Only router and cable cost are considered.

6.4 Performance Per Dollar

So far, we have compared the performance and the cost of the three network topologies (torus, dragonfly, and fat-tree) individually. In this section, we attempt to combine these two metrics together to provide a more insightful comparison of these topologies. The first step in this comparison is to compute the communication rate obtained when a communication pattern is simulated on a network. To achieve this, we find the total number of bytes that are communicated in a given simulation, and then divide it by the execution time to obtain bytes per second, which is the communication rate. As one would expect, we desire networks with higher communication rate. The communication rate is divided by the cost of the network to obtain a metric that is the focus on this section: *performance per dollar*.

4D Stencil and NN: Figure 6.11 presents a comparison of performance per dollar for the three networks when 4D Stencil is executed on them for six different message sizes. The corresponding performance results were presented in Figure 6.1. The first thing to observe is the difference in performance per dollar at the lowest message size (5 KB). At this message size, the performance of all the three network is same, but the difference in cost

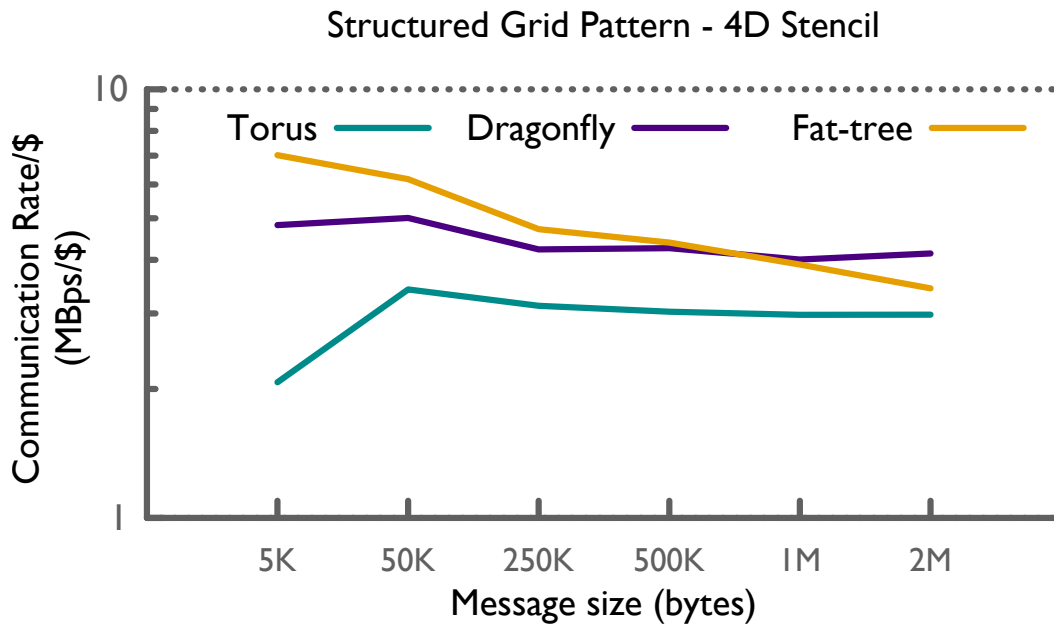


Figure 6.11: Although torus provides the best performance, its performance per dollar is worst among the three networks. As the message size increases, the superior performance of dragonfly leads to a better performance per dollar.

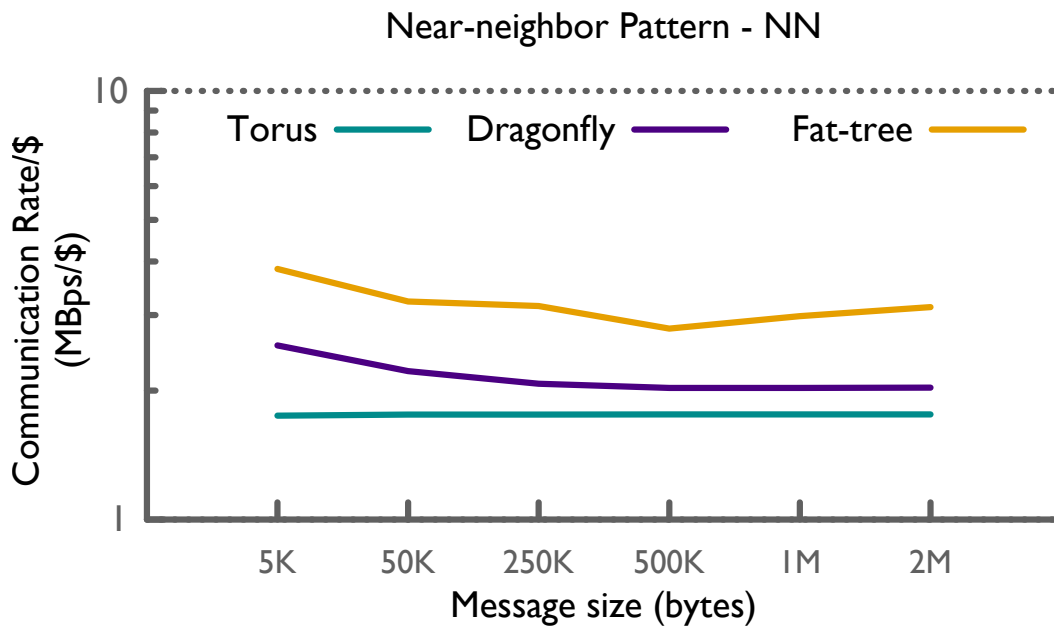


Figure 6.12: Fat-tree shows the best performance per dollar due to its low cost, although its performance is similar to dragonfly. Due to its superior performance, torus' performance per dollar is only 10% lower than the dragonfly.

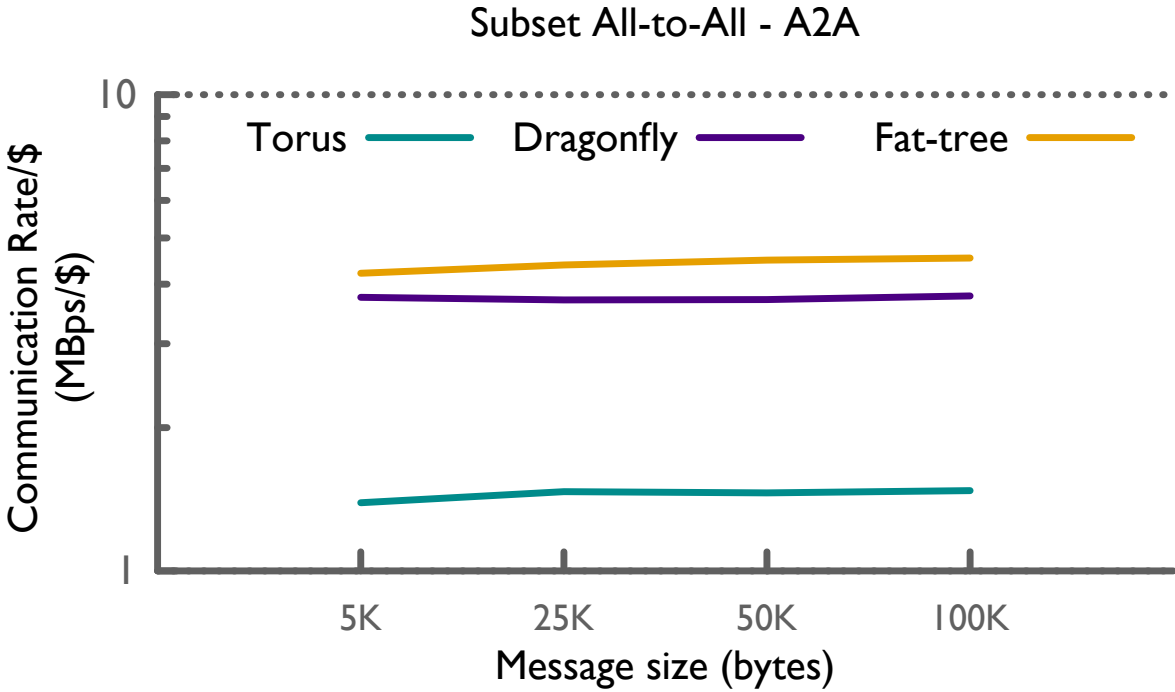


Figure 6.13: Given the similar performance of all networks, performance per dollar is significantly impacted by the cost of the networks.

leads to a large difference in the performance per dollar. As the message size is increased, although torus provides the best performance, its high cost leads to lowest performance per dollar value. The relative values of performance per dollar for dragonfly and fat-tree are interesting: as the message size increases, the superior performance of dragonfly compensates for its higher cost, and the performance per dollar for dragonfly eventually becomes better than the fat-tree.

For the near-neighbor communication executing on torus, the results shown in Figure 6.12 are similar to the results for 4D Stencil. The primary difference here is that due to its very high performance, the performance per dollar for torus is only 10% lower than the dragonfly. Although the performance of the fat-tree is similar to the dragonfly as shown in Figure 6.2, the lower cost of former results in a much higher performance per dollar.

A2A: Figure 6.13 compares the performance per dollar for execution of subset all-to-all for the four message sizes whose performance is compared in Figure 6.3. Due to similar performance of all the networks, performance per dollar is significantly impacted by the cost of the networks. As a result, the torus has performance per dollar values three times less than that of the fat-tree. Between the dragonfly and fat-tree, the performance per dollar of

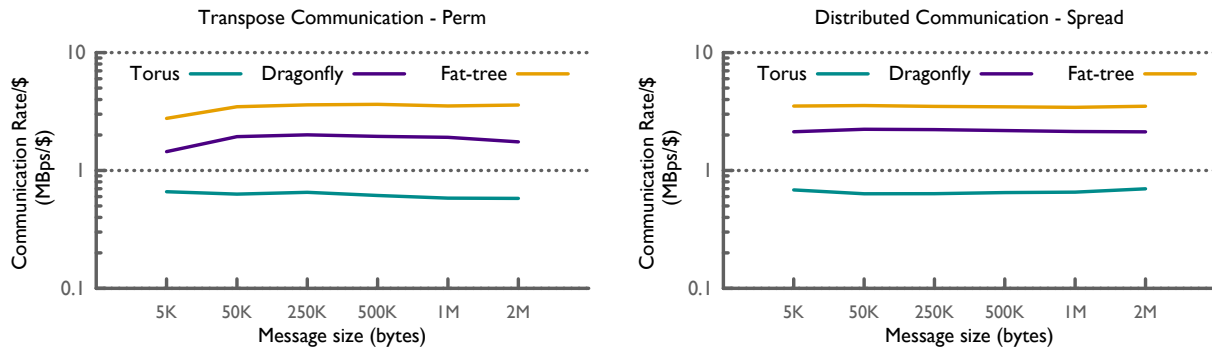


Figure 6.14: The performance difference among the three network is further enhanced by the cost difference. As a result, fat-tree show very high performance per dollar in comparison to the dragonfly, which in turn is much higher than the performance per dollar of torus.

dragonfly is approximately 17% less than the fat-tree. Note that this difference is lower than the relative difference among the cost of these networks because of the better performance of the dragonfly network.

Perm and Spread: Both the transpose and distributed communication patterns lead to communication among ranks that are typically far apart. As shown in Figures 6.4 and 6.5, the fat-tree and the dragonfly outperform the torus network for these patterns. Figure 6.14 shows that this difference is further increased by the higher cost of torus network. In fact, the performance per dollar for the fat-tree is six times the performance per dollar of the torus, although the performance slowdown due torus is 30 – 50% only. Similar higher differential is seen among fat-tree and dragonfly: the performance per dollar for dragonfly is 60% of the performance per dollar of the fat-tree network.

6.5 Summary

In this chapter, we have presented a comprehensive comparison of three network topologies that are used in supercomputers of today: torus, dragonfly, and fat-tree. Performance comparison using five distinct communication patterns shows that different networks provide better performance when different interaction patterns are executed on them. Figure 6.15 (left) summarizes one such comparison set via a spider chart that uses the communication rate obtained for fat-tree as the base value. It can be see that torus provides best performance for patterns such as Stencil and Near-neighbor. In contrast, fat-tree performs the best for transpose and distributed communication pattern. The dragonfly provides intermediate

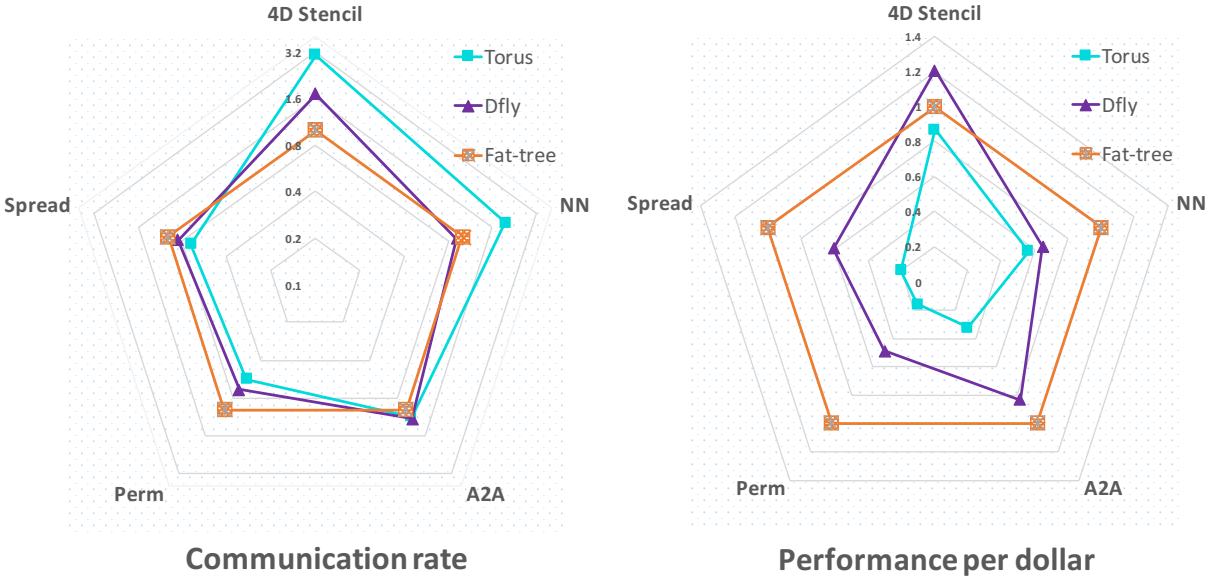


Figure 6.15: Summary of the communication rate and performance per dollar for large message sizes. Plotted values are normalized using the values for the fat-tree.

performance across all the benchmarks.

Following the performance comparison, we analyzed the cost of building networks using the three topologies. Here, we found that cost of building torus is much higher (2–4x), while fat-tree based networks can be built in a much cheaper manner. Finally, we compared the three networks for a metric that combines the performance and cost metrics. Figure 6.15 (right) provides a brief summary of results for performance per dollar metric that were discussed in this chapter. It can be seen that the performance per dollar values are highest for the fat-tree for most cases, except for the 4D Stencil benchmark. This suggests that fat-tree is the most economical topology for building networks on which the aforementioned communication patterns are executed.

Impact of Configuration on Performance

A network simulator, such as TRACER (Chapter 5), provides us with the capability to simulate *what-if* scenarios of various types. In this chapter, we make use of this capability to study the relation between configuration changes and the communication performance. At a very high level, this experimentation entails predicting the execution time when a given benchmark is run on similar systems, but with incremental modifications to the system configuration. By analyzing the results from these experiments, we hope to understand the impact of changing a given network configuration, and thus develop a better understanding of the method to create systems with good network configuration. While the mechanism being used in this chapter generally applies to all applications, we use two mini-applications from Chapter 6: 4D Stencil and Spread. These two benchmarks have been used because of the difference in their characteristics. As for the network topologies, we focus on the torus and the dragonfly topologies, though similar studies can be conducted with other network topologies too.

7.1 Stencil with unbounded resources

In the first set of experiments, we simulate torus and dragonfly topologies, assuming practically unbounded resources. The systems used in these experiments consist of $\sim 24,576$ nodes, with 16 control flows (ranks) on each node. For the torus, the nodes are arranged as a 5D torus of dimensions $8 \times 8 \times 12 \times 16 \times 2$, which is similar to the topology of Vulcan, a Blue Gene/Q at LLNL. The dragonfly network consists of 201 groups with 20 routers each. Each router has 6 nodes attached to it. The bandwidth of inter-group links is higher than the intra-group links to balance the traffic at each router. The stencil grid is divided

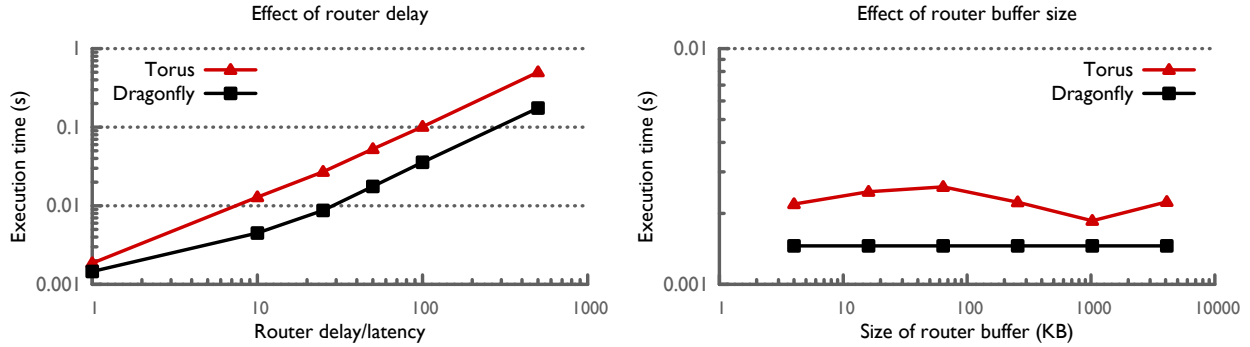


Figure 7.1: (left) When all other resources are practically unlimited, the communication performance is directly proportional to the router delay/latency. (right) Size of router buffer has no effect on the performance of the dragonfly network. Its impact on the torus network can be significant, but is hard to model.

among ranks arranged as a $72 \times 96 \times 96 \times 144$ grid. Six network configurations are varied in these experiments: router delay/latency, link bandwidth, injection bandwidth, buffer size on routers, routing policy, and injection policy. For four of these parameters, the following values are chosen to make them the *non-bottleneck* resource: 0 for delay, 1000 GBps for link bandwidth and injection bandwidth, and 1,048,576 bytes for the router buffers.

Figure 7.1 (left) shows the impact of changing the router delay/latency on execution of 4D Stencil on the torus-based and dragonfly-based prototype systems. These executions are performed with other configurations being set at their *non-bottleneck* values as mentioned above. It is easy to see that when other resources are not the bottleneck, the execution time varies linearly with the router delay. This clearly suggests the need for minimizing the router delay in order to reduce the execution time. In contrast, the impact of the size of the router buffer is uncertain. For the dragonfly network, when all other resources are practically not the bottleneck, the execution time does not depend on the size of the router buffer. However, the execution time on the torus network varies from 1.8 ms to 2.5 ms when the size of router buffer is changed. The trend though is hard to understand. The lowest execution time is obtained for router buffer size of 1 MB, while the 64 KB sized buffers lead to highest execution time. We believe these differences are due to the way adaptive routing reacts to the injection of the traffic generated by 4D Stencil on the torus network.

Next, we analyze the impact of changing either the link or injection bandwidth, while the other is kept at a high value of 1000 GBps. Figure 7.2 (left) shows the results for simulating 4D Stencil with different link bandwidths when the router latency is set to zero. For both dragonfly and torus networks, as the link bandwidth is increased, the execution time decreases linearly. However, at very large values of link bandwidth, while the execution

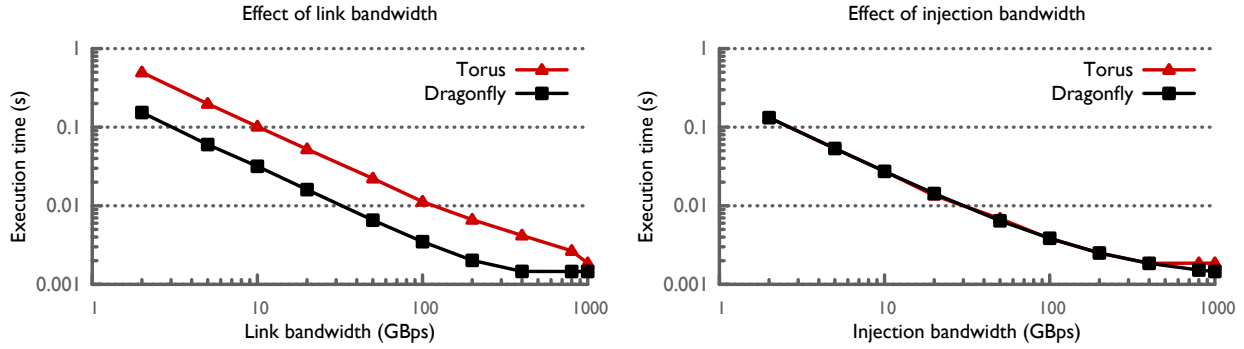


Figure 7.2: As the link or injection bandwidth is increased, the execution time drops linearly. While the dragonfly network saturates at the link bandwidth of 400 GBps, the torus network shows performance improvement till 1000 GBps. In contrast, the dragonfly network provides performance improvement till 1000 GBps when injection bandwidth is increased, but the torus network saturates at 400 GBps.

time decreases for the torus network, the performance saturates on the dragonfly network. This suggests that the dragonfly network can do with lower link bandwidth relative to the torus network. In contrast, the situation is reversed for changes in the injection bandwidth as shown in Figure 7.2 (right). For large injection bandwidth, dragonfly network continues to provide performance improvement but the torus network saturates at 400 GBps.

Figure 7.3 (left) shows the impact of changing both link bandwidth and injection bandwidth simultaneously, i.e. when the link bandwidth is set to x , the injection bandwidth is also set to x . As expected, based on the results observed for changing only link or injection bandwidth, the execution time decreases significantly as the link and injection bandwidths are increased. However, while the torus shows significant gains for large values of the bandwidths (e.g. 33% reduction when increasing the bandwidths from 400 GBps to 800 GBps), the improvement for dragonfly are low. Less than 20% improvement in execution time is observed for doubling the bandwidths above 200 GBps for the dragonfly network.

In order to understand the results better, we now compare the three trends we have presented above: impact of changing link bandwidth only, injection bandwidth only, and both link and injection bandwidth. Figure 7.4 presents this comparison for the torus on the left, and for the dragonfly on the right. It can be seen that for torus, the performance numbers are much better when only injection bandwidth is low, i.e. if the link bandwidth is high, the effect of low injection bandwidth is observed less. In contrast, for the dragonfly, performance for all three cases is relatively close. For low bandwidth values, the link bandwidth is the primary bottleneck. Moreover, limited injection bandwidth in addition to limited link bandwidth leads to worse performance. As the bandwidths are increased, the injection bandwidth

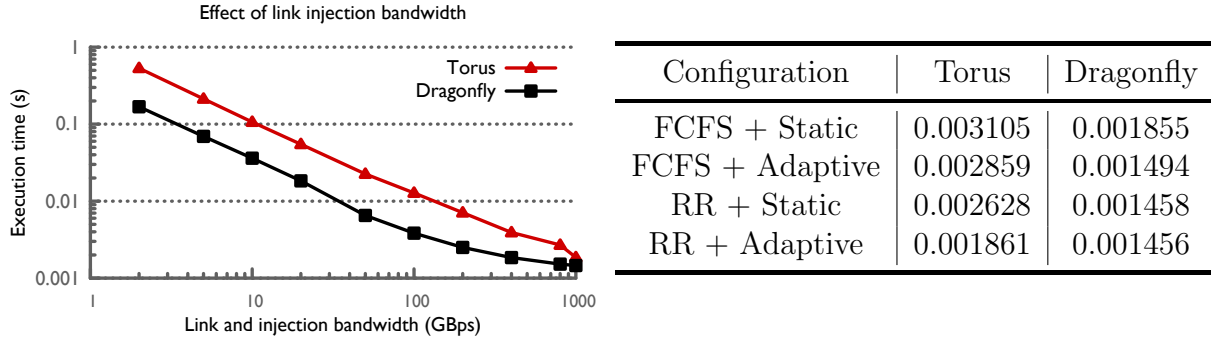


Figure 7.3: (left) Impact of changing both link and injection bandwidth on execution time. For both the networks, significant improvement in performance are observed; for dragonfly, the relative improvement reduces as the bandwidths are increased to very large values. (right) Impact of routing policy and injection policy. When other configuration parameters are not the bottleneck, both policies impact the observed performance on torus; in contrast, on a dragonfly, good choice of one makes the other irrelevant.

becomes the primary bottleneck with negligible impact due to limited link bandwidth.

Finally, we discuss the impact of routing and injection policy using the results presented in Figure 7.3 (right). Here, FCFS refers to the scheme in which the NIC injects packets from messages using a first-come-first-serve ordering. Packets from the next message are not injected till the previous message has been sent entirely. On the other hand, RR stands for the injection policy in which messages are selected in a round-robin manner to create the next packet. For the routing, Static represents the dimension-ordered shortest path routing for torus and minimal-path routing for the dragonfly. Adaptive routing signifies dynamic shortest-path routing for torus and non-minimal hybrid routing for dragonfly (Section 5.3).

Very different outcomes are observed for torus and dragonfly networks in these results. For torus, use of Adaptive routing results in 10% and 30% improvement in performance for FCFS and RR injection policy, respectively. This is because Adaptive routing is able to distribute network traffic onto more links irrespective of the injection policy. For the RR policy, the improvements are higher because packets targeted to different destinations are available for the Adaptive routing to send. This diversity in destinations provides more freedom to the routing policy, which can then better balance the load on different links. In contrast, for the dragonfly network, making a better choice for one of the routing or injection policy leaves the other choice unimportant. Use of Adaptive routing results in an improvement of 20% over the Static routing when FCFS is the injection policy. Similar improvement is observed if RR policy is used instead of FCFS with Static routing. However, using Adaptive routing with RR injection policy does not provide any benefit. This suggests that the RR policy

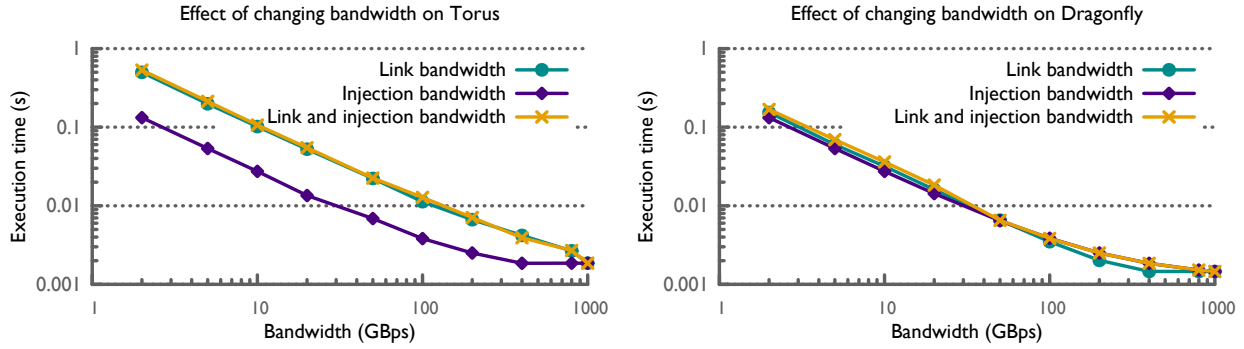


Figure 7.4: (left) For torus, link bandwidth acts as a primary bottleneck, with injection bandwidth requirement saturating at large values. (right) For dragonfly, both link and injection bandwidth are critical, with link bandwidth being more important at lower bandwidth and injection bandwidth being the bottleneck at larger values.

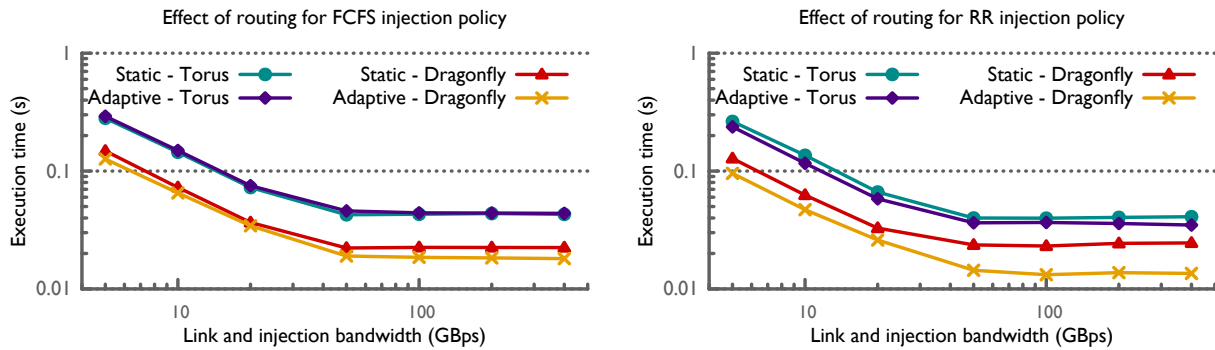


Figure 7.5: (left) For torus, the impact of routing is minimal for FCFS injection policy, when resources are limited. Adaptive routing improve performance for dragonfly, especially for large bandwidth. (right) With RR injection policy, Adaptive routing provides significant performance improvement for both torus and dragonfly networks.

provide sufficient randomization of destinations and thus routes, which is the goal of the Adaptive routing on dragonfly.

7.2 Stencil with practical resources

Having studied the impact of various network configurations on performance of 4D Stencil when availability of most resources is practically unlimited, we now focus on scenarios in which most resources are limited. These scenarios are closer to real systems which have limitations on most of the network resources, and thus potentially have multiple causes of performance bottlenecks.

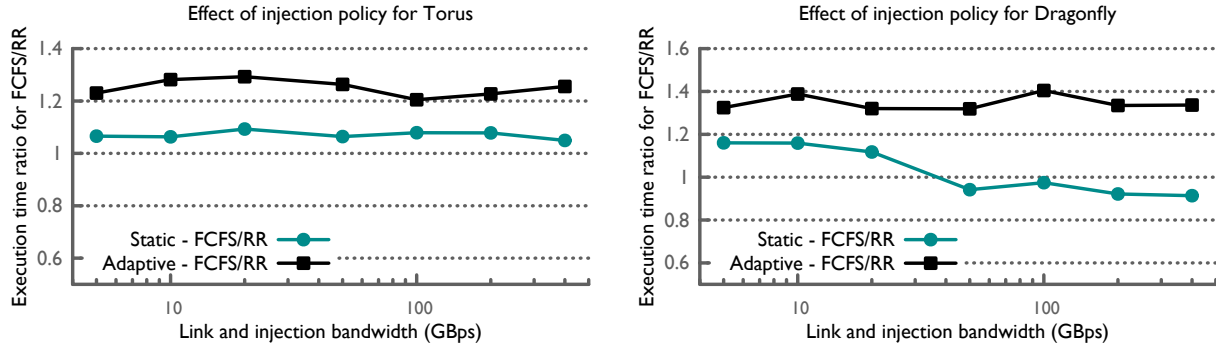


Figure 7.6: (left) On torus, using RR injection policy is highly beneficial when Adaptive routing is used. (right) When the link and the injection bandwidths are the primary bottlenecks, RR provide significant performance improvement on a dragonfly.

Figure 7.5 presents the impact of changing routing policy from Static to Adaptive for different link/injection bandwidths; the router latency is kept constant at 30 ns for these simulations. The x-axis in these plots is the link/injection bandwidth. On the left side, we see that the impact of routing policy is minimal for execution on torus with FCFS injection policy. This is because FCFS injection policy produces many packets targeted at the same destination before the next message is packetized; as a result, when a good mapping scheme is used as in our simulations, the numbers of hops and possible paths in torus are limited. In contrast, we see that the routing scheme impacts the performance on dragonfly networks where hybrid routing transmits packets from the same message via many different paths.

On the right hand side of Figure 7.5, the impact of routing for RR injection policy is shown. Here, we find that for both torus and dragonfly, use of Adaptive routing provides significant performance gains. Presence of packets destined to multiple targets is a big factor that helps Adaptive routing provide these benefits. Some of these results are also in contrast to the results we observed for the previous case, in which most network resources were unbounded. This is probably because presence of limited resources creates a congestion scenario, which is handled in a better way by the Adaptive routing.

Figure 7.6 shows the same results as Figure 7.5, but focuses on comparing the impact of injection policy on performance. The left graph shows that, on torus, the impact of injection policy is low when Static routing is used. However, with adaptive routing, the RR injection policy leads to significant performance improvements. As discussed earlier, this is because Adaptive routing is able to redistribute the network load better when RR provides it with multiple packets targeted at distinct destinations. For the dragonfly network, the graph on the right shows similar results for Adaptive routing. Up to 40% reduction in execution time is observed when RR injection policy is used instead of FCFS. For the Static routing and

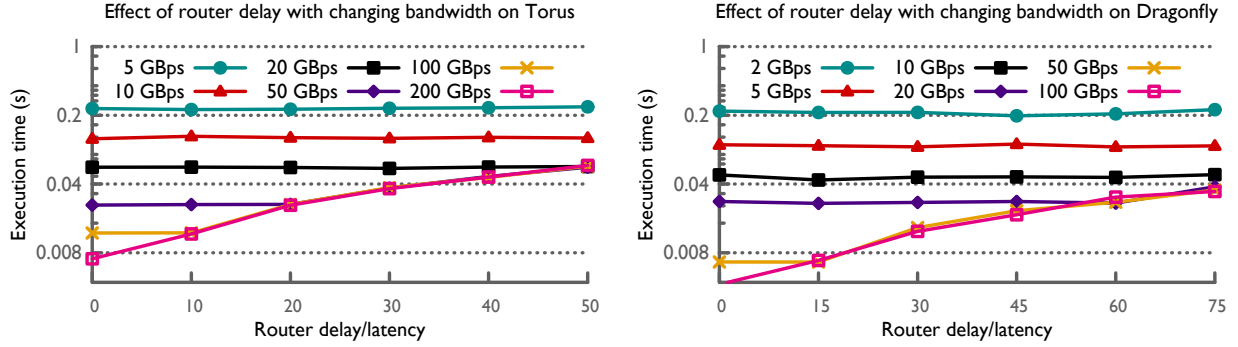


Figure 7.7: As the router delay increases, the bottleneck changes from link and injection bandwidth to the fixed delays. Conversely, for a fixed router delay, as the link/injection bandwidth decreases, the fixed delays cease to be the performance bottleneck.

smaller values of bandwidth, decent improvement in performance is also observed. However, as the link and injection bandwidths are increased, other network configurations (e.g. router delay) become the bottleneck. As a result, the improvement due to RR slowly diminishes.

Next, we analyze the impact of changing router delay/latency for scenarios in which the available link and injection bandwidth is realistic (unlike in the previous section where we assumed a 1000 GBps bandwidth). Figure 7.7 presents the impact of changing router delay/latency for different bandwidth values. For both torus and dragonfly, we observe that as the link and injection bandwidth is increased, the performance bottleneck shifts from one configuration to another. For low bandwidth values, such as 2 – 10 GBps, the link and injection bandwidths are the bottleneck. Since our packet size choice is 1024 bytes, this is understandable since at these bandwidths, the packet transmission time (100 – 500 ns) dominates the communication cost. As the bandwidth is increased further, the transmission time is comparable or even less than the router delay. Thus, the link and injection bandwidth cease to have an impact on the performance and the execution time is determined by the latency. From these results, it is clear that for 4D Stencil with large messages, the impact of router delay and link/injection bandwidth do not add up; rather a *maximum* function is what determines which of these resources is the performance bottleneck.

Finally, we present results that show an interplay of three network configurations on performance: link bandwidth, injection bandwidth, and router delay. Figure 7.8 shows the predicted execution time for four scenarios on the torus network: link bandwidth of 5, 20, 50, and 100 GBps. For each of these cases, the injection bandwidth is varied from 5 GBps to 100 GBps and router delay is varied from 0 ns to 50 ns. It can be seen that when link bandwidth is low, it determines the execution time. For those cases, latency has minimal impact and injection bandwidth is less important if its reasonably high. At very high link bandwidth,

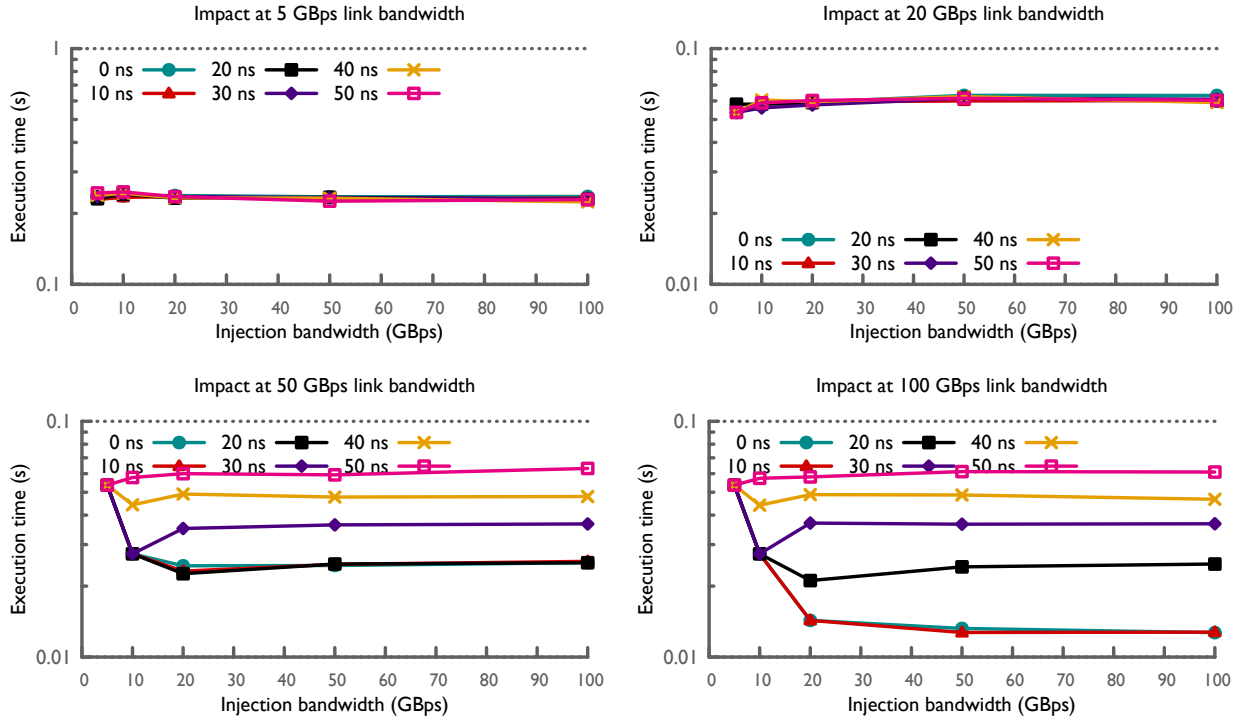


Figure 7.8: Effect of variations in the link bandwidth, injection bandwidth, and latency on execution time of 4D Stencil on torus. Increasing link bandwidth reduces the execution time, but if injection bandwidth is much lower, it limits the performance. When both link and injection bandwidth are high, very high latency can be the performance bottleneck.

e.g. 50 and 100 GBps, both injection bandwidth and router latency are important. Very low values of injection bandwidth limit the performance for cases in which router delay is low. However, as the router delay increases, the performance is entirely bound by it.

Figure 7.9 presents the results for changing link bandwidth, injection bandwidth and router delay for the dragonfly network. While many trends are similar to the ones found for torus, e.g. high router delay being the bottleneck when bandwidth is high, there are a few significant differences. First, for many scenarios, increasing injection bandwidth for a fixed link bandwidth and router delay helps reduce the execution time. The trend is clearly visible for the case with 0 ns latency for all link bandwidths, and a few other router delays. However, there are also many cases in which an increase in injection bandwidth leads to significant decrease in the performance (e.g. 75 ns router delay curves for link bandwidth 20, 50, and 100 GBps). These are most likely due to the *local* view of the Adaptive routing. As packets are injected at higher rate than the rate at which the router can transmit them, sub-optimal decisions are made for selecting the best possible routes for the packets. This is because in these scenarios the routers are forced to act based on stale local information about

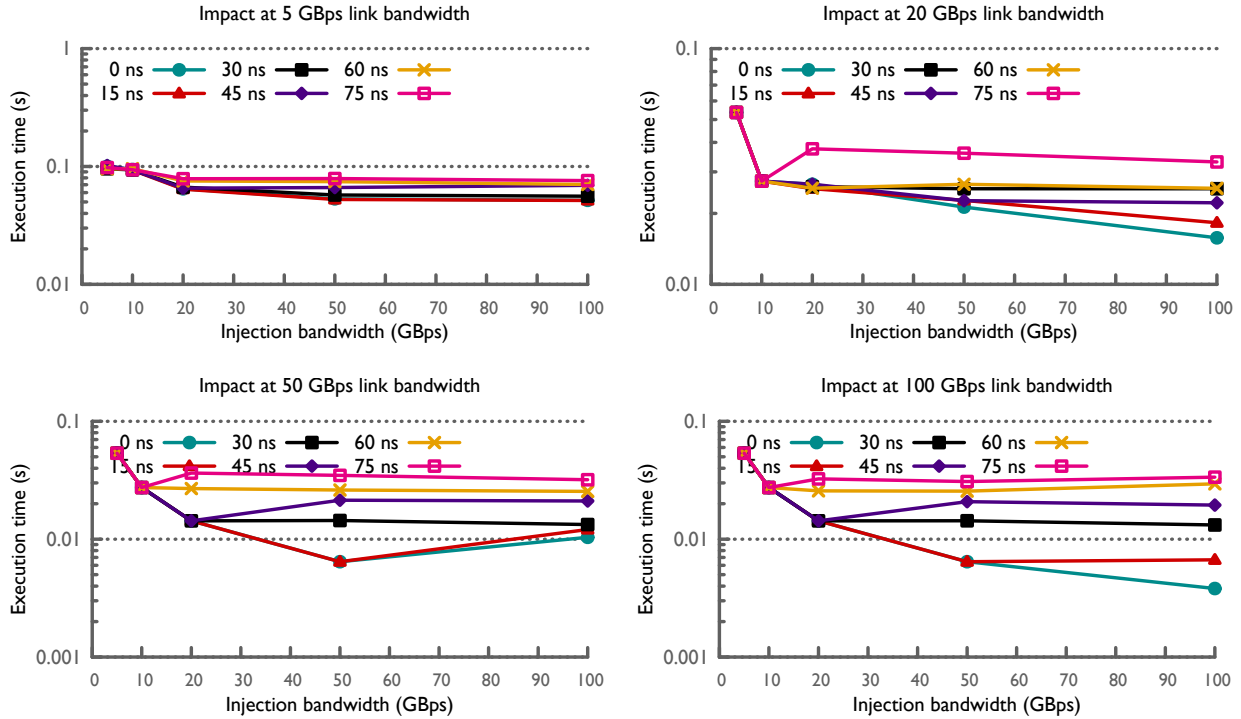


Figure 7.9: Unlike torus, an increasing injection bandwidth impacts the execution time both positively and negatively, even when link bandwidth is low. Similarly, the router latency has impact on the performance even when the link and injection bandwidth are relatively low.

the load at its connected partner routers. The second main differentiating fact about these results, in comparison to torus, is the impact of router delay even at low link bandwidths. We suspect this is because low router delays help the Adaptive routing in making marginally better use of multiple routes when injection bandwidth is high.

7.3 Spread with unbounded resources

Having analyzed how network configurations impact the execution of 4D Stencil on torus and dragonfly, we now conduct a similar study with a very different communication pattern, Spread. The prototype systems used in these experiments are the same ones as used in the previous section, and consist of $\sim 24,576$ nodes, with 16 control flows (ranks) on each node. For the Spread communication pattern, each rank communicates with 15 – 20 randomly chosen partners using messages of size 2 MB. The packet size used in these simulations is same as earlier, i.e. 1,024 bytes.

Router delay/latency is the first network configuration analyzed in this section, assuming

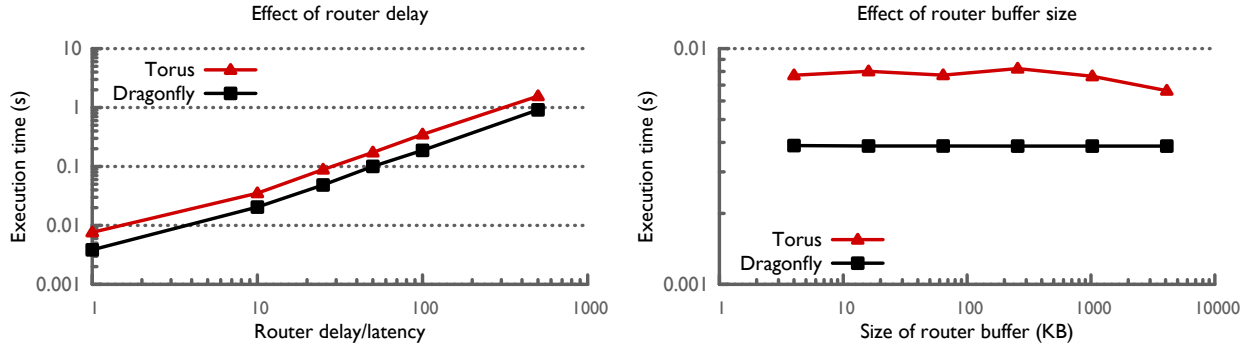


Figure 7.10: (left) The execution time increases as the router latency is increased; performance of the torus is similar to the dragonfly. (right) As for 4D Stencil, size of router buffer has no effect on the performance of the dragonfly network. Its impact on the torus network is significant, but does not follow a pattern.

other configurations are set at their practically non-bottleneck values. Figure 7.10 (left) shows that the execution time closely follows the changes in the router delay. However, unlike 4D Stencil where a linear model can precisely depict the trend, for Spread, the increase in execution time is sometimes higher and sometimes lower than predicted by a linear model. We believe this is because Spread causes significant congestion at the bisection links. Hence, the observed performance varies by a small margin based on the efficacy of the Adaptive routing at the given latency.

Figure 7.10 (right) presents the impact of the size of the router buffer with the router delay set to 0 ns. As with 4D Stencil, the size of router buffer has no impact on the execution for the dragonfly network. For the torus network, the size of network buffer does impact the performance, but it is difficult to find a model that accurately predicts the impact. The lowest execution time is obtained when router buffer size is 4 MB, while the highest time is obtained for 256 KB sized buffers. We believe these differences are due to the way adaptive routing reacts to the flow of traffic generated by Spread on the torus network.

The simulation results obtained by modifying link and injection bandwidths, individually and together are shown in Figure 7.11 and Figure 7.12 (left). The trends observed here are very similar to the ones observed for 4D Stencil. As the link or injection bandwidth is increased, the execution time reduces significantly. However, when the link bandwidth is increased beyond 200 GBps, performance saturates for dragonfly, but shows a consistent improvement for torus. The trends are reversed when injection bandwidth is increased beyond 400 GBps, i.e. the dragonfly network continues to see improvements while the torus network saturates at 400 GBps.

Figure 7.12 (left) shows the impact of changing both link bandwidth and injection band-

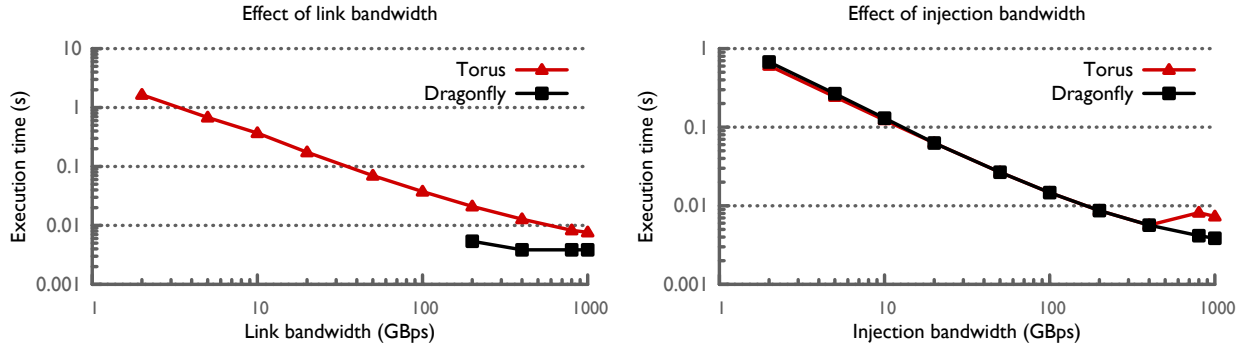


Figure 7.11: The execution drops almost linearly as the link or injection bandwidth is increased. The dragonfly network saturates at link bandwidth of 400 GBps, but the torus network observes good performance improvement till 1000 GBps. The reverse is true for injection bandwidth.

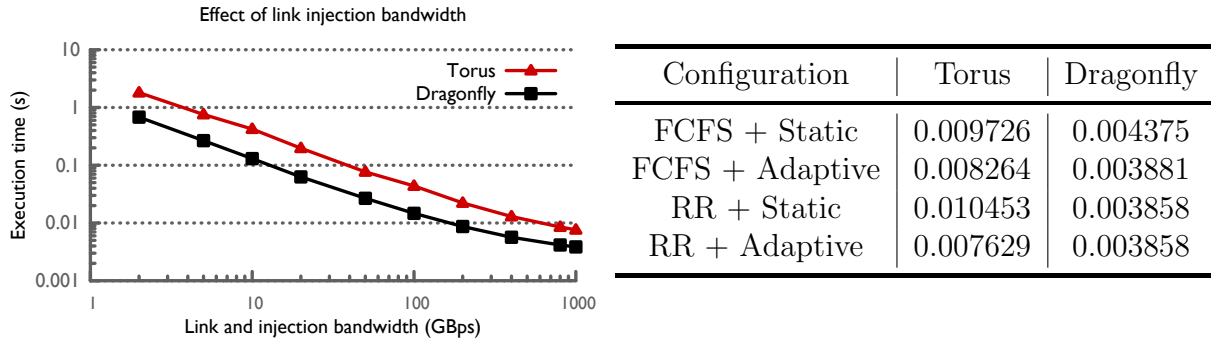


Figure 7.12: (left) For both torus and dragonfly, significant improvement in performance is observed, even for very large bandwidth. (right) Impact of routing policy and the injection policy: RR + Adaptive provides the best performance on torus, while only FCFS + Static performs badly on dragonfly.

width simultaneously on execution time of Spread. In these experiments, when the link bandwidth is set to x , the injection bandwidth is also set to x . For both torus and dragonfly, increasing the value of both link and injection bandwidth provides better performance. As opposed to 4D Stencil where the torus obtained higher improvements for large values of bandwidths, both torus and dragonfly show similar relative improvement. This is because Spread is more communication intensive, and is thus able to use the additional bandwidth to improve performance on dragonfly also.

We end this section with a discussion on the impact of routing and injection policy on performance of Spread. As shown in Figure 7.12 (right), very different trends are observed for torus and dragonfly networks. For torus, use of Adaptive routing provides 15% and 25% improvement in performance for FCFS and RR injection policy, respectively. These gains

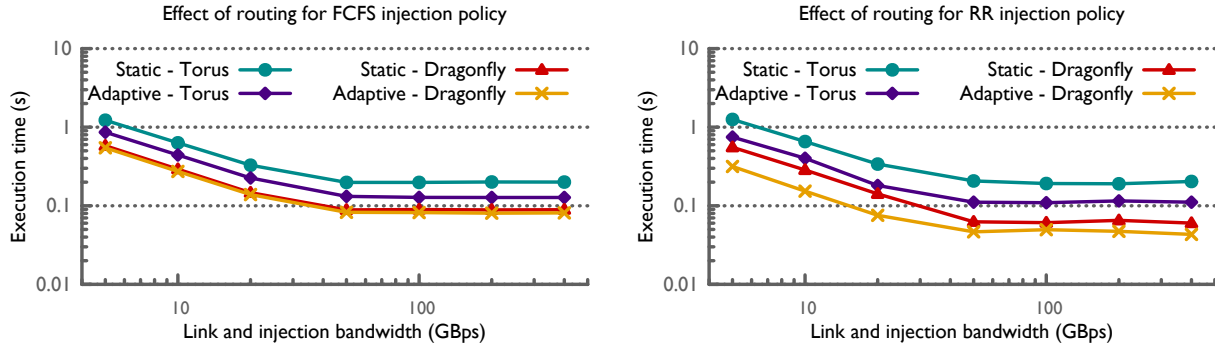


Figure 7.13: (left) Adaptive routing provides significant improvement for torus, but has minimal impact on the dragonfly network . (right) With RR injection policy, Adaptive routing provides significant performance improvement for both torus and dragonfly networks.

are similar to what we observed for 4D Stencil, and are probably due to the same reasons. However, use of RR policy with Static routing results in a performance drop of 7%. As was the case with 4D Stencil, for the dragonfly network, making a better choice for one of routing or injection policy leaves the other choice unimportant. Use of Adaptive routing results in an improvement of 12% over the Static routing when FCFS is the injection policy. Similar improvement is observed if RR policy is used instead of FCFS with Static routing. The improvement are lower in comparison to 4D Stencil because Spread pattern provides a reasonable variation in message destinations because of its characteristics, thus resulting in good performance with Static routing.

7.4 Spread with practical resources

We now switch our focus to analyzing impact of various network configurations on performance of Spread when each of the resources can be a potential bottleneck. As one can imagine, this is the more common case and thus a better indicator of the impact of various network configurations.

The impact of routing policy is presented in Figure 7.13 for FCFS and RR injection policies. For the FCFS policy (left graph), the Adaptive routing provides significant performance improvement for the torus network, while showing minimal impact on dragonfly. These results are in contrast to the results for 4D Stencil where the Adaptive routing improves performance for the dragonfly network only. This is because of the placement of the communicating neighbors in the two benchmarks: in 4D Stencil, the communicating processes are typically placed on physically close nodes which provides fewer links for the

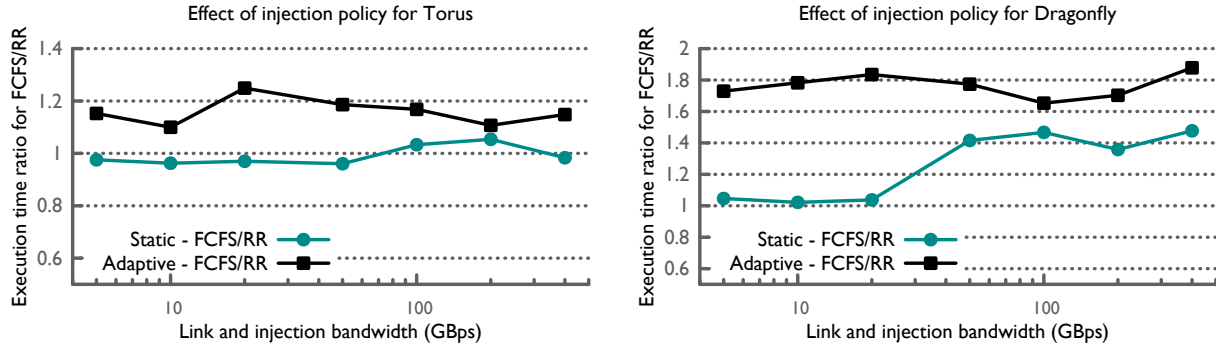


Figure 7.14: (left) The RR injection policy provides better performance when Adaptive routing is used, but FCFS shows similar performance for Static routing. (right) On the dragonfly network, performance improvements are high for Adaptive routing and for Static routing if the bandwidth is high.

Adaptive routing to utilize on torus. However, the random spread of neighbors in Spread ensures that Adaptive routing has more links to utilize on torus and even Static routing is able to distribute traffic more evenly for the dragonfly network. Thus, Adaptive routing helps torus, while does not improve performance much for dragonfly.

For the RR injection policy, the factors described above continue to benefit Adaptive routing on torus as shown in Figure 7.13 (right). For the dragonfly network, Adaptive routing also helps when the injection policy is RR. However, the gains are lesser when the bandwidth is high. We suspect this is because when bandwidth is low, the Adaptive routing can disperse the packets created from different messages in an even manner, and thus utilize the network better. While the Static routing also works on a similar well distributed set of destinations, it does not make use of many links in the system. At higher bandwidth, other resources become the performance bottleneck, and thus the impact of routing is low.

Next, we compare the performance of different injection policies in Figure 7.14. On torus, the results are similar to what we observed for 4D Stencil: the impact of injection policy is minimal for Static routing, while it is significant for Adaptive routing. As discussed earlier, this is because presence of packets targeted at different destinations is helpful to the Adaptive routing in making use of more links. RR policy also improves performance significantly when Adaptive routing is used on dragonfly. We believe this is because RR policy helps make use of many more links with packets targeted at a much larger set of nodes, thus reducing the hotspots. For the Static routing, the trend is reversed in comparison to 4D Stencil: RR provides no benefit when bandwidth is low, but leads to significant improvement when bandwidth is high. We are currently unsure of the reasons that lead to such a behavior.

Figure 7.15 shows the change in execution time as the router delay is increased for different

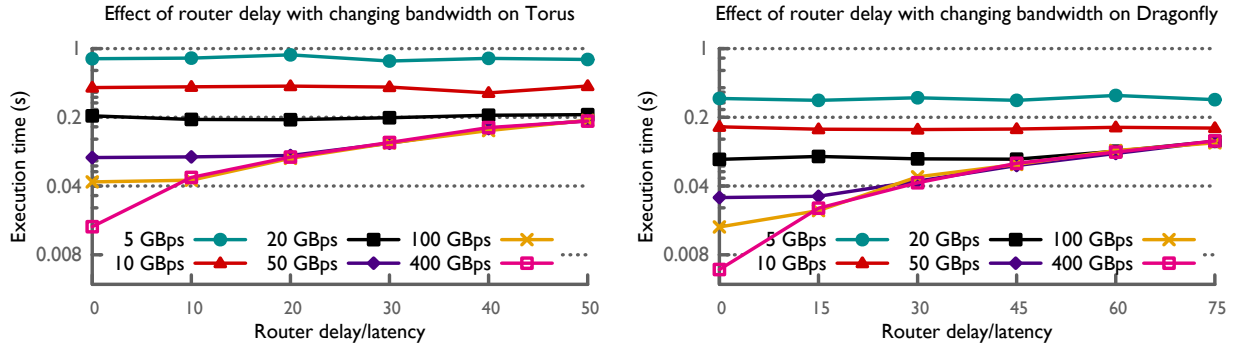


Figure 7.15: For low bandwidth, the router delay does not impact the execution time, but as the router delay increases, it becomes the performance bottleneck.

bandwidths. The general pattern found in these results is similar to the one observed for 4D Stencil (Figure 7.7). For both torus and dragonfly, we observe that as the link and injection bandwidth is increased, the performance bottleneck shifts from one configuration to another. As was the case for 4D Stencil, it is clear that the effects of router delay and link/injection bandwidth do not add up; rather a *maximum* function is what determines which of these resources is the performance bottleneck.

The last set of results presented in this chapter study the relationship between link bandwidth, injection bandwidth, and router delay. Figure 7.16 shows the predicted executed time for four scenarios: link bandwidth of 5, 20, 50, and 100 GBps. For each of these cases, the injection bandwidth is varied from 5 GBps to 100 GBps and router delay is varied from 0 ns to 50 ns. As the link bandwidth is increased, we observe a significant drop in the execution time, though if the injection bandwidth is low, the execution time is high even with high link bandwidth. At low delays, torus shows low requirement for injection bandwidth (than link bandwidth) to obtain its best performance. As the delay is increased at high link bandwidth, the behavior is different from the one observed in 4D Stencil. While in 4D Stencil, we observe a linear increase in execution time when link/injection bandwidth is high and router latency is increased, the increase in execution time for Spread is significantly less. This is because at very high link bandwidth, the adaptive routing is able to make use of alternate routes in parallel and hence reduce the impact of increased router latency. This is not possible in 4D Stencil because of near-neighbor placement of message destinations.

Figure 7.17 presents the results for changing link bandwidth, injection bandwidth and router delay for the dragonfly network. These results are very similar to the one obtained for 4D Stencil (Figure 7.17), but are significantly different from the torus. While the execution time decreases as the link bandwidth is increased, the best performance is obtained when injection bandwidth is also increased proportionally or by a higher margin. When the

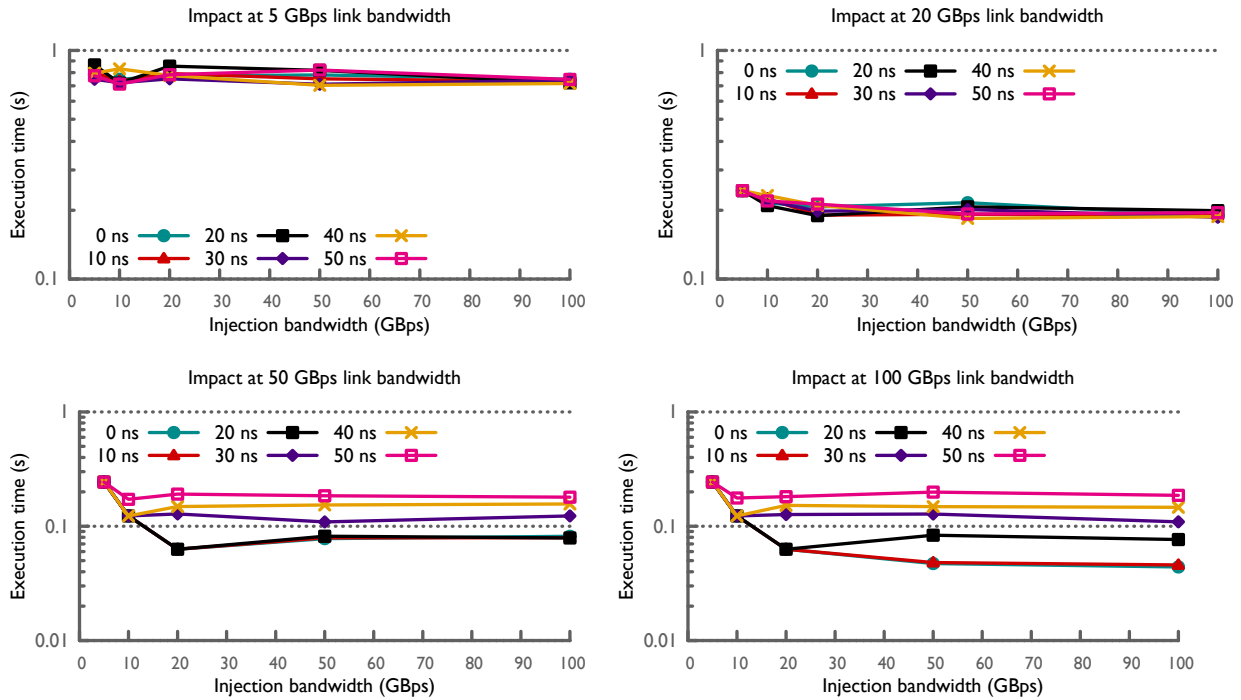


Figure 7.16: Performance improves with increasing link bandwidth, but saturates quickly when injection bandwidth is increased. Impact of latency is less prominent in comparison to 4D Stencil.

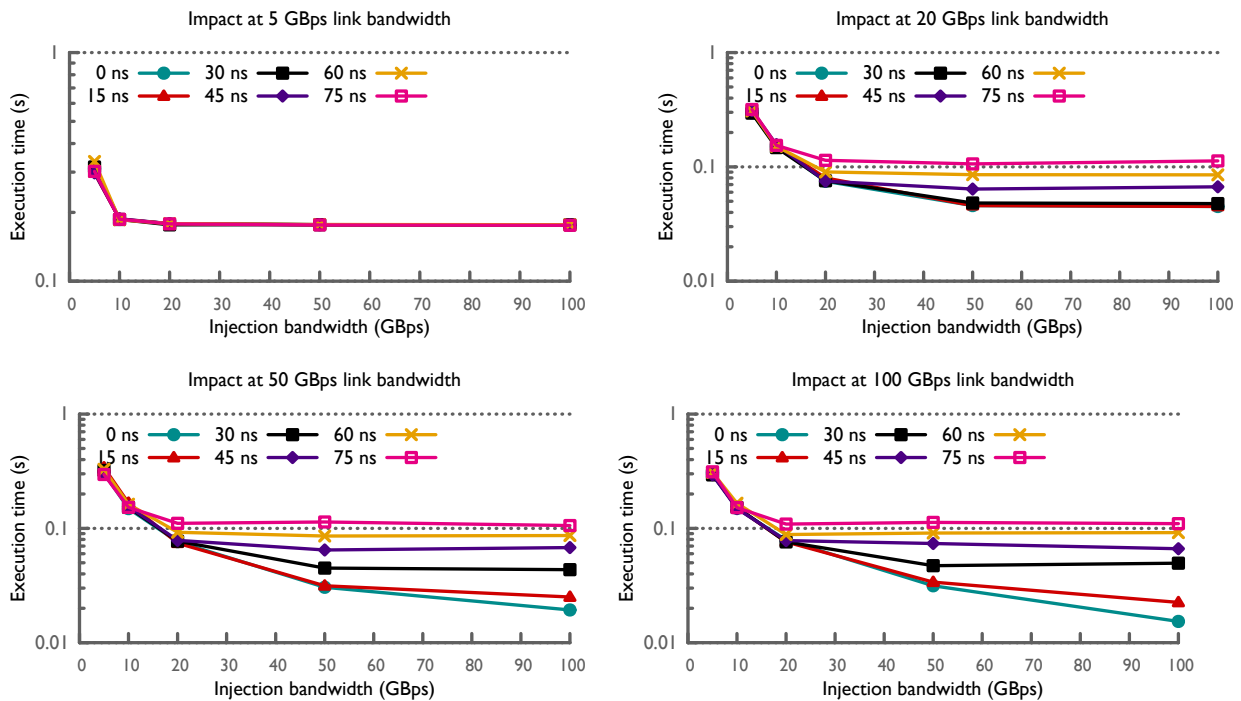


Figure 7.17: An increasing injection bandwidth impacts the execution time positively, even beyond the link bandwidth value. The impact of the router latency is prominent and proportional to the delay.

combined effect of link and injection bandwidth is high and saturates, the router latency shows up as the bottleneck. The increase in execution time is linearly related to the increase in the delay as shown in the graphs for 50 and 100 GBps link bandwidths.

7.5 Summary and discussion

Parametric evaluation of what-if scenarios for networks being used for building large scale systems can be a powerful method to design networks that suit the requirements of an HPC center and its users. This chapter has presented a novel way of conducting and analyzing such an evaluation. For two commonly used networks, torus and dragonfly, we have performed an in-depth analysis of impact of various network configurations for two distinct communication patterns. By further expanding such an evaluation to more networks and communication patterns, we plan to create a comprehensive report (and possibly develop prediction models) on behavior of several communication patterns on various networks.

From the results presented in this chapter, the following conclusions can be drawn about torus and dragonfly networks for the two communication patterns we have studied, 4D Stencil and Spread:

- Dragonfly provides best performance when injection bandwidth is higher than link bandwidth, while torus saturates at lower injection bandwidth.
- Round-robin injection policy in combination with Adaptive routing typically provides better performance than the FCFS policy.
- In order to obtain the best performance, tradeoffs should be studied among router delays, link bandwidth, injection bandwidth, and packet size.

Communication Algorithms

8.1 Analysis of collectives on dragonfly networks

Many scientific applications use data movement collectives such as *Broadcast*, *Scatter*, *Gather*, *Allgather*, *All-to-all*, and computation collectives such as *Reduce*, *Reduce-scatter*, and *Allreduce* [124]. The performance of these MPI collectives is critical for improved scalability and efficiency of parallel scientific applications. In recent years, there have been an increasing number of applications such as web analytics, micro-scale weather simulation and computational nanotechnology, that involve processing extremely large scale data requiring collective operations with large messages. Performance of such large message collectives is significantly affected by network bandwidth constraints.

On networks with small radix, such as torus, transmitting packets from a source to destination involve traversal through a large number of nodes/switches. The multiplicity of hops makes these networks congestion prone, especially when performing collectives on large data/messages. To counter the effects of congestion, carefully designed topology aware algorithms have been used for collectives on such networks [64, 66]. In addition, there is a set of topology oblivious algorithms which perform reasonably well on most systems [125–127]. How well are these algorithms suited to dragonfly networks? This section attempts to answer this question, and proposes a new set of topology aware algorithms for collectives on dragonfly networks for large messages. These new algorithms are designed to exploit the high radix routers and the multi-level structure of a dragonfly network. A cost model based on the link utilization is used to evaluate the effectiveness of proposed algorithms in comparison to general topology oblivious algorithms.

8.1.1 Cost model and assumptions

We assume an in order mapping of MPI ranks onto the cores in the system. Consider a system with sn groups/supernodes, each consisting of nps (nodes per supernode) nodes with cpn (cores per node) cores each. Hence, we have $p = sn * nps * cpn$ cores whose in order mapping is performed as following. Consider a global numbering of supernodes from 0 to $sn - 1$. Within a supernode and a node, nodes and cores are locally numbered from 0 to $(nps - 1)$ and from 0 to $(cpn - 1)$ respectively. In the global space, cores are numbered (by MPI) from 0 to $(p - 1)$ using the core's supernode, node and position within the node as the key. For example, cores in supernode 0 get ranks from 0 to $(nps * cpn - 1)$. Following the cores in supernode 0, cores in supernode 1 get ranks from $nps * cpn$ to $(2 * nps * cpn - 1)$ and so on.

Further, we assume a dragonfly network with round robin connections at the second level (L2 links) to connect supernodes. A connection from supernode $S1$ to supernode $S2$ originates at node $(S2 \text{ modulo } nps)$ in supernode $S1$. This link connects to node $(S1 \text{ modulo } nps)$ in supernode $S2$. Therefore, each node is connected to $spn = \frac{sn}{nps}$ supernodes. We consider the case in which job allocation onto nodes and supernodes happen in a uniform manner. To keep things simple, we assume the cases where the entire machine is being used by an application. Algorithms and results for the other case, where allocation is not uniform, can be derived with minor variations and will be discussed in a future work.

As we focus on large message collectives, we use a bandwidth based model to estimate the cost of a collective algorithm. The start up cost and latency effects are ignored as the bandwidth term dominates for large messages. We assume that the time taken to send a message between any two nodes is $n\beta$, where β is the transfer time per byte, if only 1 link is being used to send n bytes of data. In case of a computation operation, we add a γ computation cost component per byte. We also use a two step approach to find link utilization which provides a more accurate estimate of performance of an algorithm. In the first step, given a collective operation, the algorithm to use, number of MPI ranks or cores and the data length information (required by the operation), *pattern-generator* generates a list of communication exchange between every pair of MPI ranks. The data generated by *pattern-generator* is fed to *linkUsage*. Given a list of communication exchange, *linkUsage* generates the amount of traffic that will flow on each link in the given network.

Operation	Algorithm	Cost (n bytes)
Scatter	Binomial Tree	$\frac{p-1}{p}n\beta$
Gather	Binomial Tree	$\frac{p-1}{p}n\beta$
Allgather	Ring, Recursive Doubling	$\frac{p-1}{p}n\beta$
Broadcast	DeGeijn's Scatter with Allgather [127]	$2\frac{p-1}{p}n\beta$
Reduce-Scatter	PairWise Exchange	$\frac{p-1}{p}(n\beta + n\gamma)$
Reduce	Rabenseifner's Reduce-Scatter with Gather [125]	$\frac{p-1}{p}(2n\beta + n\gamma)$

Table 8.1: Commonly used algorithms.

Topology oblivious algorithms

Table 8.1 lists the algorithms which are generally used to perform various collective operations in a topology oblivious manner for large message sizes. Many of these algorithms are used in MPICH as the default option [126].

8.1.2 Two-tier algorithms

Given the clique property and the multiple levels of connections, the dragonfly networks naturally leads to a new set of algorithms which we refer to as *two-tier algorithms*. The common idea in any two-tier algorithm is stepwise dissemination, transfer or aggregation (SDTA) of data. SDTA refers to simultaneous exchange of data within a level in order to optimize the overall data exchange. Performing SDTA ensures that the algorithms use maximum possible links for best bandwidth, and collate information to minimize the amount of data exchanged at higher levels. Without loss of generality let us assume that the root of any operation is core 0 of node 0 of supernode 0. In our discussion, we use core to refer to any entity which takes part in the collective operation. An MPI process and Charm++ chore are examples of such entities.

Scatter and Gather

Scatter is a collective operation used to disseminate core specific data from a source core to every other core. The two-tier algorithm for Scatter using SDTA is as follows:

1. Core 0 of node 0 of supernode 0 sends data to core 0 of every other node in supernode 0. The data sent to a core is the data required by the cores residing in the supernodes connected to the node of that core.
2. Core 0 of every node within supernode 0 sends data to core 0 of every node outside

supernode 0 that the node is connected to. The data sent to a node is the data required by the cores in the supernode to which this destination node belongs.

3. Core 0 of every node that has data (including node 0 of supernode 0) sends data to core 0 of every other node within its supernode. This data is required by the cores within the node that the data is being sent to.
4. Core 0 of every node shares data, required by the other cores, with all other cores in their node.

The four step process described above implies that the source core first spreads the data within its supernode. The data is then sent to exactly one node of every other supernode by the nodes that received the data. Thereafter, nodes that have data to be distributed within their supernode spreads the data within their supernodes. Gather can be performed using this algorithm in the reverse order.

For collectives with personalized data for each core such as Scatter, the dissemination of data can also be done using direct message send. The data will take exactly the same path as described in the above scheme. We have described our approach using Scatter because of its simplicity, and ease of understanding.

Broadcast

Broadcast can be performed using the approach used for Scatter if the entire data, without personalization, is sent in the four steps. We refer to this type of Broadcast as *base broadcast*. However, using the following scheme better performance can be obtained.

1. Core 0 of node 0 of supernode 0 divides the data to be broadcasted into nps chunks and sends chunk i to core 0 of node i of supernode 0.
2. Core 0 of every node within supernode 0 sends data to core 0 of exactly one node outside supernode 0 that the node is connected to. Exactly one node is chosen to avoid duplication of data delivery in following steps.
3. Core 0 of every node that received data in the previous step sends data to core 0 of every other node within their supernode.
4. Core 0 of all the nodes that received data in Step 2 and Step 3 send data to core 0 of all other nodes outside their supernode that they are connected to.
5. Now, these cores share data with core 0 of all other nodes in their supernode.

6. Core 0 of every node shares data with all other cores in their node.

This algorithm begins with the source core dividing the data into chunks, and distributing it within its supernode (as if performing Scatter over a limited set of cores). In the second step, every node in supernode 0 share the chunk with exactly 1 node outside their supernode. Thereafter, the nodes that received the chunk in the previous step share the data with other nodes in their supernode. As a result, all nodes in some of the supernodes have a chunk of initially divided data which needs to be sent to other supernodes. This is done in the next step, following which all nodes, which have received a chunk so far, share these chunks with other nodes in their supernode.

Allgather

An Allgather operation is equivalent to Broadcast being performed by all cores simultaneously. The SDTA based algorithm begins with all cores within every node exchanging data and collecting it at core 0 of the node. In the second step, all nodes within a supernode exchange data in an all-to-all manner using L1 links, and thus every node in every supernode contains the data which a supernode wants to broadcast to other supernodes. In the following step, supernodes exchange data in an all-to-all manner in parallel. Finally the nodes which receive data in the previous step disseminate this data to other nodes within its supernode. In addition, core 0 of every node has to share this data with all other cores in its node. This algorithm can be seen as a base broadcast being done by all nodes simultaneously (refer to Section 8.1.2).

Please note that in many cases, multiple steps of SDTA can be performed by a send from the source of one step to eventual destination of the following step. An example case will be when core 0 of node 0 of supernode 0 has to send data to core 0 of nodes that are connected to other nodes of supernode 0. We have presented them as separate steps in which initially core 0 of node 0 sends the data to core 0 of other nodes of supernode 0. These nodes then forward the data to core 0 of nodes of other supernodes. This has been done only for ease of understanding, and comparison results will not reflect them.

Computation Collectives

Although the same two-tier approach presented in the previous section can be used to perform computation collectives such as Reduce, it may not result in the best performance. The inefficiency in the previous approach derives from the fact that computation collectives

require some computation on the incoming data, and therefore if some node receives a lot of data from multiple sources, the computation it has to perform on the incoming data will become a bottleneck. We assume that the multiple cores do not share memory, and hence will not be able to assist in the computation to be performed on the incoming data. Also, the presented algorithms assume commutative and associative reduction operation.

Let us define an owner core as the core that has been assigned a part of the data that needs to be reduced. This core receives the corresponding part of the data from all other cores and performs the reduction operation on them. Consider a clique of k cores on which a data of size m needs to be reduced, and be collected at core 0. The algorithm we propose for such a case is the following:

1. Each core is made owner of $\frac{m}{k}$ data - assume a simple rank based ownership.
2. Every core sends the data corresponding to the owner cores (in their data) to the owner cores.
3. The owner cores reduce the data they own using the corresponding part in their data, and the data they receive.
4. Every owner core sends the reduced data to core 0.

Essentially, what we are doing is a divide and conquer strategy. The data is divided among cores, and they are made responsible for reduction on that data. Every core divides their data, and sends the corresponding portion to the owner cores. The owner cores reduce the data, and eventually send it to core 0.

Reduce - The above strategy can be used in multiple stages to perform the overall reduction in a two-tier network:

1. Perform reduction among cores of every node; collect the data at core 0.
2. Perform reduction among nodes of every supernode - owners among nodes are decided such that instead of collecting data at node 0, the data can be left with the owner nodes and directly exchanged in the next step. This may require a node to be owner of scattered chunks in the data depending on the supernode connections.
3. Perform reduction among supernodes and collect the data at supernode 0.

Reduce-Scatter - We can use the same algorithm as above to perform Reduce-scatter with a minor modification. Since the Reduce-scatter requires the reduced data to be scattered over all cores, in the last phase of reduction (i.e. reduction among supernodes), we decide

Operation	Base Cost	Two Tier Cost
Scatter	$\frac{p-1}{p}n\beta$	$n\beta * \max\{\frac{1}{nps}, \frac{1}{sn}\}$
Gather	$\frac{p-1}{p}n\beta$	$n\beta * \max\{\frac{1}{nps}, \frac{1}{sn}\}$
Allgather	$\frac{p-1}{p}n\beta$	$n\beta(\frac{1}{nps} + \frac{1}{sn} + \frac{1}{sn*nps})$
Broadcast	$2\frac{p-1}{p}n\beta$	$n\beta(\frac{3}{nps})$
Reduce-Scatter	$\frac{p-1}{p}(n\beta + n\gamma)$	$n\beta(\frac{1}{nps} + \frac{1}{sn} + \frac{1}{sn*nps}) + 2n\gamma$
Reduce	$\frac{p-1}{p}(2n\beta + n\gamma)$	$n\beta(\frac{1}{nps} + \frac{2}{sn}) + 2n\gamma$

Table 8.2: Cost model based comparison.

owners of data such that a supernode becomes owner of the data which its cores are required to receive in a reduce-scatter. Thereafter, instead of collecting all data at supernode 0 in the final step, the algorithm scatters the data within every supernode as required by Reduce-scatter.

8.1.3 Analysis of collectives

This section presents a comparison of the topology oblivious algorithms with the two-tier algorithms. The prototype dragonfly network that has been analyzed for these comparisons consists of 64 supernodes. Each supernode consists of 16 nodes each of which has 16 cores. The given configuration implies that there are 4,032 L2 links (inter-supernode connections) and 15,360 L1 links (intra-supernode connections) in the system. Note that we ignore the time spent in sharing data within a node by the cores.

Table 8.2 compares the two-tier algorithms with other algorithms using the cost model mentioned in Section 8.1.1. Among the data collectives, for Scatter and Gather, we observe that the two-tier algorithms which distributes data using all L1 links simultaneously within a source supernode provides theoretical speedup of factor nps i.e. nodes per supernode. This speedup may be affected by sn , i.e., the number of supernodes. If there are too few L2 links, they may become the bottleneck, and the speedup hence is bounded by $\min\{nps, sn\}$. For Allgather, we find that the speedup provided by two-tier algorithms depends on both sn and nps . For Broadcast, which happens in three phases, the theoretical speedup is $\frac{nps}{3}$. Finally, for computation collectives, we observe that our approach leads to more computation being performed. This is because the reduction happens in two phases and some computation, which could have been avoided, is performed. However, as with data collectives, the speedup for data transfer is substantial and should mask the effect of increase in computation.

	Scatter		Broadcast	
	Binomial	Two-tier	DeGeijn	Two-tier
L1 Links Used	1036	960	1588	15360
L1 Links Min Traffic	1 MB	1 MB	2 MB	64 MB
L1 Links Max Traffic	141 MB	64 MB	1.1 GB	128 MB
L2 Links Used	56	63	95	3937
L2 Links Min Traffic	16.7 MB	1 MB	32 MB	64 MB
L2 Links Max Traffic	520 MB	16 MB	1.09 GB	64 MB

Table 8.3: Link usage comparison for Scatter and Broadcast.

Scatter, Gather and Broadcast

We consider a Scatter operation in which the root sends 64 KB data to each of the remaining cores. In Table 8.3, we present a comparison of binomial algorithm link utilization with the two-tier algorithm. The important thing to note in the comparison is the maximum load binomial algorithm puts on a link in comparison to what two-tier algorithm puts. For L1 links, we find that two-tier algorithm puts a maximum load of 64 MB whereas binomial algorithm performs much worse, and puts a load of 141 MB. The difference is much more significant when it comes to L2 links where binomial algorithm puts a factor 32 times more load. Exactly same results are found for Gather operation due to its inverse nature to Scatter.

We also present the link utilization statistics for a 1 GB Broadcast in Table 8.3. Link utilization improves substantially both in terms of number of links used and the load which is put on links when two-tier algorithm is used. We expect an order of magnitude improvement in the execution time as the worst case link load goes down from 1.1 GB to 128 MB.

Allgather

As mentioned earlier, we study the performance of Allgather using two algorithms - recursive doubling and ring. The amount of data that each MPI rank/core wants to send is 64 KB. In Table 8.4, we present a comparison of two-tier algorithm with the recursive doubling and ring algorithms. It can be seen that while two-tier algorithm uses all the available L1 and L2 links in the system, the other two algorithms use a very small fraction of available links. Moreover, the load which two-tier algorithm puts on the links is orders of magnitude smaller in comparison to the other algorithms. It strongly suggests that the two-tier algorithm will outperform the other two algorithms. These results also conforms with the fact that for large messages, ring algorithm is better than recursive-doubling [126].

	Recursive Doubling	Ring	Two-tier Algorithm
L1 Links Used	10496	1080	15360
L1 Links Min Traffic	16 MB	1 GB	65 MB
L1 Links Max Traffic	15.1 GB	1 GB	65 MB
L2 Links Used	384	634	4032
L2 Links Min Traffic	4.2 GB	1 GB	16 MB
L2 Links Max Traffic	4.3 GB	1 GB	16 MB

Table 8.4: Link usage comparison for Allgather.

	Reduce-Scatter		Reduce	
	Pairwise Exchange	Two-tier	Rabenseifner	Two-tier
L1 Links Used	15360	15360	15360	15360
L1 Links Min Traffic	2 GB	65 MB	2 GB	66 MB
L1 Links Max Traffic	2 GB	65 MB	3 GB	130 MB
L2 Links Used	4032	4032	4032	4032
L2 Links Min Traffic	4 GB	16 MB	4 GB	16 MB
L2 Links Max Traffic	4 GB	16 MB	5 GB	32 MB

Table 8.5: Link usage comparison for Reduce-scatter and Reduce.

Computation collectives

In Table 8.5, we present a comparison of link utilization for Reduce-scatter and Reduce. For this experiment, the overall reduction size is 1 GB, and hence each core receives 64 KB reduced data when Reduce-Scatter is performed. We observe an order of magnitude difference in the load put on the links by two-tier algorithms in comparison to other algorithms. This can be attributed to the step wise manner in which two-tier algorithms perform reduction. Only the necessary data go out of a node or a supernode, and hence two-tier algorithm reduces the load put on the links significantly. Given this large difference in communication load, two-tier algorithms should outperform most other algorithms despite the additional computational load they put on the cores.

8.2 Charm-FFT

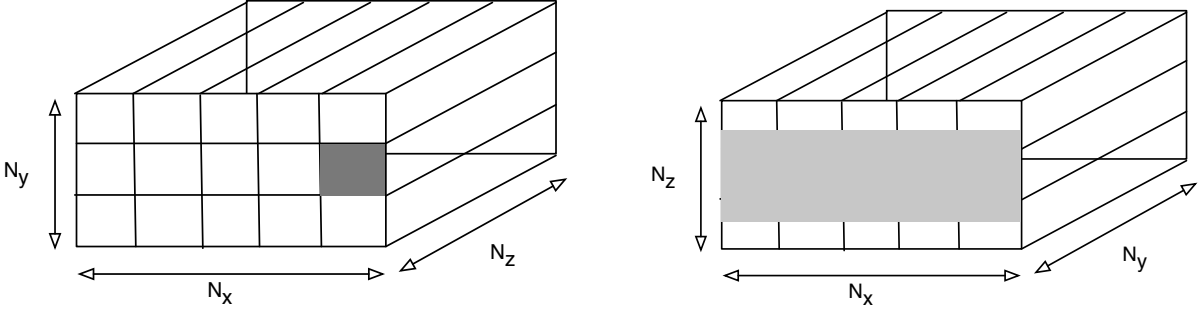
In many applications, parallel FFT, especially that of a distributed 2D or 3D grid of data, is often a critical bottleneck that limits their scalability. The communication heavy all-to-all pattern of parallel FFT is one of the primary reason for the limited scalability of FFTs. Moreover, parallel FFTs are commonly performed based on a 1D-decomposition of data.

Hence, the amount of parallelism available is limited by the length of the smallest dimension of the data grid. For example, in a typical OpenAtom [128] simulation, each of the electronic state is represented using a grid that ranges from $100 \times 100 \times 100$ to $300 \times 300 \times 300$. As a result, 3D-FFTs of these grids and that of density-grid exhibit limited parallelism (100-300). This affects the scalability of OpenAtom significantly, especially on large machines with powerful compute nodes.

8.2.1 Overview

In order to eliminate the scaling bottleneck induced by parallel FFTs, we have developed a fully asynchronous Charm++ based FFT library, called Charm-FFT. This library allows users to create multiple instances of itself and perform concurrent FFTs with them. Each of the FFT calls runs in the background concurrently with other parts of the user code, and a callback is invoked when the FFT is complete. Currently, FFTW [68] is used to perform sequential line FFTs in Charm-FFT. The key features of this library are:

1. 2D-decomposition to enable more parallelism: at higher core counts, users can define fine-grained decomposition to increase the amount of available parallelism and better utilize the network bandwidth.
2. Cutoff-based reduced communication: provides an option for users to specify a cutoff, which can be used to prune parts of the grid. Such cutoffs are typically based on scientific property of the entities represented by the grid. For example, the density grid in OpenAtom has a *g-space* spherical cutoff [129]. Charm-FFT can improve performance by avoiding communication of data beyond the cutoff points and by performing fewer line FFTs.
3. User-defined mapping of library objects: the placement of objects that constitute the FFT instance can be defined by user based on the application's other concurrent communication.
4. Overlap with other computational work and other FFTs: given the callback-based interface and Charm++'s asynchrony, the FFTs are performed in the background while other application work is performed.



(a) 2D-decomposition: each D1 object performs line FFT on a few pencils (along Z dimension).

(b) Cutoff in Z dimension: D1 objects send only those parts of the grid to D2 objects that satisfy the linear cutoff constraint in Z dimension.

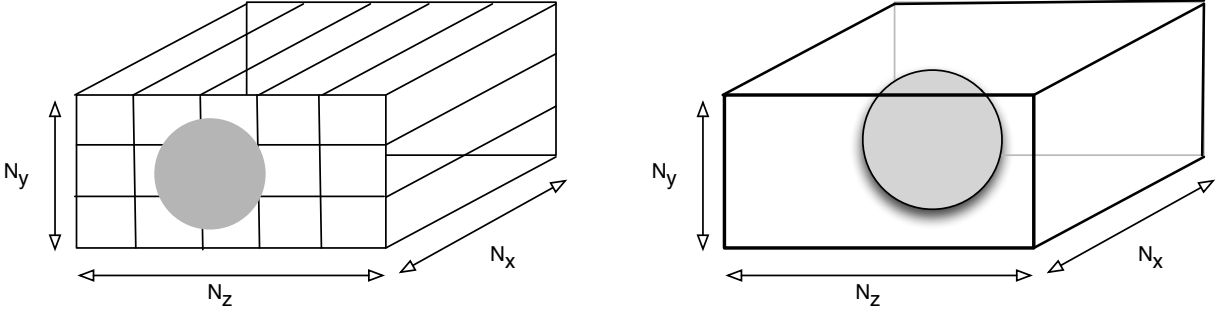
Figure 8.1: Phase 1 and 2 of Charm-FFT.

8.2.2 Implementation details

The creation of an instance of the FFT library is performed by calling *Charm_createFFT* from one of the processes. The user is required to specify the size of the FFT grid and the desired decomposition. Optionally, a cutoff and mapping of FFT objects can be specified. A callback can also be specified; this callback is invoked when the distributed creation of the FFT library is completed. Internally, the FFT library creates three types of Charm++ objects: D1, D2, and D3. Each of the D1, D2, and D3 objects owns a thin bar (a pencil) of the FFT-grid in one of the dimensions (say Z, Y, X). The decomposition of the FFT-grid among these objects is decided based on the user input and can be obtained by querying the library. Figure 8.1a shows an example of decomposing a grid among D1 objects. Since D1 objects own pencils along Z dimension, a balanced distribution is obtained by dividing the XY-plane using a 2D-decomposition.

Typically, D1 objects are associated with the grid in the time (real) space. The D3 objects own pencils along X dimensions, and are associated with the output grid in the wave space. In order to load balance the number of points in the sphere owned by each of the D3 objects, the X-pencils are assigned to D3 objects in a round-robin fashion. Unlike D1 objects which are part of a two-dimensional collection, D3 objects are a 1D collection. The D2 objects are not visible to the user as they are used for the intermediate Y-pencil FFTs and transpose. They are also a two-dimensional collection with decomposition in the XZ-plane. Due to the cutoff constraints, the typical length of Z-dimension in D2 objects is shorter than the Z-pencil length used by D1 objects.

Either when the FFT library is created, or when the callback is invoked after the set up is complete, the application obtains a unique identifier for the newly created library



(a) After D2 objects perform line FFTs along Y dimension, only a cylindrical grid is left within the cutoff.

(b) D3 objects perform the final FFTs along X-dimension, and provide the user with the sphere that satisfies the cutoff constraint.

Figure 8.2: Phase 3 and 4 of Charm-FFT.

instance. It can then make distributed calls to query the parts of FFT-grid owned by various processes. Before computing a FFT, the processes are required to assign the grid memory to D1 and D3. This can be done by calling the functions *Charm_setInputMemory* and *Charm_setOutputMemory* in a distributed manner. The plans required for performing line FFTs in FFTW can either be explicitly created before invoking the FFTs, or will be created by the library before performing the FFT.

After the set up is complete and the grid memory has been assigned, FFT computation can be started on a instance by calling *Charm_doForwardFFT* or *Charm_doBackwardFFT*. As suggested by their names, these calls are for performing forward FFTs (time to wave space, D1 to D3 objects) and backward FFTs (wave to time space, D3 to D1 objects), respectively. Both these calls are non-blocking and returns immediately without actually performing the FFT. The requests to perform FFTs are registered with the runtime system.

If *Charm_doForwardFFT* is invoked, lines FFTs are performed locally along Z dimension by D1 objects. Following these line FFTs, any data along Z axis that is beyond the cutoff is ignored, and *only the data within the cutoff is communicated* to D2 objects. In Figure 8.1b, the cutoff is represented by the grey region and is based only on the Z-coordinate/index of the grid points. At D2, line FFTs along Y dimension are performed. After these FFTs, the FFT-grid within the cutoff is further reduced to a cylinder of thin bars. This is because both Y and Z coordinates are now used to determine if a point falls in the cutoff range (Figure 8.2a).

The D2 objects communicate the data within the cylinder to D3 objects. The third and the final line FFT is performed by D3 objects along X dimension. Now, the grid points within the cutoff range are defined by a sphere as shown in the Figure 8.2b. At this point, the

#Objects	Decomposition	Time (ms)
100	10×10	80
300	300×1	76
300	75×4	69
300	20×15	45
400	20×20	35
900	30×30	24
1600	40×40	24
2500	50×50	22
3600	60×60	23

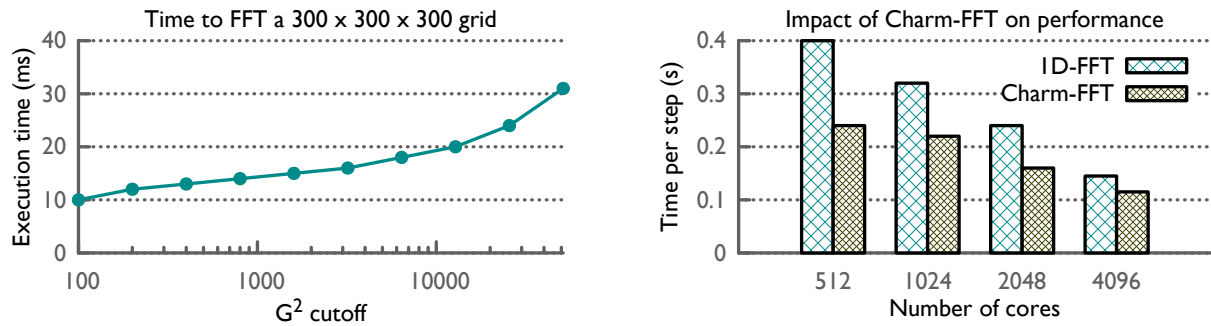
Table 8.6: Effect of decomposition on time to perform FFT on a $300 \times 300 \times 300$ grid. Representative best values are shown for each of the object counts.

application specified callback is invoked which informs the application of FFT’s completion. As mentioned earlier, the distribution of pencils, which contain the data in this sphere, to D3 objects is performed such that the total number of grid points that are in the sphere are load balanced across D3 objects. If *Charm_doBackwardFFT* is invoked, these steps are performed in reverse order. Note that if FFTs are started on multiple instances one after the other, all of them are performed concurrently.

8.2.3 Performance of Charm-FFT

Table 8.6 presents the impact of decomposition choice on the time to perform 3D-FFT on a $300 \times 300 \times 300$ grid executed on 512 nodes of Blue Gene/Q. The baseline performance is taken to be 76 ms which is obtained when 1D-decomposition of the grid is performed, i.e. the grid is divided among 300 objects.

In Table 8.6, it can be seen that as we perform finer decomposition of the grid along two dimensions, the time to FFT reduces significantly. The best performance is obtained when we divide the grid among 2,500 objects that are arranged as a 2D grid of size 50×50 . The time to perform FFT is reduced by 70% in comparison to the baseline FFT. Further decreasing the decomposition granularity leads to communication overheads which impact the performance negatively. Another important observation from the table is the performance difference between two cases in which the same object count is used, but with different layouts. A 20×15 decomposition improves the FFT performance by 40% over a 300×1 decomposition. This is because use of a 2D-decomposition results in small sized messages which can be communicated effectively.



(a) As the G^2 cutoff decreases, the time to FFT reduces due to the decrease in the communication performed during the transpose by Charm-FFT.

(b) OpenAtom performance for Water test system: by exploiting 2D-decomposition and cutoff-based reduced communication, Charm-FFT is able to improve the iteration time of OpenAtom by up to 40%.

Figure 8.3: Performance of Charm-FFT.

Next, we compare the effect of the choice of G^2 cutoff on the time taken to perform FFT. Figure 8.3a shows that the choice of cutoff can cause the time to FFT to vary from 10 ms to 30 ms for a $300 \times 300 \times 300$ grid executed on 512 nodes of Blue Gene/Q using the best 2D-decomposition. While a cutoff of 100 is unrealistic from a science perspective, G^2 values that eliminate as many as half the grid points are common. In Figure 8.3a, the data point at 6,400 cutoff is close to such a scenario. Hence, by the use of cutoff aware FFT, Charm-FFT can reduce the time to FFT by 41%, which can significantly improve performance of FFT-heavy applications such as OpenAtom.

Finally, in Figure 8.3b, we present the impact of using Charm-FFT in OpenAtom for the Water system scaled to core counts that are at its parallelization limits. For most of the core counts, use of Charm-FFT reduces the per step time of OpenAtom by 30%-40%. For core counts less than 400, we observe that the performance of the default version of OpenAtom matches closely with the performance of the version that deploys Charm-FFT. This is expected since for smaller core counts, the default 1D-decomposition is able to utilize most of the network bandwidth. At very large counts (3,584 in this case), the improvement due to the Charm-FFT is around 20%. We believe the reduction at very large core count is due to the increased overheads of sending and processing many small messages.

8.3 Summary

In this chapter, we have presented two diverse case studies in which use of communication-centric algorithmic design leads to significant improvement in communication performance. The first of these studies was on collectives in MPI, which are often the performance bottleneck in scaling applications. We presented a new set of algorithms, called two-tier algorithms, for performing collectives on the dragonfly networks to take advantage of the interconnect topology. A comparison, based on a cost model and network utilization, was done to show the superiority of these new algorithms in comparison to well know algorithms. In the second case study, we presented Charm-FFT, a fully asynchronous Charm++ based FFT library, which facilitates 2D-decomposition of data grid to increase the amount of available parallelism. Additionally, this library uses a science based cutoff to reduce the amount of communication and computation required to perform the FFT. We showed that use of Charm-FFT in OpenAtom improves its performance by up to 40%.

Conclusion

This thesis has explored the flow of communication on various HPC networks when applications with diverse patterns are executed on the systems. Detailed studies that focus on different components of the communication stack, such as the application's communication pattern, runtime's communication support, injection policy on the system, routing protocol, etc., have been presented. Impact of various environment parameters, e.g. the network topology and the job placement policy, on communication performance of different applications has also been analyzed and presented in this thesis. From the experience we gained in these case studies, we conclude that communication optimization should not be performed in isolation. While optimizing one aspect of communication can lead to significant improvements, an inclusive approach that spans different components and configurations, which affect communication, should be adopted to obtain the best possible communication performance.

Several contributions have been made in different chapters of this thesis, most of which have been discussed at the end of the chapters. Here, we summarize them briefly for completeness:

- A new functional model to predict steady state traffic distribution on dragonfly networks has been proposed and its scalable parallel implementation using MPI, Damsel, has been described.
- Using Damsel, we have shown that adaptive hybrid routing with randomized placement at the granularity of nodes and routers is the suggested choice to obtain best network utilization for parallel workloads executing on a dragonfly network.
- A first of its kind machine learning approach is presented to understand network congestion on supercomputer networks. Using this approach, three features are shown

to be highly correlated with the observed communication performance: maximum receive buffers on intermediate nodes, average network link load and maximum injection FIFOs length (arranged in decreasing order of importance). Using these features accuracy close to 1.0 is obtained for predicting performance of various communication kernels and production applications.

- A three-step guide to analyze and improve communication performance in production applications has been presented. As a demonstration of these guidelines, communication performance of two production applications, pF3D and MILC, has been improved by 79% and 21%, respectively.
- A new set of algorithms, called two-tier algorithms, are proposed for performing collectives on the dragonfly networks to take advantage of their topology. A cost model and network utilization based comparison with well know algorithms shows that the the two-tier algorithms significantly outperform most other algorithms.
- Charm-FFT, a new scalable FFT library which uses 2D-decomposition and minimizes communication by using cutoff based on scientific properties of the data grid, is presented and its benefits are shown.
- A highly scalable trace-driven packet-level simulator, TRACER, is presented to enable simulation of real HPC codes on very large networks. It is shown to outperform state-of-the-art simulators such as BigSim and SST in serial mode and significantly lower the simulation time on large core counts.
- TRACER is used to compare three common networks - torus, dragonfly, and fat-tree, using three metrics - performance, cost, and performance per dollar. It is shown that different networks provide the best performance depending on the communication pattern. In terms of cost, torus is shown to be most expensive by a significant margin, while dragonfly and fat-tree are found to be much cheaper. Finally, the fat-tree is shown to be the best network for the performance per dollar metric.
- TRACER is also used to conduct a novel parametric evaluation of what-if scenarios on the torus and dragonfly networks. It is found that dragonfly provides best performance when injection bandwidth is higher than link bandwidth, while torus saturates at lower injection bandwidth. Round-robin injection policy in combination with Adaptive routing is also shown to typically provide better performance than the FCFS policy.

REFERENCES

- [1] S. Kamil, L. Olikar, A. Pinar, and J. Shalf, “Communication requirements and interconnect optimization for high-end scientific applications,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 2, pp. 188–202, Feb. 2010.
- [2] B. Duzett and R. Buck, “An overview of the ncube 3 supercomputer,” in *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, Oct 1992, pp. 458–464.
- [3] C. Leiserson, “Fat-trees: Universal Networks for Hardware-Efficient Supercomputing,” *IEEE Transactions on Computers*, vol. 34, no. 10, October 1985.
- [4] *The Connection Machine CM-5 Technical Summary*, Thinking Machines Corporation, 245 First Street, Cambridge, MA 02154-1264, October 1991.
- [5] X. Yuan, “On nonblocking folded-clos networks in computer communication environments,” in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, May 2011, pp. 188–196.
- [6] “Lonestar supercomputer at TACC,” <https://www.tacc.utexas.edu/systems/lonestar>.
- [7] “Stampede supercomputer at TACC,” <https://www.tacc.utexas.edu/stampede/>.
- [8] “Wikipedia entry on torus,” <http://en.wikipedia.org/wiki/Torus>.
- [9] M.Blumrich, D.Chen, P.Coteus, A.Gara, M.Giampapa, P.Heidelberger, S.Singh, B.Steinmacher-Burow, T.Takken, and P.Vranas, “Design and Analysis of the Blue Gene/L Torus Interconnection Network,” *IBM Research Report*, December 2003.
- [10] Y. Ajima, S. Sumimoto, and T. Shimizu, “Tofu: A 6d mesh/torus interconnect for exascale computers,” *Computer*, vol. 42, pp. 36–40, 2009.
- [11] Cray Inc., “Cray XE6 Specifications,” <http://www.cray.com/Assets/PDF/products/xe/CrayXE6Brochure.pdf>, 2010.
- [12] S. Kumar, A. Mamidala, D. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burow, “PAMI: A parallel active message interface for the BlueGene/Q supercomputer,” in *Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Shanghai, China, May 2012.

- [13] “Wikipedia article on torus interconnect,” https://en.wikipedia.org/wiki/Torus_interconnect.
- [14] “Llnl page on bg/q,” <https://computing.llnl.gov/tutorials/bgq/>.
- [15] J. Kim, W. J. Dally, S. Scott, and D. Abts, “Technology-driven, highly-scalable dragonfly topology,” *SIGARCH Comput. Archit. News*, vol. 36, pp. 77–88, June 2008.
- [16] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, “Cray cascade: A scalable hpc system based on a dragonfly network,” in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, Nov 2012.
- [17] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony, “The PERCS High-Performance Interconnect,” in *2010 IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)*, August 2010, pp. 75–82.
- [18] Michalakes, J., J. Dudhia, D. Gill, T. Henderson, J. Klemp, W. Skamarock, and W. Wang, “The Weather Research and Forecast Model: Software Architecture and Performance,” in *Proceedings of the 11th ECMWF Workshop on the Use of High Performance Computing In Meteorology*, October 2004.
- [19] C. Bernard, T. Burch, T. A. DeGrand, C. DeTar, S. Gottlieb, U. M. Heller, J. E. Hetrick, K. Orginos, B. Sugar, and D. Toussaint, “Scaling tests of the improved Kogut-Susskind quark action,” *Physical Review D*, no. 61, 2000.
- [20] C. H. Still, R. L. Berger, A. B. Langdon, D. E. Hinkel, L. J. Suter, and E. A. Williams, “Filamentation and forward brillouin scatter of entire smoothed and aberrated laser beams,” *Physics of Plasmas*, vol. 7, no. 5, p. 2023, 2000.
- [21] X. Ni, L. V. Kale, and R. Tamstorf, “Scalable asynchronous contact mechanics using charm++,” in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (to appear)*, ser. IPDPS ’15. IEEE Computer Society, May 2015, ILNL-CONF-663041.
- [22] R. V. Vadali, Y. Shi, S. Kumar, L. V. Kale, M. E. Tuckerman, and G. J. Martyna, “Scalable fine-grained parallelization of plane-wave-based ab initio molecular dynamics for large supercomputers,” *Journal of Computational Chemistry*, vol. 25, no. 16, pp. 2006–2022, Oct. 2004.
- [23] J. Phillips, G. Zheng, and L. V. Kalé, “Namd: Biomolecular simulation on thousands of processors,” in *Workshop: Scaling to New Heights*, Pittsburgh, PA, May 2002.

- [24] F. Gygi, E. W. Draeger, M. Schulz, B. R. de Supinski, J. A. Gunnels, V. Austel, J. C. Sexton, F. Franchetti, S. Kral, C. W. Ueberhuber, and J. Lorenz, “Large-scale electronic structure calculations of high-z metals on the bluegene/l platform,” in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: <http://doi.acm.org/10.1145/1188455.1188502>
- [25] F. Ercal and J. Ramanujam and P. Sadayappan, “Task allocation onto a hypercube by recursive mincut bipartitioning,” in *Proceedings of the 3rd conference on Hypercube concurrent computers and applications*. ACM Press, 1988, pp. 210–221.
- [26] S. Wayne Bollinger and Scott F. Midkiff, “Processor and Link Assignment in Multi-computers Using Simulated Annealing,” in *ICPP (1)*, 1988, pp. 1–7.
- [27] Soo-Young Lee and J. K. Aggarwal, “A Mapping Strategy for Parallel Processing,” *IEEE Trans. Computers*, vol. 36, no. 4, pp. 433–442, 1987.
- [28] G. Bhanot, A. Gara, P. Heidelberger, E. Lawless, J. C. Sexton, and R. Walkup, “Optimizing task layout on the Blue Gene/L supercomputer,” *IBM Journal of Research and Development*, vol. 49, no. 2/3, pp. 489–500, 2005.
- [29] H. Yu, I.-H. Chung, and J. Moreira, “Topology mapping for Blue Gene/L supercomputer,” in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 116.
- [30] T. Agarwal, A. Sharma, and L. V. Kalé, “Topology-aware task mapping for reducing communication contention on large parallel machines,” in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2006*, April 2006.
- [31] A. Bhatele, E. Bohm, and L. V. Kale, “Optimizing communication for charm++ applications by reducing network contention,” *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 211–222, 2011.
- [32] B. G. Fitch, A. Rayshubskiy, M. Eleftheriou, T. J. C. Ward, M. Giampapa, and M. C. Pitman, “Blue Matter: Approaching the Limits of Concurrency for Classical Molecular Dynamics,” in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM Press, 2006.
- [33] F. Gygi, E. W. Draeger, M. Schulz, B. R. D. Supinski, J. A. Gunnels, V. Austel, J. C. Sexton, F. Franchetti, S. Kral, C. Ueberhuber, and J. Lorenz, “Large-Scale Electronic Structure Calculations of High-Z Metals on the Blue Gene/L Platform,” in *Proceedings of the International Conference in Supercomputing*. ACM Press, 2006.
- [34] A. Bhatele, “Automating Topology Aware Mapping for Supercomputers,” Ph.D. dissertation, Dept. of Computer Science, University of Illinois, August 2010, <http://hdl.handle.net/2142/16578>.

- [35] T. Hoefler and M. Snir, “Generic topology mapping strategies for large-scale parallel architectures,” in *Proceedings of the international conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 75–84.
- [36] R. A. Fiedler and S. Whalen, “Improving task placement for applications with 2d, 3d, and 4d virtual cartesian topologies on 3d torus networks with service nodes,” in *Cray User Group Conference*. The National Energy Research Scientific Computing Center, CA, 2013.
- [37] J. C. Phillips, Y. Sun, N. Jain, E. J. Bohm, and L. V. Kale, “Mapping to Irregular Torus Topologies and Other Techniques for Petascale Biomolecular Simulation,” in *Proceedings of ACM/IEEE SC 2014*, New Orleans, Louisiana, November 2014.
- [38] A. Bhatele, T. Gamblin, S. H. Langer, P.-T. Bremer, E. W. Draeger, B. Hamann, K. E. Isaacs, A. G. Landge, J. A. Levine, V. Pascucci, M. Schulz, and C. H. Still, “Mapping applications with collectives over sub-communicators on torus networks,” in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. IEEE Computer Society, Nov. 2012 (to appear), ILNL-CONF-556491.
- [39] Aleliunas, R. and Rosenberg, A. L., “On Embedding Rectangular Grids in Square Grids,” *IEEE Trans. Comput.*, vol. 31, no. 9, pp. 907–913, 1982.
- [40] S.-K. Lee and H.-A. Choi, “Embedding of complete binary trees into meshes with row-column routing,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, pp. 493–497, May 1996.
- [41] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauer, E. Santos, R. Subramonian, and T. von Eicken, “Logp: Towards a realistic model of parallel computation,” in *Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, San Diego, CA, May 1993.
- [42] A. Alexandrov, M. F. Ionescu, K. E. Schauer, and C. Scheiman, “Loggp: incorporating long messages into the logp model one step closer towards a realistic model for parallel computation,” in *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '95. New York, NY, USA: ACM, 1995. [Online]. Available: <http://doi.acm.org/10.1145/215399.215427> pp. 95–105.
- [43] M. I. Frank, A. Agarwal, and M. K. Vernon, “Lopc: modeling contention in parallel algorithms,” in *Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPOPP '97. New York, NY, USA: ACM, 1997. [Online]. Available: <http://doi.acm.org/10.1145/263764.263803> pp. 276–287.
- [44] C. A. Moritz and M. I. Frank, “Logpc: Modeling network contention in message-passing programs,” *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 1, pp. 254–263, June 1998.

- [45] C. Moritz and M. Frank, “Logpg: Modeling network contention in message-passing programs,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 12, no. 4, pp. 404–415, apr 2001.
- [46] W. Chen, J. Zhai, J. Zhang, and W. Zheng, “Loggpo: An accurate communication model for performance prediction of mpi programs,” *Science in China Series F: Information Sciences*, vol. 52, no. 10, pp. 1785–1791, 2009.
- [47] D. Martinez, J. Cabaleiro, T. Pena, F. Rivera, and V. Blanco, “Accurate analytical performance model of communications in mpi applications,” in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009, pp. 1–8.
- [48] T. Hoefer, T. Schneider, and A. Lumsdaine, “LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model,” in *Proceedings of the 19th ACM International Symposium on HPDC*. ACM, Jun. 2010, pp. 597–604.
- [49] G. Zheng, G. Kakulapati, and L. V. Kalé, “BigSim: A parallel simulator for performance prediction of extremely large parallel machines,” in *18th International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, New Mexico, April 2004, p. 78.
- [50] K. Underwood, M. Levenhagen, and A. Rodrigues, “Simulating red storm: Challenges and successes in building a system simulation,” in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, 2007*, pp. 1–10.
- [51] “DUMPI: The mpi trace library,” http://sst.sandia.gov/about_dumpi.html.
- [52] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, J. Kim, and W. J. Dally, “A detailed and flexible cycle-accurate network-on-chip simulator,” in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2013.
- [53] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. V. Hensbergen, and L. Zhang, “Mambo: a full system simulator for the PowerPC architecture,” *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 4, pp. 8–12, 2004.
- [54] S. Bohm and C. Engelmann, “xSim: The extreme-scale simulator,” *HPCS*, 2011.
- [55] S. Girona and J. Labarta, “Sensitivity of performance prediction of message passing programs,” *The Journal of Supercomputing*, 2000.
- [56] M. M. Tikir, M. A. Laurenzano, L. Carrington, and A. Snaveley, “Psins: An open source event tracer and execution simulator,” *HPCMP Users Group Conference*, vol. 0, pp. 444–449, 2009.
- [57] B. Penoff, A. Wagner, M. Tuxen, and I. Rungeler, “Mpi-netsim: A network simulation module for mpi,” in *Parallel and Distributed Systems (ICPADS)*. IEEE, 2009.

- [58] C. Minkenbergh and G. Rodriguez, “Trace-driven co-simulation of high-performance computing systems using OMNeT++,” in *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, 2009, p. 65.
- [59] H. Casanova et al., “Versatile, scalable, and accurate simulation of distributed applications and platforms,” *Journal of Parallel and Distributed Computing*, June 2014.
- [60] R. Rabenseifner, “Automatic Profiling of MPI Applications with Hardware Performance Counters,” in *6th European PVM/MPI Users’ Group Meeting on Recent Advances in PVM and MPI*, 1999, pp. 35–42.
- [61] R. Thakur, R. Rabenseifner, and W. Gropp, “Optimization of Collective Communication Operations in MPICH,” *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.
- [62] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, “MPICH: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard,” *Parallel Computing*, vol. 22, no. 6, pp. 789–828, September 1996.
- [63] M. Shroff and V. D. Geijn, “Collmark: Mpi collective communication benchmark,” Tech. Rep., 2000.
- [64] N. Jain and Y. Sabharwal, “Optimal bucket algorithms for large mpi collectives on torus interconnects,” in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS ’10. New York, NY, USA: ACM, 2010. [Online]. Available: <http://doi.acm.org/10.1145/1810085.1810093> pp. 27–36.
- [65] E. Chan, M. Heimlich, A. Purkayastha, and R. Geijn, “Collective Communication: Theory, Practice, and Experience FLAME Working Note #22,” 2006.
- [66] A. Faraj, S. Kumar, B. Smith, A. Mamidala, J. Gunnels, and P. Heidelberger, “Mpi collective communications on the blue gene/p supercomputer: algorithms and optimizations,” in *Proceedings of the 23rd international conference on Supercomputing*, ser. ICS ’09, 2009, pp. 489–490.
- [67] S. Kumar, Y. Sabharwal, R. Garg, and P. Heidelberger, “Optimization of all-to-all communication on the blue gene/l supercomputer,” in *Proceedings of the 2008 37th International Conference on Parallel Processing*, ser. ICPP ’08. Washington, DC, USA: IEEE Computer Society, 2008. [Online]. Available: <http://dx.doi.org/10.1109/ICPP.2008.83> pp. 320–329.
- [68] M. Frigo and S. Johnson, “FFTW: an adaptive software architecture for the FFT,” *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3, pp. 1381–1384 vol.3, May 1998.
- [69] M. Eleftheriou, B. G. Fitch, A. Rayshubskiy, T. J. C. Ward, and R. S. Germain, “Scalable framework for 3D FFTs on the Blue Gene/L supercomputer: Implementation and early performance measurements,” *IBM Journal of Research and Development*, vol. 49, no. 2/3, 2005.

- [70] A. Chan, P. Balaji, W. Gropp, and R. Thakur, “Communication analysis of parallel 3d fft for flat cartesian meshes on large blue gene systems,” in *High Performance Computing - HiPC 2008*, ser. Lecture Notes in Computer Science, P. Sadayappan, M. Parashar, R. Badrinath, and V. Prasanna, Eds. Springer Berlin Heidelberg, 2008, vol. 5374, pp. 350–364. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89894-8_32
- [71] C. Young, J. A. Bank, R. O. Dror, J. P. Grossman, J. K. Salmon, and D. E. Shaw, “A 32x32x32, spatially distributed 3D FFT in four microseconds on Anton,” in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–11.
- [72] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, “There goes the neighborhood: performance degradation due to nearby jobs,” in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. IEEE Computer Society, Nov. 2013, LLNL-CONF-635776.
- [73] A. Bhatele, N. Jain, W. D. Gropp, and L. V. Kale, “Avoiding hot-spots on two-level direct networks,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 76:1–76:11.
- [74] A. Bhatele, N. Jain, K. E. Isaacs, R. Buch, T. Gamblin, S. H. Langer, and L. V. Kale, “Optimizing the performance of parallel applications on a 5D torus via task mapping,” in *Proceedings of IEEE International Conference on High Performance Computing (to appear)*, ser. HiPC '14. IEEE Computer Society, Dec. 2014, LLNL-CONF-655465.
- [75] J. Vetter and C. Chambreau, “mpip: Lightweight, Scalable MPI Profiling,” <http://mpip.sourceforge.net>.
- [76] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, “Hpctoolkit: Tools for performance analysis of optimized parallel programs,” *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.
- [77] I.-H. Chung, R. E. Walkup, H.-F. Wen, and H. Yu, “Mpi performance analysis tools on blue gene/l,” in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006.
- [78] A. Bhatele, G. Gupta, L. V. Kale, and I.-H. Chung, “Automated Mapping of Regular Communication Graphs on Mesh Interconnects,” in *Proceedings of International Conference on High Performance Computing (HiPC)*, 2010.
- [79] A. Bhatele and L. V. Kale, “Heuristic-based techniques for mapping irregular communication graphs to mesh topologies,” in *Proceedings of Workshop on Extreme Scale Computing Application Enablement - Modeling and Tools*, September 2011.

- [80] M. Deveci, S. Rajamanickam, V. J. Leung, K. Pedretti, S. L. Olivier, D. P. Bunde, U. V. Çatalyürek, and K. Devine, “Exploiting geometric partitioning in task mapping for parallel computers,” in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '14. IEEE Computer Society, May 2014.
- [81] M. Collaboration, “MIMD Lattice Computation (MILC) Collaboration Home Page,” <http://www.physics.indiana.edu/~sg/milc.html>.
- [82] N. Jain, A. Bhatele, M. P. Robson, T. Gamblin, and L. V. Kale, “Predicting application performance using supervised learning on communication features,” in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. IEEE Computer Society, Nov. 2013, ILNL-CONF-635857.
- [83] “NERSC-8: Trinity Benchmarks.” [Online]. Available: <https://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement>
- [84] A. G. Landge, J. A. Levine, K. E. Isaacs, A. Bhatele, T. Gamblin, M. Schulz, S. H. Langer, P.-T. Bremer, and V. Pascucci, “Visualizing network traffic to understand the performance of massively parallel simulations,” in *IEEE Symposium on Information Visualization (INFOVIS'12)*, Seattle, WA, October 14-19 2012, LLNL-CONF-543359.
- [85] N. Jain, A. Bhatele, X. Ni, N. J. Wright, and L. V. Kale, “Maximizing throughput on a dragonfly network,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014. [Online]. Available: <http://dx.doi.org/10.1109/SC.2014.33> pp. 336–347.
- [86] K. Antypas, J. Shalf, and H. Wasserman, “NERSC6 Workload Analysis and Benchmark Selection Process,” Lawrence Berkeley National Lab, Tech. Rep. LBNL-1014E, 2008.
- [87] B. Austin, M. Cordery, H. Wasserman, and N. Wright, “Performance measurements of the nersc cray cascade system.” Cray, Inc., May 2013.
- [88] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, “Exascale computing study: Technology challenges in achieving exascale systems,” 2008.
- [89] B. Prisacari, G. Rodriguez, P. Heidelberger, D. Chen, C. Minkenbergh, and T. Hoefler, “Efficient task placement and routing of nearest neighbor exchanges in dragonfly networks,” in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14. ACM, 2014. [Online]. Available: <http://doi.acm.org/10.1145/2600212.2600225> pp. 129–140.
- [90] C. Huang, O. Lawlor, and L. V. Kalé, “Adaptive MPI,” in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003)*, LNCS 2958, College Station, Texas, October 2003, pp. 306–322.

- [91] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé, “NAMD: Biomolecular simulation on thousands of processors,” in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Baltimore, MD, September 2002, pp. 1–18.
- [92] G. Kresse and J. Hafner, “Ab initio molecular dynamics for liquid metals,” *Phys. Rev. B*, vol. 47, p. 558, 1993.
- [93] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, W. Wang, and J. G. Powers, “A description of the advanced research wrf version 2,” NCAR, Tech. Rep. Technical Note NCAR/TN-468+STR, June 2005.
- [94] A. Bhatele, A. R. Titus, J. J. Thiagarajan, N. Jain, T. Gamblin, P.-T. Bremer, M. Schulz, and L. V. Kale, “Identifying the culprits behind network congestion,” in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (to appear)*, ser. IPDPS ’15. IEEE Computer Society, May 2015, ILNL-CONF-663150.
- [95] C. Hyatt and D. P. Agrawal, “Congestion control in the wormhole-routed torus with clustering and delayed deflection,” in *Parallel Computer Routing and Communication*, ser. Lecture Notes in Computer Science, S. Yalamanchili and J. Duato, Eds. Springer Berlin Heidelberg, 1998, vol. 1417, pp. 33–38.
- [96] A. Bhatelé and L. V. Kalé, “Quantifying Network Contention on Large Parallel Machines,” *Parallel Processing Letters (Special Issue on Large-Scale Parallel Processing)*, vol. 19, no. 4, pp. 553–572, 2009.
- [97] J. Escudero-Sahuquillo, E. Gran, P. Garcia, J. Flich, T. Skeie, O. Lysne, F. Quiles, and J. Duato, “Combining congested-flow isolation and injection throttling in hpc interconnection networks,” in *2011 International Conference on Parallel Processing (ICPP)*, Sept 2011, pp. 662–672.
- [98] A. Bhatelé, L. V. Kalé, and S. Kumar, “Dynamic topology aware load balancing algorithms for molecular dynamics applications,” in *23rd ACM International Conference on Supercomputing*, 2009.
- [99] Shahid H. Bokhari, “On the Mapping Problem,” *IEEE Trans. Computers*, vol. 30, no. 3, pp. 207–214, 1981.
- [100] S. Langer, A. Bhatele, and C. H. Still, “pF3D simulations of laser-plasma interactions in National Ignition Facility experiments,” *Computing in Science and Engineering*, vol. 99, Aug. 2014, ILNL-JRNL-648736. [Online]. Available: <http://doi.ieeecomputersociety.org/10.1109/MCSE.2014.79>
- [101] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [102] L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

- [103] J. H. Friedman, “Stochastic gradient boosting,” *Computational Statistics & Data Analysis*, vol. 38, no. 4, pp. 367–378, 2002.
- [104] A. Natekin and A. Knoll, “Gradient boosting machines, a tutorial,” *Frontiers in neurobotics*, vol. 7, 2013.
- [105] “Kendall tau rank correlation coefficient,” http://en.wikipedia.org/wiki/Kendall_tau_rank_correlation_coefficient.
- [106] J. Cope, N. Liu, S. Lang, P. Carns, C. Carothers, and R. Ross, “Codes: Enabling co-design of multilayer exascale storage architectures,” in *Proceedings of the Workshop on Emerging Supercomputing Technologies*, 2011.
- [107] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. V. Kalé, “Simulation-based performance prediction for large parallel machines,” in *International Journal of Parallel Programming*, vol. 33, no. 2-3, 2005, pp. 183–207.
- [108] C. D. Carothers, D. Bauer, and S. Pearce, “ROSS: A high-performance, low-memory, modular Time Warp system,” *Journal of Parallel and Distributed Computing*, vol. 62, no. 11, pp. 1648–1669, 2002.
- [109] B. Acun, N. Jain, A. Bhatele, M. Mubarak, C. D. Carothers, and L. V. Kale, “Preliminary evaluation of a parallel trace replay tool for hpc network simulations,” in *Workshop on Parallel and Distributed Agent-Based Simulations*, ser. PADABS, EURO-PAR, Aug. 2015.
- [110] L. V. Kale and A. Bhatele, Eds., *Parallel Science and Engineering Applications: The Charm++ Approach*. Taylor & Francis Group, CRC Press, Nov. 2013.
- [111] R. B. R. Misbah Mubarak, Christopher D. Carothers and P. Carns, “A case study in using massively parallel simulation for extreme-scale torus network codesign,” in *Proceedings of the 2nd ACM SIGSIM PADS*. ACM, 2014, pp. 27–38.
- [112] P. D. Barnes, Jr., C. D. Carothers, and D. R. e. a. Jefferson, “Warp speed: Executing time warp on 1,966,080 cores,” in *Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM-PADS, New York, NY, USA, 2013. [Online]. Available: <http://doi.acm.org/10.1145/2486092.2486134> pp. 327–336.
- [113] L. Schwiebert and D. N. Jayasimha, “On measuring the performance of adaptive worm-hole routing,” *hipc*, vol. 00, p. 336, 1997.
- [114] D. Chen, N. Eisley, P. Heidelberger, S. Kumar, A. Mamidala, F. Petrini, R. Senger, Y. Sugawara, R. Walkup, B. Steinmacher-Burow, A. Choudhury, Y. Sabharwal, S. Singhal, and J. J. Parker, “Looking under the hood of the ibm blue gene/q network,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2388996.2389090> pp. 69:1–69:12.

- [115] E. Zahavi, G. Johnson, D. J. Kerbyson, and M. Lang, “Optimized infinibandtm fat-tree routing for shift all-to-all communication patterns,” *Concurrency and Computation: Practice and Experience*, vol. 22, no. 2, pp. 217–231, 2010. [Online]. Available: <http://dx.doi.org/10.1002/cpe.1527>
- [116] B. Prisacari, G. Rodriguez, C. Minkenberg, and T. Hoefer, “Fast pattern-specific routing for fat tree networks,” *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 36:1–36:25, Dec. 2013. [Online]. Available: <http://doi.acm.org/10.1145/2555289.2555293>
- [117] S. Brookes and A. Roscoe, “Deadlock analysis in networks of communicating processes,” *Distributed Computing*, vol. 4, no. 4, pp. 209–230, 1991. [Online]. Available: <http://dx.doi.org/10.1007/BF01784721>
- [118] R. Cypher and L. Gravano, “Storage-efficient, deadlock-free packet routing algorithms for torus networks,” *Computers, IEEE Transactions on*, vol. 43, no. 12, pp. 1376–1385, Dec 1994.
- [119] C. Carrion, R. Beivide, J. Gregorio, and F. Vallejo, “A flow control mechanism to avoid message deadlock in k-ary n-cube networks,” in *High-Performance Computing, 1997. Proceedings. Fourth International Conference on*, Dec 1997, pp. 322–329.
- [120] V. Puente, R. Beivide, J. Gregorio, J. Prellezo, J. Duato, and C. Izu, “Adaptive bubble router: a design to improve performance in torus networks,” in *Parallel Processing, 1999. Proceedings. 1999 International Conference on*, 1999, pp. 58–67.
- [121] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale, “Overcoming scaling challenges in biomolecular simulations across multiple platforms,” in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.
- [122] R. B. R. Misbah Mubarak, Christopher D. Carothers and P. Carns, “Modeling a million-node dragonfly network using massively parallel discrete-event simulation,” *SCC, SC Companion*, 2012.
- [123] *Cray T3D System Architecture Overview*, Cray Research, Inc., March 1993.
- [124] “MPI: A Message Passing Interface Standard,” in *MPI Forum*, <http://www.mpi-forum.org/>.
- [125] R. Rabenseifner, “A new optimized MPI reduce algorithm,” 1997.
- [126] R. Thakur and W. D. Gropp, “Improving the Performance of Collective Operations in MPICH,” *Lecture Notes in Computer Science*, vol. 2840, pp. 257–267, October 2003.
- [127] M. Barnett, S. Gupta, D. G. Payne, L. Shuler, R. Geijn, and J. Watts, “Interprocessor Collective Communication Library (InterCom),” in *In Proceedings of the Scalable High Performance Computing Conference*, 1994, pp. 357–364.

- [128] S. Kumar, Y. Shi, E. Bohm, and L. V. Kale, “Scalable, fine grain, parallelization of the car-parrinello ab initio molecular dynamics method,” UIUC, Dept. of Computer Science, Tech. Rep., 2005.
- [129] B. Fang, G. Martyna, and Y. Deng, “A fine grained parallel smooth particle mesh ewald algorithm for biophysical simulation studies: Application to the 6-d torus QCDOC supercomputer,” *Computer Physics Communications*, vol. 177, no. 4, pp. 362 – 377, 2007. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0010465507002445>