OPTIQ: A Data Movement Optimization Framework for Data-centric Applications on Supercomputers

by

ANH HUY BUI
B.A. (Hanoi University of Science and Technology) 2006
M.S. (Politecnico di Milano) 2009

Thesis 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 Chicago, 2015

Chicago, Illinois

Defense Committee:

Andrew Johnson, Chair and Advisor
Ugo Buy
Luc Renambot
Jason Leigh, University of Hawaii at Manoa
Venkatram Vishwanath, Argonne National Laboratory

To my Mom and Dad for their endless support!

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisors: Andrew Johnson, Jason Leigh and Venkatram Viswanath for their guidance, motivation, encouragement, patient and support. Their roles have been inevitable to my success in the PhD program.

Also, I express my profound gratitude to my committee members: Andrew Johnson, Ugo Buy, Jason Leigh, Luc Renambot and Venkatram Vishwanath. I highly appreciate the value of their time and their feedback on my work.

I am really grateful to scientists, postdocs, staffs who are/had been at Argonne National Laboratory for their financial and technical support, helps and discussions. My special thank to Venkatram Viswanath, Michael E. Papka, Preeti Malakar, Eun-Sung Jung, Todd S. Munson, Sven Leyffer, Robert Jacob, Sebastien Boisvert and Jeff Hammond.

I sincerely thank staffs, colleages and students at Electronic Visualization Laboratory for their support, especially to Maxine Brown, Dana M. Plepys and Lance Long.

Last but not least, I would like to thank my family for being supportive of whatever I do.

AHB

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ALCF	Argonne Leadership Computing Facility
AMPL	A Modeling Language for Mathematical Programming
BG/Q	Blue Gene/Q
CESM	Community Earth System Model
CPLEX	Simplex Method Implemented in the C Programming Language
HACC	Hardware/Hybrid Accelerated Cosmology Code
IB	InfiniBand
IBM	International Business Machines
ILP	Integer Linear Programming
LP	Linear Programming
MPI	Message Passing Interface
OPTIQ	A Data Movement Optimization Framework for Data-centric Applications on Supercomputers
PAMI	Parallel Active Message Interface
QDR	Quad Data Rate (QDR) 10 Gb/s

LIST OF ABBREVIATIONS (Continued)

SNOPT	Sparse Nonlinear Optimization
XGFT	eXtended Generalized Fat Tree

SUMMARY

A holistic approach to improve and optimize throughput of data flows in data-centric applications on supercomputers is the research problem under consideration in this dissertation. The research presented in this dissertation comes from an observation that on supercomputing systems, data-centric applications need to reliably and rapidly compute and move large amounts of data through interconnect networks. This same trend is also observed in commodity clusters. Thus, improving and optimizing data movement performance is essential at extreme scales in order to effectively utilize these systems.

In many current supercomputing systems, data movement optimization has been done separately at different layers of the systems. Separate optimization is rational due to the complexity, diversity and sometimes conflicting nature of application communication patterns. However, it leads to suboptimal solutions. Most of the optimizations are carried out at the system routing level, favoring certain communication patterns. Communication patterns of applications, however, have a wide variation. They vary from application to application and even from phase to phase in a given application. This results in high networking performance for some communication patterns but lower performance for several other communication patterns.

In order to achieve close to global optimization, optimization of data movement needs to be carried out in a holistic approach. This dissertation presents approaches that take system network routing, interconnection network topology and application communication patterns into the optimization yielding better performance over current data movement mechanisms.

The approaches are realized in a Data Movement Optimization Framework (OPTIQ) that provides an application programming interface (API) requiring minimal changes in applications for integration. The OPTIQ framework is also extensible, allowing further development and expansion including algorithms for recommending multiple paths for data movement, different ways to schedule data transfer, and various mechanisms to transfer the data. It can also be extended to other systems.

This dissertation demonstrates the OPTIQ framework via a use case of improving and optimizing data movement by leveraging multi-path data movement to balance the load on physical network links on supercomputers. The dissertation proposes two greedy heuristic algorithms and a model-based optimization approach. The greedy heuristic approaches improve data movement with locally optimal solutions while the model-based approach yields globally optimal solutions requiring longer search time. Depending on the complexity of the systems and application communication patterns, the framework supports options to search for solutions offline, on-the-fly, or a hybrid combination of the two. The dissertation demonstrates the efficacy of the proposed approaches via a set of synthetic benchmarks and application benchmarks. The benchmarks show that over 91 experiments the presented approaches improved performance by 43% – 67% on average on up to 4096 nodes on a Blue Gene/Q supercomputer. The presented approaches improved data movement throughput for several important application communication patterns on two applications from 30% – 50%.

The presented data movement optimization framework proposes several solutions for the problem of optimization at large scale by using combined techniques, and offers different options

for time-to-solution based on the complexity of the problem. It also increases our understanding of the relationships between application communication patterns and system topologies and routing algorithms, giving deeper insight into supercomputing systems for the design of exascale systems.

CHAPTER 1

INTRODUCTION

The relentless march of Moore's law, combined with the growing computational demands of contemporary scientific applications, are resulting in ever-larger and more powerful supercomputers. Extracting the maximum performance from these leadership-class systems requires tuning not just microprocessor performance, but also achieving optimal networking performance in the supercomputer interconnect. This in turn requires optimizing the communication patterns of the applications and the routing algorithms used by the supercomputers. The challenge here is that application communication patterns are time-varying and have widely varying communication characteristics. Furthermore, supercomputer routing algorithms are optimized for typical communication patterns, and this can result in lower throughput for several other communication patterns, resulting in suboptimal performance for certain critical scientific applications. Thus, improving communication performance for generic communication patterns on a leadership-scale system is important for improving performance of data-centric applications which are run on such large-scale systems.

1.1 Motivation

A holistic approach to improve and optimize throughput of data flows in data-centric applications on supercomputers is the research problem under consideration in this dissertation. On supercomputers, data-centric applications need to reliably and rapidly compute and move large

amounts of data through interconnect networks. The same trend is also observed in commodity clusters. Thus, improving and optimizing data movement throughput is essential at extreme scales in order to effectively utilize supercomputers.

In many current supercomputing systems, data movement optimization has been done separately at different layers of the systems. Separate optimization is rational due to the complexity, diversity, and sometimes conflicts in application communication patterns. However, it leads to suboptimal solutions. At the system level, data movement optimization has been thoroughly investigated on routing algorithms. However it favors certain communication patterns over others leading to high performance for some communication patterns and low performance for other communication patterns. Optimizing data movement at the middleware level is optional and system-specific. At the application layer, we have seen load-balancing work done, but many data movement patterns are not optimized.

As separate and independent optimizations do not bring a close-to-global optimal solution, it is necessary to consider a holistic approach that brings together important factors in data movement from the system level to the application level. This dissertation proposes a holistic approach that takes system network routing, interconnect network topology, and application communication patterns into account in the optimization, promising to yield better performance over current data movement mechanisms.

The approach is realized in a Data Movement Optimization Framework (OPTIQ) that provides an application programming interface (API) requiring minimal changes when integrating it into the applications. The framework is designed to be extensible for further extensions of its

functionality and applying it on different systems. The Table I lists the inputs for optimizing data movement at different layers of supercomputers used by OPTIQ.

Layer	Inputs for optimization			
	Interconnect topology	System routing	Comm. routines	Comm. patterns
System	Yes		Most used	No
Middleware libraries	Limited, system specific	Yes (wrappers)	No	No
Applications	Limited	No	No	Limited
OPTIQ	Yes	Yes	No	Yes

TABLE I: Inputs for different layers on optimizing data movement on supercomputers.

Recently built supercomputers vary in term of their interconnection networks, routing algorithms, and compute node allocation policies. Their interconnection networks are becoming more complicated, and provide more links with higher throughput and lower latency. In current supercomputers we have various types of interconnection networks such as 3D, 5D torus, Fat Tree, and Dragonfly with various routing algorithms belonging to both oblivious and adaptive types of routing. However, improving data movement on different systems share certain features. Building a generic framework that can be further developed and customized and expanded upon for use on different systems may help to reduce development time and thus speed up scientific discoveries.

Improvement in data movement performance for the data-centric applications on supercomputers can reduce simulation time, leading to faster time-to-solution. This also can help to reduce energy consumption by applications on supercomputers.

1.2 Research Goals

Improving data movement performance for data-centric applications on supercomputers is the ultimate goal of this research. In order to achieve this goal, the dissertation investigates a holistic approach that takes important data movement factors, ranging from the system layer to the application layer, into consideration. These factors include the interconnection network topology, default routing policies, location of the running tasks of the applications, and the applications communication patterns.

The research aims to build a framework comprised of the necessary components for data movement improvement. These components need to cover reading the system configuration, receiving inputs from the applications, searching for paths to move data, and scheduling and moving data for the applications. The components expose an application programming interface(API) making it easy to implement, extend and expand to different supercomputers.

Path searching plays an important role in the framework as the data movement paths can significantly affect data movement performance. One of the goals of this research is to develop approaches to search for paths for data movement. The framework aims to provide several approaches to search for paths for data movement such as heuristics, linear programming, or hybrid combinations. Heuristic approaches can return paths quickly but they are usually system-specific, communication pattern specific, and do not guarantee a close-to-global opti-

mization. Optimization-based approaches can return globally optimal solutions but usually require longer time. Investigating the trade-off of different approaches in terms of calculating time and solution quality is important in this research.

The framework needs to provide different path searching modes such as offline, on-the-fly, or a combination of the two. This is because not all inputs for path searching may be available at one time. The framework takes several inputs to search for paths including coordinates of compute nodes, interconnection network topology, network bandwidth, routing algorithms, and data movement patterns. The inputs can be available to the framework at certain times depending on the supercomputers and the applications. In some supercomputers, the coordinates of nodes can be known before application execution. In others it can be known at the running time. The same is true with network link bandwidth, application communication patterns as well as other inputs. As the inputs can be available at different times, the framework needs to provide different path searching modes such as offline, on-the-fly, or a combination of the two. For example, if the application patterns are known in advance and compute nodes allocation is static, the framework supports calculating routing paths offline. Otherwise, the path searching might be done on-the-fly. The hybrid approach combines offline and on-the-fly approaches. It is used when partial information on data movement can be computed prior to runtime and the remaining data movement information is only available for collecting and calculating at runtime. If the framework calculates the data movement at runtime, due to additional time caused by collecting inputs and calculating data movement paths, the applications benefit from

the framework if the additional time for collecting and calculating is amortized from transfer time.

Evaluating the framework and proposed approaches is also important in this research. The efficacy of the framework and approaches need to be demonstrated through a set of experiments and synthetic benchmarks as well application benchmarks.

1.3 Thesis Organization

The dissertations is organized as follows: Chapter 2 briefly introduces data movement in data-centric applications. In Chapter 3, the dissertation gives an survey of optimizing and improving data movement in supercomputers and closely related problems and solutions. A brief comparison of the thesis contributions to previous work is also included. Chapter 4 presents the OPTIQ framework and its components, design, and functionalities in detail. The OPTIQ framework is extensible and includes algorithms for recommending multiple paths for data movement, different ways to schedule data transfer, and various mechanisms to transfer the data. It can also be extended to other systems. Chapter 5 presents two heuristic approaches and a model-based approach to leverage multipaths to improve and optimize data movement in supercomputers. Chapter 6 demonstrates the efficacy of OPTIQ via set of synthetic benchmarks and application benchmarks. Overall, our approaches improved performance by 43% – 67% on average for three different communication patterns in 91 experiments on up to 4096 nodes on a Blue Gene/Q supercomputer. The benchmarks show that OPTIQ can improve performance by 30% – 40% for the two applications that were experimented on. Chapter 7 gives concluding remarks and future work.

CHAPTER 2

DATA MOVEMENT IN DATA-CENTRIC APPLICATIONS

This chapter provides a preliminary understanding of data-centric applications on supercomputers and how supercomputers transfer data throughout their interconnection networks.

2.1 Data-centric Applications on Supercomputers

Supercomputers are powerful tools for scientific discovery. They provide an immense amount of mathematical calculation that is required for certain applications such as weather forecasting, fluid dynamic calculations, nuclear energy research, petroleum exploration, universe simulation, and human brain simulation, just to name a few. Supercomputers are not only capable of processing a large amount of data but also producing and exchanging vast amounts of data in a short time. Supporting communication in supercomputers are their low latency, high throughput interconnection networks. Applications running on the compute nodes of supercomputers exchange data through these interconnection networks. The data exchanged can be for computing or can be for I/O purposes. The applications can be either compute-centric, where most of the simulation time is for computing and the amount of data is relatively small, or data-centric where applications produce and exchange large amounts of data over the simulation time. This dissertation focuses on data-centric applications on supercomputers. In this section, the dissertation presents two, among many, scientific applications being run on supercomputers to demonstrate variations in the communication patterns and data sizes and exchange frequencies.

Hardware/Hybrid Accelerated Cosmology Code (HACC) (1) is a large-scale cosmology code suite that simulates the evolution of the universe through the first 13 billion years after the Big Bang. For I/O purpose, it has two different types of datasets: an analysis dataset and a checkpoint/restart dataset. The checkpoint dataset is typically 10 to 40 times larger than the analysis dataset, but also is written at lower frequency. Both of the datasets are transferred from a set of compute nodes to a set of I/O nodes through a high-speed interconnection network.

Community Earth System Model (CESM) (2) is a coupled global climate model simulating the earth's climate system. It consists of several main models: Atmosphere, Ocean, Land and Ice. Different models use different computational methods. During simulation the models communicate with each other through an intermediate component called the Coupler. This allows different models to be developed and modified independently. Modification of a model, if needed, is done in the model and the Coupler without affecting other models. When simulating on supercomputers, different models reside at different physical locations in a given partition of the compute nodes. Different models also have different communication patterns, with different data sizes and different communication frequencies. One example of the models' physical locations could be as follows: The Atmosphere model and the Coupler are on the same nodes. Both Land and Ice models reside within the Coupler. The Ocean model is disjoint from the Coupler. Different models also have different communication patterns, with different data sizes and different communication frequencies. Message sizes in the application can be as small as a few kilobytes per exchange, but can be adjusted to have larger message sizes.

The two applications presented here briefly introduced the diversity of communication patterns, and frequencies and message sizes of applications on supercomputers. In the next section, the dissertation gives an overview of how data is transferred on supercomputers.

2.2 Data Movement in Data-centric Applications on Supercomputers

Data movement in the data-centric applications on supercomputers considered in this dissertation can be either between compute nodes or between compute nodes and I/O nodes. In this section the dissertation presents the layers in a supercomputer that data travels through to reach to its destination from its source.

On supercomputers, optimization of data movement is carried out separately at different layers as depicted in Figure 1.

At the lowest level System Routing, data is divided into packets, which are routed along paths selected by the system. The system routes a packet based on its routing strategies. Two common routing strategies are oblivious and adaptive. In oblivious routing, routing paths are computed in advance without taking into account the state of the current traffic. Several optimal paths can be computed in advance and optimized for certain communication patterns (usually the most used patterns) but not all patterns. After the paths have been computed, the system uses them at runtime regardless of the current traffic situation. This is good for the patterns that were optimized, but not for other patterns, or for situations when traffic varies. Adaptive routing on the other hand, takes the state of the current traffic into account when calculating the routing paths. It is usually not optimized for any communication patterns; the optimization is dynamic at runtime, and this leads to limited optimization due to strict time constraints.

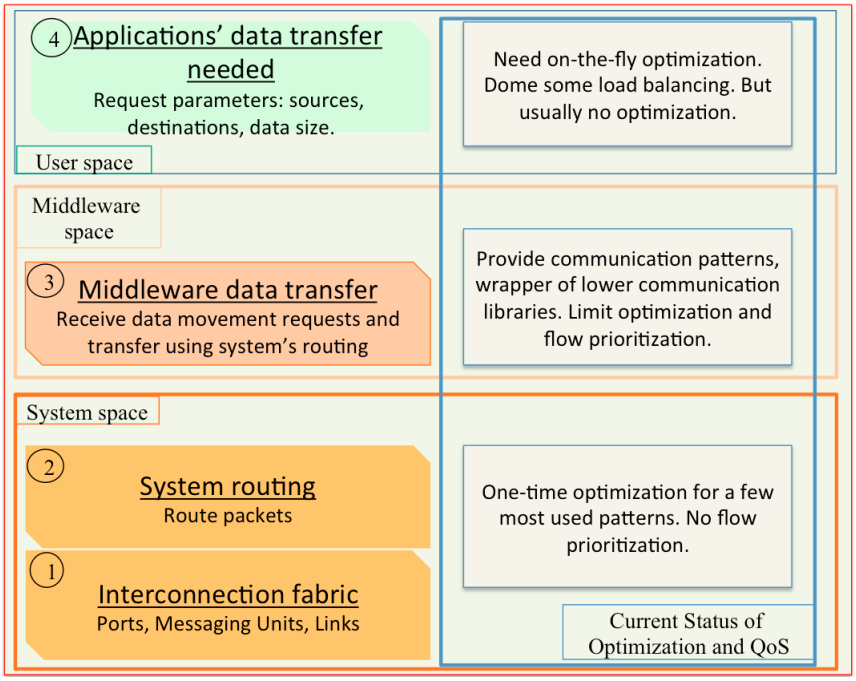


Figure 1: Data movement optimization at systems' layers

At the system routing layer, since switches are aware of the local traffic and may be partially aware of global traffic at the destination, a combination of local traffic aware with global path selection for the most used communication patterns can result in good performance. However, the systems still do not have global optimization for many other communication patterns and flow patterns at the application level.

Along with the increasing computing capability of compute nodes, the complexity and sophistication of the interconnection networks are also increasing. The current generation of supercomputers has various interconnection network topologies from 5D in the Blue Gene/Q supercomputer to Dragonfly in the Cray Cascade supercomputers.

At the middleware layer, highly optimized low-level communication libraries such as Parallel Active Messages Interface (PAMI) or User Generic Network Interface (uGNI) are working only on certain systems such as BG/Q and Cray respectively. High level and compatible communication libraries such as MPI, PGAS-family with different libraries (GASnet, Charm++, UPC, Chapel, X10...) provide common communication routines. They are usually wrappers of lower level communications libraries. Therefore, optimization on these libraries is optional and system-specific. I/O libraries such as MPI-IO, Adios, GLEAN also work well on specific systems under specific conditions.

At the application layer, application developers or scientists usually use the provided routines in these systems as-is (MPI routines are most common). Research has been done on load balancing and improving latency-sensitive communication. Nevertheless, optimization for various communication patterns is lacking at this layer.

In conclusion, the data of an applications that needs to be exchanged is converted into appropriate units that can be transferred by using one of the middleware libraries. The middleware libraries use lower level libraries to inject units of data into network in the form of small packets. The packets travel on physical links to their final destination and move in the opposite direction at the destination node until reaching the application level. At the application level, the data communication patterns are known but the availability of networking resources are not. At the system level, the availability of networking resources is known but the communication patterns is not.

The next chapter gives a detailed survey on data movement improvement and optimization on supercomputers and closely related work.

CHAPTER 3

RELATED WORK

In this chapter, the dissertation gives a survey of optimizing and improving data movement in supercomputers and closely related problems and solutions. Optimizing data movement in supercomputers can be done at different levels and with different approaches.

3.1 Works in data movement optimization

Data movement optimization with a specific network topology and communication matrix is equivalent to a multi-commodity flow problem (3). For this problem, if integral flow is required, the problem modeled as integer linear programming (ILP) formulation is NP complete. However, if fractional flow is allowed, there exists a linear programming (LP) formulation solvable in polynomial time. Several of the works discussed below use similar ILP or LP formulations.

Raecke *et al.* in (4) showed that an oblivious routing algorithm can be found within a symmetric network such that the contention is only poly-logarithmic increment with the optimum of any traffic patterns. However, Raecke's construction is not polynomial time. Azar *et al.* in (5) formulated the problem using linear programming such that it can be solved in polynomial time using Ellipsoid algorithm. However, the formulation grows exponentially with the network size. Applegate *et al.* in (6) concluded that even with a fairly limited knowledge of the applicable traffic demands, a robust routing that is nearly optimial is possible to obtain.

Kinsy *et al.* in (7) proposed an application-aware deadlock-free oblivious routing framework. The work presented a mixed integer-linear programming (MILP) approach together with a heuristic approach to produce deadlock-free application-aware routes that minimize latency. The framework used a heuristic algorithm to calibrate the MILP algorithm.

Rodriguez *et al.* in (8) proposed oblivious routing schemes in extended generalized fat tree networks. It extended two algorithms called S-mod-k and D-mod-k to provide a better oblivious solutions for slimmed networks.

Prisacari *et al.* in (9) showed that there is potential to optimize all-to-all collective exchange communication patterns at the system level in fat-tree networks and proved it mathematically and via simulations. The work is then extended in (10) to propose a generic method to determine optimal pattern-specific routing for eXtended Generalized Fat Tree (XGFT). The method uses a hybrid combination of integer linear programming (ILP) and dynamic programming. The interconnection network is divided into small subdomains and ILP is used for each subdomain. The local solution is then combined using dynamic programming. The proposed method takes up to several hours for several thousands of compute nodes in an interconnect network. In the work, the authors reduce the solving time by optimizing globally with dynamic programming and locally with ILP. However, the search space was still large and led to long search time. No heuristic was proposed to reduce the search time. The approaches in this dissertation present networks as graphs, so the approaches can work with any type of network. In order to reduce the search time, the search space is significantly pruned by feeding into the algorithms only a

number shortest paths. In order to validate the approach, Prisacari used simulation while this dissertation demonstrated actual implementation and experiments on a real system.

Optimizing data movement on supercomputers can also be done via task mapping. Agarwal *(*et al.) in (11) minimized the impact of topology in a process mapping strategy by heuristically minimizing the total number of hop-bytes communicated. However, the work focused on latency of data movement rather than throughput which is considered in this dissertation.

Valiant in (12) proposed a randomized routing mechanism mathematically proved to be able to route data globally with no sharing links. However the routing mechanism did not work with local traffic as shown in (13). Pifarre *et al.* in (14) proposed a minimal routing optimization. However, it did not work well with global optimization as shown in (13) . To address the limitation of both approaches, in (13), Singh et al. proposed GOAL – Globally Oblivious Adaptive Locally to balance load using adaptive routing algorithms for torus networks. GOAL balances the load by choosing the routing direction for each dimension randomly and routing in the selected direction adaptively. GOAL routes data at the system level and does not consider the application's data movement patterns. This dissertation leverages the system's routing and takes the application's data movement patterns into consideration.

In the Blue Gene/Q supercomputer (BGQ), Chen *et al.* in (15) proposed a heuristic routing for Blue Gene/Q supercomputing systems. The routing path is computed dynamically at routing time based on the coordinates of the source and destination, partition shape, and message size. The systems route a packet along the longest dimension of a partition first, shortest last. BG/Q systems route a message using single path. Different messages however

might be routed using different paths. In a similar work, Garcia *et al.* in (16) proposed two deadlock-free routing mechanisms that support on-the-fly adaptive routing on the Cray XC30 system for Dragonfly networks. However, both works were proposed for routing at the system level. The approaches in this dissertation are proposed as a supplementary work that leverages existing system routing and topologies to improve data transfer throughput. They take both the application's communication patterns and system's routing/topology into consideration.

At the middleware layer, communication libraries use low level libraries to provide common communication patterns for applications. The most common library specifications are MPI-2/3 and GASNet (17). Some optimizations are done based on system specific characteristics such as (18) who proposed optimization for MPI_Allreduce in BG/Q based on observations of the number of links and other hardware support on the compute nodes.

Data movement optimization at the application level has been very limited. Some work has been done on load balancing such as load balancing on FLASH for multi-level adaptive mesh (19).

For wide-area networks, Gaurav *et al.* in (20) proposed a file transfer scheduling algorithm that incorporates two key optimizations - multi-hop path splitting and multi-pathing. It is tested on various transfer patterns such as one-to-all broadcast, all-to-one gather, and data redistribution. (21) used linear programming to optimize data movement between multiple sites in a Grid environment.

Previous work in (22) and (23) addressed the problem of I/O data aggregation. In a more recent work (24) the authors presented approaches for improving the performance of sparse

Prior works	Optimization	Interconnection	Use case	Scalable
[Leighton95]	LP + Approximate algorithm	General	Offline	
[Azar03]	LP + Ellipsoid algorithm	General	Offline	No
[Kinsy09]	MILP + Heuristics	General	Offline	
[Rodriguez09]	Heuristics	Fat Tree	On-the-fly	Yes
[Prisacari13]	ILP + Dynamic programming	XGFT	Offline	Several thousands
[Valiant81]	Randomized	General	On-the-fly	Yes
[Gustavo91]	Minimal routing paths	General	On-the-fly	Yes
[Singh03]	Globally oblivious, adaptive locally	Torus	On-the-fly	Yes
[BG/L,P]	Heuristic	3D Torus	On-the-fly	Yes
[BG/Q]	Heuristic	3D Torus	On-the-fly	Yes
[Cray XC30]	Heuristic	Dragonfly	On-the-fly	Yes
[Kumar'13]	Heuristic	5D Torus	On-the-fly	Yes
[Khana'08]	Heuristic	Grid	LP	No

TABLE II: Previous works on optimizing data movement

communication patterns. This dissertation however presents approaches for data movement optimization with broader ranges of application communication patterns.

Table II summarizes related works and important features presented in this chapter.

In the next chapter, this dissertation presents OPTIQ, a framework to optimize data movement for data-centric applications on supercomputers.

CHAPTER 4

OPTIQ FRAMEWORK

4.1 Overall Approach

A data movement optimization framework(OPTIQ) is built to realize the holistic approach proposed in this dissertation. OPTIQ has four main components: Path generation, Path searching, Schedule, and Transport depicted in Figure 2.

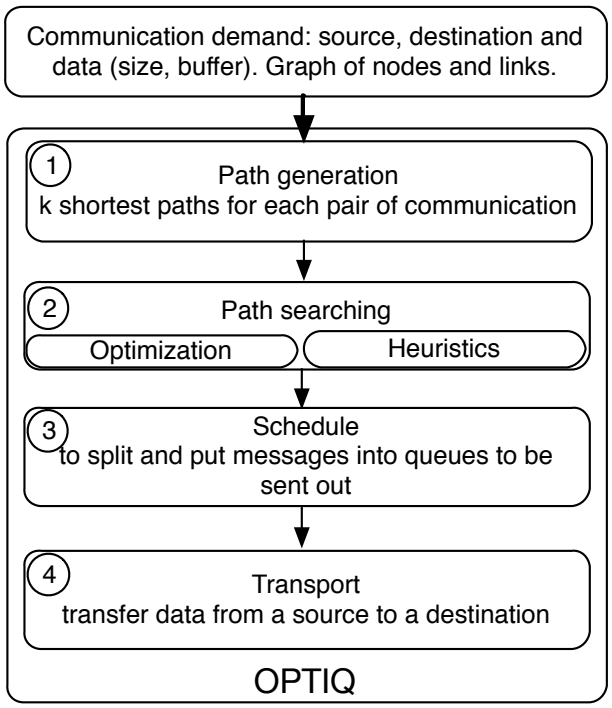


Figure 2: Components of OPTIQ framework

Applications that use the framework need to provide communication information including sources, destinations, and demands - the amount of data to be transferred between the sources and the destinations. The information is passed as an input for the framework.

Main functionality of each component is described as follows:

- Path generation: generates k shortest paths that can be used as candidates for data transfer. We need to generate paths to reduce the search space.
- Path searching: searches for paths to transfer data from a set of source nodes to a set of destination nodes. Multiple paths or a single path can be found using a set of algorithm. The user can decide what algorithm to be used or let the framework use a default algorithm.
- Schedule: splits a data buffer that needs to be transferred into smaller messages and puts the messages into a queue of the transport layer to be transferred. It also handles incoming messages by keeping messages for itself and forwarding other messages to their next destinations.
- Transport: transfers a message from one node to another node in the system.
- Auxiliary component: gets system specific information such as partition size, topology, coordinates, and torus, and computes the neighbors of available nodes given to an application.

4.2 Framework Components

In this section, the dissertation describes the details of the components.

4.2.1 Path Generation

Compute nodes in a supercomputer communicate and exchange data through an interconnection network. Data travels along physical network links of the interconnection network from sources to destinations. A set of physical network links that connect a source to a destination forms a path that can be used to transfer data from the source node to the destination node. For any pair of source and destination, there might be multiple paths that the framework can employ to transfer data. For interconnection networks on recent supercomputers, the number of paths connecting any two nodes is exponential. Thus, the framework needs to have a component to generate a finite set of paths to be employed for data movement. The important features of the component are as follows:

- Generated paths to be used as inputs for the Path searching component in the framework for path searching and demand assigning. A smaller number of generated paths (smaller search space) leads to shorter search time but might result in lower search quality (more shared paths, higher congestion). A larger number of generated paths (large search space) can lead to a longer search time but also potentially leads to higher search quality (less shared paths, lower congestion).
- The longer a path is, the longer the time data will be travelling along the path. But the Path searching component might need to employ longer paths to have a significantly larger search space.

Trading off between the above factors is important in the final result of improving data movement. The dissertation suggests a few options as follows:

- The component is allowed to generate at most k shortest paths between any pair of source and destination nodes that need to transfer data. The value of k can be determined empirically.
- The component is allowed to generate paths with the number of hops not more than the longest distance between any nodes in the current partition. In some supercomputers such as the Blue Gene/Q, this is the diameter of the partition is given for the application.

The framework provides a set of functions allowing it to generate paths, and write/read paths to/from a file. With the generated paths, the framework now can search for paths to be employed to transfer data. In the next section, the dissertation presents the path searching component.

4.2.2 Path Searching

The Path Searching component searches for paths from the given set of generated paths from the Path Generation component. It also assigns data to the selected paths.

The Path Searching component supports different search options for an application as follows:

- Online search: the search occurs while the application is running.
- Offline search: the search is carried out in advance before the application runs or part of the search is computed in advance.

- Hybrid: part of search is computed in advance and the remainder of the search is computed when more information is available.

The framework also provides different types of search allowing a trade off between search time and search quality:

- Heuristic search: shorter search time with local optimization solutions.
- Formal optimization mode-based search: longer search time with global optimization solutions.

The Path Searching component assigns data to the selected paths and passes the paths to the Scheduler component where data is scheduled to transfer.

4.2.3 Scheduler

As data is transferred from sources to destinations along the paths, it is routed from source nodes through intermediate nodes to the destination nodes. Any compute node can play from none to all of the roles below:

- Source: A source sends its data along paths with the amount given by the Path Searching component.
- Destination: A destination receives its data from source nodes. Data travels along paths might pass through intermedate nodes in between a source and a destination before reaching to the destination.
- Intermediate/Forward: An intermediate/forward node forwards data from a previous node to a next node on a path.

- Idle: An idle node does nothing.

The Figure 3 depicts the work of the Scheduler component at a node.

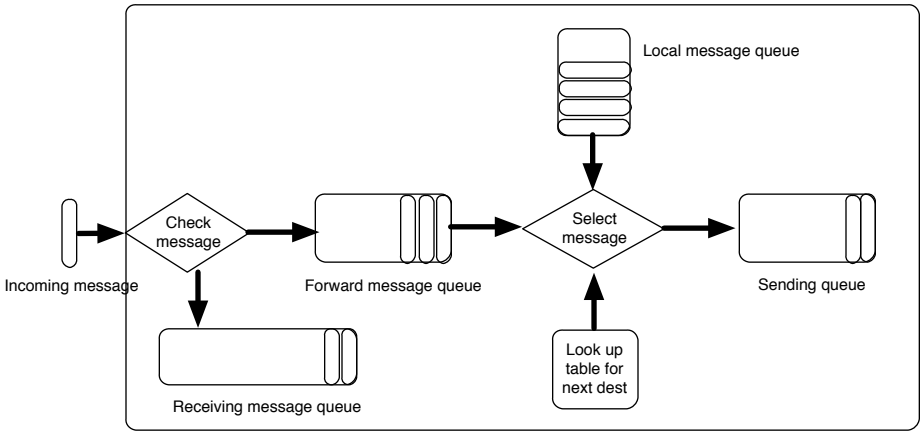


Figure 3: Scheduler component

At a node, the Scheduler needs to do the following:

- For incoming messages: Check if an incoming message is for itself or not. If the message is for itself, it puts the message into the *Receiving message queue*. If not the message needs to be forwarded to the next destination. The scheduler puts the message into the Forward message queue.

- For local or forward the sending queue. The order of selecting message from either queue can be either Forward first, Local first or Round Robin. The transport component selects messages in the sending queue for sending.

A message needs to contain information so that the Scheduler knows how to process the data. The information includes: message id, source, destination, size, original buffer offset, and path id. The Scheduler maintains a table of (path id, previous node and next node) for forwarding purposes. Whenever it receives a message it can check to see if the message has come to its final destination. If not, it looks up in the table for the next destination of the message.

When there are multiple tasks per node, the Scheduler can select any tasks in the next node to be next destination. However, the selection mechanisms must be uniform for at all tasks.

Messages in the sending queue are selected on a first-come-first-serve order by the Transport component.

4.2.4 Transport

The Transport component selects a message from the *sending queue* and transfers the message from its source node to its destination node. The Transport component can employ any available libraries for data movement available for transferring messages such as MPI, uGNI, PAMI, Charm++, UPC... However, it prefers to choose low level libraries for better communication performance for small messages.

The framework exposes an application programming interface that allows users to use or develop different communication libraries.

4.2.5 Auxiliary Component

The framework also has a set of functions to read system's information such as topology, coordinate, torus, size, and routing information, to compute the neighbors and generate the graph for a partition given for an application.

In this chapter, the dissertation presents OPTIQ framework, a realization of the holistic approach. In the next chapter, the dissertation demonstrates a development of the framework for Path generation, Path searching components for multi-path data movement.

CHAPTER 5

MULTI-PATH DATA MOVEMENT

5.1 Multi-path Data Movement/Routing

In large-scale systems such as the Blue Gene/Q (BG/Q), data is routed through its interconnect using the default routing algorithms. They perform well for some communication patterns (15). However, for certain communication patterns (shown later in this dissertation), the default routing algorithms result in poor performance due to unbalanced load on the physical network links. This results in a significantly larger amount of data being transferred over few links. On the BG/Q, the default algorithm uses a single path to transfer data between any two nodes in the system. In addition, data traverses along fixed paths on certain links using a static routing regardless of the overall load of the system. Thus, some links are overloaded while other links have less data or may even be idle. This overloading is a major bottleneck for data transfer throughput. Balancing the load while exploiting multiple physical network links can improve the throughput.

The above problem can be formulated as a multi-commodity integral flow problem, which is shown to be NP-complete (25). Given a set of source and destination nodes, and the amount of data to be transferred from the sources to destinations, the problem is to find a set of paths from the source nodes to the destination nodes that result in high throughput. Additionally, the objective is to balance the overall system load in order to avoid congestion in the interconnect,

and to avoid overloading the physical network links. In this chapter (Sections 5.3.1, 5.3.2 and 5.4.1), we propose three approaches to solve this problem, taking into consideration the system topology. The first approach is a greedy heuristic that select paths to minimize the maximum number of paths on the physical network links. The second approach is also a greedy heuristic that selects paths so that the load is balanced on the links. The third approach is an optimization based model that selects globally optimal paths for source-destination pairs.

We leverage the idle or lightly-loaded links for data transfer in order to balance the load. For this we need to search for multiple paths between source and destination nodes and assign an appropriate load on each path. Present-day supercomputers have thousands of nodes and hundreds of thousands of edges due to their complex interconnect topology. This implies a large search space for multiple paths. The brute-force approach of searching for paths can lead to significant amount of time being spent on searching load-balanced paths. We use Yen's algorithm (26) to search for a set of shortest paths. To reduce the search time we prune the search space by constraining the number of hops on each path.

5.2 K Shortest Paths

In order to search for paths, we model the interconnect network as a graph. Each compute node is modeled as a vertex and each physical link is modeled as an edge. Algorithm 1 depicts the algorithm to generate k shortest paths from a source to its destination, where k is an input to the algorithm.

The input to the Algorithm 1 includes a set of pairs of source-destination (s_i, d_i), a graph of nodes and a desired number of shortest paths k for each pair (s_i, d_i). The output is a set of

```
   Input: Input:
1  Set of pairs of source-destination (s_i, d_i). Graph of nodes. Desired number of shortest
   paths k.
   Output: Output:
2  Set of paths: k paths for each source-destination pair.
3  Procedure FindKShortestPaths()
4      for each pair of (s_i, d_i) do
5          while (Less than k paths have been discovered and there are more paths available)
           do
6              Use Yen's algorithm to search for the shortest path p;
7              if Number of hops of p ≤ partition's network diameter then
8                  Add p into k_paths;
9              end
10         end
11     end
```

Algorithm 1: k shortest paths generation.

at most k paths for each pair of source and destination node. We search for a path by iterating through the set of pairs (line 3). For each pair, we search for one shortest path at a time using Yen's algorithm. If the found path's number of hops is at most the partition's network diameter, we add the path into the set of *k_paths* of the pair (line 5-8). The search completes when we either have k paths or there are no more paths found by Yen's algorithm (line 4). In the next sections, we present our heuristic approach and optimization model that use *k_paths* for finer searches for paths between pairs of sources and destinations.

5.3 Heuristic approaches

5.3.1 Heuristic Approach 1

In this section we describe our heuristic to select paths for each source-destination pair, given the set of k shortest paths k_paths. In this approach we assume that same amount of data is transmitted over each path. Thus the total load on a link can be represented by the number of paths using the link. In order to avoid overloading of the physical links and to achieve high performance, we select paths in such a way that satisfies the following two conditions:

1. Select as many paths as possible for any pair of source/destination.
2. The maximum number of paths on any physical link is less than a given *maxload* value.

The above conditions ensure that we have multiple paths for each pair, and the load is balanced, and it is within the upper limit of the maximum load on the physical links. The brute force algorithm explores all combinations of paths for all source-destination pairs and examines the above two conditions in order to find the best set of paths. However its time complexity is exponential. We propose a heuristic that outputs a set of paths without exhaustively searching the entire search space. Our heuristic iterates through all pairs of source/destination nodes to search for more paths until the *maxload* value is reached. The pseudocode of our approach is presented in Algorithm 3. The algorithm is composed of two functions `HeuristicSearch` and `FindPaths`. The function `HeuristicSearch` invokes `FindPaths` until paths for all source-destination pairs are found.

Input: Set of source-destination pairs $\mathcal{R} = \{(S, D) \mid S, D \in N\}$ and their k shortest paths k_paths. Maximum allowed load on a link $maxload$. $N \times N$ physical link load matrix $load$.

Output: Set of selected paths for data movement: s_paths.

```
1  Procedure FindPaths()
2      while (R != φ) do
3          Select (s,d) from R
4          Let i be the index of (s,d)
5          p ← Select a path from k_paths_i
6          k_paths_i = k_paths_i - p
7          links ← set of links in path p
8          if (load(l) + 1 ≤ maxload ∀l ∈ links) then
9              s_paths_i = s_paths_i ∪ p
10             Update load ∀l ∈ links
11         if (k_paths_i == φ and s_paths_i != φ) then
12             Remove (s,d) from R
13         if (k_paths_i == φ ∀i) then
14             break
15     end
17     return
18 Procedure HeuristicSearch()
19     k_paths_i = call FindKShortestPaths()
20     while (s_paths_i != φ ∀i ∈ R) do
21         call FindPaths()
22         maxload++
23     end
25     return
```

Algorithm 2: Heuristic to search paths for each source-destination pair from k shortest paths.

The input to the algorithm includes the set of source-destination pairs $\mathcal{R} = \{(\text{S, D}) \mid \text{S, D} \in N\}$, their k shortest paths k_paths, the maximum allowed load on a link $maxload$ and the link load matrix $load$. $load$ is a table of loads on all physical links. Whenever a link $l = (u, v)$ is selected by the heuristic the corresponding entry $load(l)(= load[u][v])$ is incremented by 1. The heuristic outputs a set of selected paths s_paths for data movement for each source-destination pair.

`HeuristicSearch` (lines 18–25) finds paths for all source-destination pairs in $\mathcal{R}$. `FindPaths` is invoked until there exist some paths between all source-destination pairs. In `FindPaths`, we iterate through all pairs of source-destination in $\mathcal{R}$ and add at most one path per pair at a time. For each source-destination pair, we select a path p from k_paths (lines 3–5). The selected path is removed from the set k_paths_i (line 6), where i is the index of the selected source-destination pair. A path from the source node to the destination node is a set of links through intermediate nodes. Let this set of physical links be denoted as $links$ (line 6). We check if adding p to s_paths violates the load condition for links in p (lines 8–10). If current load $load(l)$ on all links $l \in links$ is below $maxload$, the path p is added to s_paths_i (line 9). Also, the $load$ table is updated for all links $l \in links$ (line 10).

If all the k shortest paths have been used for a pair, we remove the pair from $\mathcal{R}$ (lines 11–12). The function `FindPaths` returns when k_paths is empty for all source-destination pairs (lines 13–14). At this point, we increase the maximum load limit $maxload$ by 1 and re-invoke `FindPaths` from `HeuristicSearch` (line 21–22). Once there exist paths for all source-destination pairs, the

algorithm terminates. We then divide the data for each source-destination pair equally among the selected paths for the pair.

5.3.2 Heuristic Approach 2

In this section we describe our heuristic to select paths for each source-destination pair, given the set of k shortest paths k_paths. In this approach we assume pairs of sources-destinations have different amounts of data called *demand*. Data of a pair is split into smaller chunks and assigned into its set of paths in such a way that minimizes the maximum total amount of data assigned on physical links. At each time of assignment, we only assign an amount of *chunk*. This allows us to iterate through all pairs and assign *chunk* into their paths. After assigning a *chunk* to a path, all physical links comprising the path have an additional *chunk* on its load.

Each path maintains a *maxload* value, which is the maximum value of current loads on physical links comprising the path. After an assignment of a *chunk* to a path, not only the path's *maxload* value needs to be checked and updated but any paths that share the path's links also need to be checked and update their *maxload* values.

In order to reduce the total amount assigned on any physical link, we allow the pair with the largest amount of data to select a path to assign *chunk* amount of data first. We maintain a max heap *heap* of pairs with the pair with the largest remaining data amount being at the top of the heap. The assignment is as follows. We pop the pair at the top, select in its k_paths a path with minimum *maxload*, assign a *chunk* of data to the path and update its remaining data. If there is still data to be assigned, we push it back to the *heap* and do heapify. We update the *maxload* value on the paths and loads on the physical links. Our assignment finishes when

all pairs assign all of their data to their paths. The pseudocode of our approach is presented in Algorithm 3.

```
Input: Set of source-destination pairs P = {(S, D) | S, D ∈ N} and their k shortest
       paths k_paths. Data size chunk for each assignment of data to paths.
Output: Set of source-destination pairs P_out with assigned data for each path.
1  Procedure FindPaths()
2      Make heap heapP from P.
3      while (heapP != φ) do
4          Heapify the heapP
5          pair = heapP.pop()
6          Select a path p in pair's k_paths with minimum maxload value
7          Assign chunk data to p
8          Update maxload value of p and any paths that use p's physical's links and
           corresponding physical links
9          pair.demand -= chunk
10         if pair.demand > 0 then
11             heapP.pushback(pair);
12         else
13             P_out.add(pair)
14         end
15     end
```

Algorithm 3: Heuristic to search paths for each source-destination pair from k shortest paths.

In the Algorithm 3, we pick the pair with largest amount of remaining data first. This allows its paths and corresponding physical links to be selected first. The pairs with lower remaining amounts of data can select paths later. Thus, load is balanced between pairs with higher load

and pairs with lower load. Among all paths of a pair, the path with least *maxload* is selected first. Thus, the load is balanced among paths belonging to the same pair.

The data assignment in this heuristic approach is greedy and uses local optimization. In the next section, we present the second approach, in which we employ a mathematical model to optimize data movement by assigning an optimal amount of data to the paths.

5.3.3 Comparison between Heuristic 1 and Heuristic 2

Both Heuristic 1 and Heuristic 2 share a common goal of minimizing maximum load on physical links. However, they differ from each other by what they consider the load. While Heuristic 1 assummes that all pairs of sources and destinations have a similar number of paths and each path carries a similar amount of data. Thus, Heuristic 1 considers the number of paths as the load while Heuristic 2 considers the actual amount of data as the load. This would lead to a better data distribution among the selected paths for Heuristic 2. But the trade-off is that we need to know the amounts of data in advance, and without knowing it, we cannot use Heuristic 2. In that case, Heuristic 1 is appropriate to use.

The Table III shows the comparison between Heuristic 2 and Heuristic 1.

5.4 Modeling Approach

5.4.1 Optimization-based Approach

In this approach, we find the optimal assignment of data to transfer along multiple paths from source nodes to destination nodes. Given the amount of data to be transferred, and the k shortest paths from source to destination nodes (Algorithm 1), we formulate the problem of finding paths for transferring data from sources to destinations as an optimization problem.

Comparison factors	Heuristic 1	Heuristic 2
Load on physical link	Number of paths that used a link.	Actual amount of data passing through a link.
Pair iteration	Each pair 1 time to get 1 path.	The pair that has the largest amount of remaining data.
Use case	Do not need to know amounts of data of pairs in advance	Need to know amounts of data of pairs in advance.

TABLE III: Comparison between Heuristic 1 and Heuristic 2

The objective of the optimization problem is to minimize the total transfer time by finding paths that are uniformly loaded. Next, we describe the problem parameters.

The data transfer request of a source-destination pair is denoted as a job. The set of all jobs is denoted by $Jobs$. Each job has an amount of data to be transferred from its source to its destination. This amount of data for job job is denoted by $Demand[job]$, which can be transferred over selected paths from a pre-computed set of paths $kpaths_{job}$ (computed prior in Algorithm 1). A path is denoted by p. The amount of data transferred per unit time for job job on path p is called a flow of the job and is denoted as $flow(job, p)$. A path p might comprise one or more links. (i, j) denotes a link from vertex i to vertex j. All links on path p accommodate the same flow. $flow(job, p)_{ij}$ denotes the flow over link (i, j) of path p. $c(i, j)$ denotes the capacity of link (i, j). The link capacities are known link bandwidths of the interconnect. The objective of the optimization problem is to minimize time t to transfer data for all jobs in $Jobs$ subject to two constraints. The decision variables are the set of selected paths for a job and the flow along each path $flow(job, p)_{ij}$. We describe our linear program formulation below.

Objective function:

$$\text{minimize } t$$

Constraints:

- Total flow of a job is equal to the sum total of all its flows along its paths. For any job in $Jobs$:

$$\sum_{\forall p \in kpaths_{job}} flow[job, p] = \frac{Demand[job]}{t} \tag{5.1}$$

- Total flow of an arc is less than its capacity. For any arc (i, j):

$$\sum_{\forall job \in Jobs} \sum_{\forall p \in kpaths_{job}} flow[job, p]_{ij} \leq c(i, j) \tag{5.2}$$

The first constraint in Equation Equation 5.1 captures a job's flow distribution. The total amount of data of a job $Demand[job]$ is transferred in a time t through a set of paths in $kpaths_{job}$. For each path, the job job is assigned a throughput $flow[job, p]$. The total data transferred for a job must be less than $Demand[job]$. Thus, the job's throughput (rate of data transfer) needs to be equal to the combined throughput on all its paths. The second constraint in Equation Equation 5.2 captures the bound for a link's capacity. The total throughput of all jobs on a link should not exceed the link's capacity.

We used AMPL (A Modeling Language for Mathematical Programming) (27) to model the optimization problem. We used the SNOPT solver to solve the linear program. The solution

times are listed in Section 6.4.4.10. In the next section, we present our experiments and results to demonstrate the efficacy of our approaches.

5.4.2 A Mathematical Programming Language (AMPL)

A Mathematical Programming Language (AMPL) is an algebraic modeling language (28). It can be used for describing and solving high-complexity, large-scale optimization and scheduling-type problems. AMPL supports a number of both commercial and open source solvers such as SNOPT, CPLEX, MINOS... AMPL is chosen due to the similarity of its syntax to mathematical notation of optimization problems.

AMPL uses the following notations in describing a problem:

- Sets: are one of the most fundamental concepts in AMPL. They can be used to index variables, constraints, parameters or even other sets.
- Parameter: are constant values that are not changed during the solving time. Users can assign values for parameters at the beginning of solving time.
- Vars: are used to represents for variables in mathematical programs. Solvers are allowed to change them while looking for solutions.
- Objective function: is a function of decision variables. This is either maximized or minimized.
- Constraints: are conditions that solvers need to satisfy while looking for solutions.

In the next section, the dissertation presents the problem in AMPL.

5.4.3 AMPL Model

The model is written in AMPL (A Mathematical Programming Language). The model is described in **Model** 1.

```
set Nodes;
set Arcs within Nodes cross Nodes;

set Jobs;
set Paths{Jobs};
set Path_Arcs{job in Jobs, p in Paths[job]}
    within Arcs;

param Capacity{Arcs} >= 0 default Infinity;
param Demand {Jobs} default 0;

var Flow {job in Jobs, Paths[job]} >= 0;
var Z >= 0;

maximize obj: Z;

subject to

demand_con {job in Jobs}: sum {p in Paths[job]} Flow[job,p] = Demand[job]*Z;

capacity_con {(i,j) in Arcs}:
  sum {job in Jobs, p in Paths[job]:
    (i,j) in = Path_Arcs[job,p]} Flow[job,p] <= Capacity[i,j];
```

Algorithm 3: Model 1 Data movement optimization

The notions used in **Model** 1 are explained as follows:

- sets:
  - *Nodes*: set of nodes in the network, each node represent a compute node in the supercomputer.
  - *Arcs*: set of arcs in the network. Each arc represent a physical link in the supercomputer.
  - *Jobs*: set of jobs. Each jobs has a source and a destination.
  - *Paths*: set of paths for each job.
  - *Path_Arcs*: set of arcs on each path of each job.
- params:
  - *Capacity*: capacity of each arc i.e. bandwidth of the physical link.
  - *Demand*: amount of data to be transferred of each job between a pair of source and destination.
- vars:
  - *Flow*: flow of each job on a path. It can be seen as the proportional bandwidth assigned for the job on that path.
  - Z: is reversed of total time.
- objective function: we want to minimize the time or maximize its reversed value i.e. maximize Z.

- constraints(subject to):
  - *demand_con*: flow of a job on equals to the demand of the job divided by the transfer time.
  - *capacity_con*: total flow on an arc is less than its capacity.

The model takes a set of nodes, a set of arcs and their corresponding capacity, a set of jobs (source/destination pairs), a demand for each job, a set of paths for each job, and a set of arcs for each path as inputs. It searches for an assignment of flow values (proportional capacity) for paths of all the jobs such that the transfer time for demands of all jobs is minimum.

We feed the model into solvers together with data of nodes, arcs, capacity, paths for jobs and get the paths with given proportional bandwidth. Based on proportional bandwidth, each path can take proportional demand of a job.

CHAPTER 6

IMPLEMENTATION AND EVALUATION

6.1 Mira

The Argonne Leadership Computing Facility (ALCF) maintains several compute-analysis systems used by the scientific community. In this section, the dissertation describes the supercomputing systems in which the framework was developed and tested. Figure 4 depicts the architecture of the primary LCF resources, consisting of the Blue Gene/Q compute cluster (Mira), the data analysis cluster (Tukey), and the file server nodes.

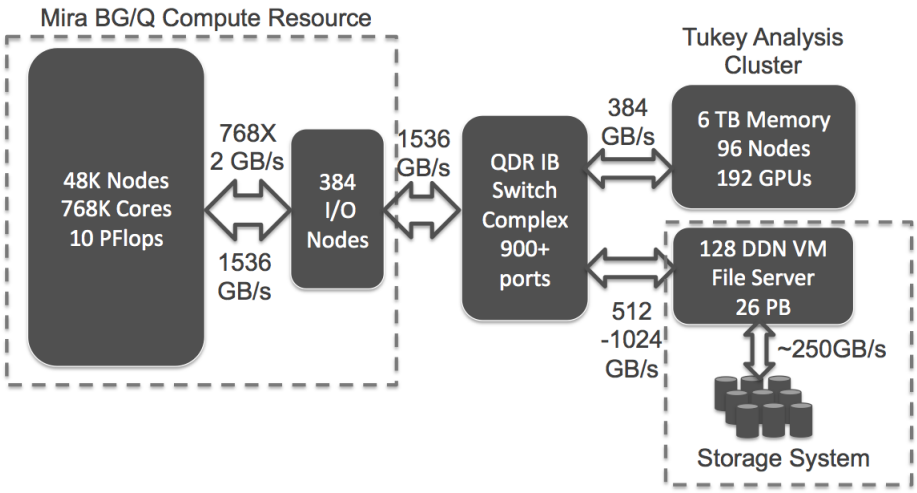


Figure 4: The ALCF maintains the 768K core Blue Gene/Q compute cluster (Mira), data analysis cluster (Tukey), and file server nodes.

The Blue Gene/Q system (15), Mira, with 48 compute racks (48K nodes and 768K cores) at the ALCF provides 10 PFlops theoretical peak performance. Each node has an 18-core, 64-bit PowerPC A2 processor, together with 32KB cached L1, 32 MB cache L2 and 16 GB of memory.

The I/O and interprocess communications of the Blue Gene/Q travel on a 5D torus network both for point-to-point and for collective communications. This 5D torus interconnects a compute node with its 10 neighbors at 2 GB/s theoretical peak over each link in each direction, making a total of 40 GB/s bandwidth in both directions for one single compute node. However, due to packet and protocol overheads, up to 90% of the raw data rate (1.8GB/s) is available for user data. The machine can be partitioned into non-overlapping rectangular sub-machines; these sub-machines do not interfere with each other except for I/O nodes and corresponding storage system.

An overview of the network is also given in (29) and (30). Each compute node has 11 send units and 11 receive units, 10 for the 10 links of the torus and 1 for the I/O link. All packets are injected into and pulled out of network injection/reception FIFOs by the Messaging Unit (MU). The number of FIFOs is enough to saturate all links. Outgoing packets can be put in any injection FIFOs and may go out to any link. However, incoming packets at a receiver are placed only in its reception FIFO.

For interconnect network traffic, BG/Q supports both deterministic and dynamic routing (15). In deterministic routing, packets are routed based on dimension-ordered routing; packets are routed along the longest first to the shortest last. In dynamic routing, routing is still dimension-ordered however programmable, enabling different routing algorithms to be used. It

is called “zone routing”. There are four zone ids from 0 to 3. Routing algorithm select a zone id based on the flexibility metric and message size. The flexibility value is computed based on the torus size and hop distance between two nodes doing communication. The selection of a zone id given the values is experiment-based and is hard coded in the low-level library (31). Among the 4 routing zones, routing zone id 1 is unrestricted routing in which packets are routed in a random order. Routing zone id 0 is longest-to-shortest routing. However, dimensions with the same lengths can be chosen randomly. Routing zone ids 2 and 3 are deterministic routing. For these two routing zone ids, given the size of a certain message, routing is always the same and its path is known before it is routed. These are the default routing algorithms and the default routing algorithm cannot be changed during run time. However, the routing zone id can be set by using the PAMI_ROUTING environment variable. As BG/Q uses single path data routing, for sending/receiving a message, only one link is used out of 10 links available. Hence, there is one reception FIFO at a receiver. In addition, for point-to-point communication, the number of hops between 2 nodes has negligible effect on performance.

With respect to I/O traffic on the Mira BG/Q system, the compute nodes connect to an analysis cluster and the file servers through the I/O nodes and a QDR IB Switch Complex. Every 128 compute nodes (forming a pset) has two bridge nodes; two nodes in the pset have an additional functionality as a bridge node. Each bridge node has a 11th 2GB/s-bandwidth link connecting to an I/O node, making total 4 GB/s bandwidth for I/O per pset. I/O traffic is routed from compute nodes to bridge nodes over the torus network deterministically, and then traverses over the 11th links from bridge nodes to I/O nodes

Parallel Active Message Library (PAMI) is a lower level communication library for BG/Q (32). PAMI provides low overhead communication by using various techniques such as accelerating communication using threads, scalable atomic primitives, lockless algorithms to speed up messaging rate. As MPI is implemented on top of PAMI, direct use of PAMI would provide higher messaging rates as well as lower latencies in comparison with MPI.

6.2 Application Programming Interface (API)

The framework provides an application programming interface (API) that allows users to develop, extend and use the framework easily. The API is generic so that it can be extended on different supercomputers requiring minimal effort. The disseration presents some simple examples showing the framework's API, its functions, usage, and extensibility.

The Table IV shows several examples of the API. The framework needs an initialization method to set up before applications can use it and a finalization method to terminate the framework when the applications no longer use it. The framework also needs to read system information such as topology, coordinates, torus, node id. It also provides different methods to generate paths, search for paths, create schedules, and transport at low layers.

In order to make it easy to integrate into existing applications, the framework allows applications to hand over all the communication work. Table V shows two examples of using the API for data movement between set of nodes and I/O data movement. The inputs in the two cases from the applications are the similar to *MPI_Alltoallv* and *MPI_File_write*.

In the first case, the framework provides *optiq_alltoallv* that has the same inputs as *MPI_Alltoallv* so the usage is simlar. The source code example is shown in **Source Code Example 1**.

Functionality	**Methods**	**Parameters**	**Purpose**
Initialization & Finalization	optiq_init	int argc, char **argv	Call other inits to initialize the framework
	optiq_finalize	None	Call other finalization methods to terminate the framework
Topology	optiq_topology_init	None	To init topology
	optiq_topology_get_size	int *size	To get topology's size e.g 2x4x4x4x2
	optiq_topology_get_nodeId	int *nodeId	To get a node id
	optiq_topology_get_coordinate	int *coord	To get coordinate of node
	optiq_topology_get_torus	int *torus	To get torus of a partition
	optiq_topology_finalize	None	To finalize topology
Algorithm	optiq_alg_init	None	To init algorithms
	optiq_alg_generate_kpaths	char *graphfile, std::vector<struct job> &jobs, int num_paths	To generate k shortest paths
	optiq_alg_search_heu	std::vector<struct job> &jobs, enum alg	To search for paths using heuristic algorithsm
	optiq_alg_finalize	None	To finalize algorithms
Schedule	optiq_schedule_init	None	To init schedule
	optiq_schedule_create_schedule	std::vector<struct job> &jobs	To generate a schedule for the current jobs
	optiq_schedule_finalize	None	To finalize schedule
Transport	optiq_transport_init	None	To init transport
	optiq_transport_execute	None	To execute data movement based on the schedule
	optiq_transport_finalize	None	To finalize transport
Data movement	optiq_alltoallv	void *sbuf, int *scounts, int *sdispls, void *rbuf, int *rcounts, int *rdispls	To transport data as MPI_Alltoallv
	optiq_execute_jobs_from_file	char *jobfile, int datasize	To execute a data movement written in a file

TABLE IV: Examples of OPTIQ framework's API

```
void *sendbuf, recvbuf;
int *sendcounts, *senddispls, *recvcounts, *recvdispls;

/* Applications to init the buffers, counts and displacements
 * for sending and receiving data
 */

/*To move data*/
In MPI:
MPI_Alltoallv(sendbuf, sendcounts, senddispls, recvbuf, revcounts,
recvdispls);

In OPTIQ:
/* Initialize OPTIQ once at the beginning */
optiq_init();

/* Move data */
optiq_alltoallv(sendbuf, sendcounts, senddispls, recvbuf, revcounts,
recvdispls);

/* Finalize OPTIQ once at the end */
optiq_finalize();
```

Application needs	**MPI**	**OPTIQ**
Moving data form a set of source nodes to a set of destination nodes	MPI_Alltoallv	• optiq_init at the begining • optiq_alltoallv • optiq_finalize at the end
File I/O with paths pre-calculated from file	MPI_File_write	• optiq_init at the begining • optiq_execute_jobs_from_file to aggregate data to bridge nodes • MPI_File_write at the bridge nodes • optiq_finalize at the end

TABLE V: Examples of API usage by applications

In the second case, the framework needs to do file I/O. In this case, the paths can be computed in advance as bridge nodes, I/O nodes, and compute nodes are fixed. When compute nodes need to do I/O, they can call either *MPI_File_write* or they can call *optiq_execute_jobs_from_file* to aggregate data first and then call *MPI_File_write* only at the bridge nodes to write the data out. Aggregating I/O data using OPTIQ framework is faster than the MPI default mechanism as shown in the benchmarks for applications in 6.5.2. The source code example of this case is shown in **Source Code Example 2**. The usage of the OPTIQ framework in the applications is quite simple and straightforward, making minimal changes needed in the applications.

In the next section, the disseration presents some details of the implementation.

```
void *buf, *recvbuf /* To receive data at bridge node */;
MPI_File fh;
MPI_Offset offset;
char *filename;
MPI_Comm comm;
int count, recvcounts, amode = MPI_MODE_CREATE | MPI_MODE_WRONLY, rankID;
MPI_Info info = MPI_INFO_NULL;
MPI_Status status;

/* Application prepare data to write to file */

/* Open file */
int MPI_File_open(comm, filename, int amode, info, &fh);

In MPI:
/*All nodes that write data exchange lengths to get offsets then do I/O*/
MPI_File_write_at(fh, offset, buf, count, MPI_BYTE, &status);

In OPTIQ:
/* Initialize OPTIQ and read paths to bridge nodes, once at the beginning */
optiq_init();
char *pathfile = "paths_to_bridge_nodes_file";
std::vector<struct job> jobs;
optiq_job_read_jobs_from_file(jobs, pathfile);

/*Assign data buffer and length*/
for (int i = 0; i < jobs.size(); i++) {
    if (jobs[i].source == rankID) {
        jobs[i].demand = count;
        jobs[i].buf = buf;
    }
}

/*Aggregate data*/
optiq_execute_jobs(jobs);

/*If bridge nodes, the get the offset and write data*/
if(isBrigeNode) {
    MPI_File_write_at(fh, offset, recvbuf, recvcount, MPI_BYTE, &status);
}
/* Finalize OPTIQ once at the end */
optiq_finalize();
```

Algorithm 3: Source Code Example 2: Using OPTIQ to aggregate data before I/O

6.3 Implementation Details

In the implementation, the framework alters the data movement paths. Therefore, at each source node, the framework splits the data of each path into smaller chunks and enqueues these chunks into a *send* queue. Each node also has a *forward* queue to store the data it receives from its sender before relaying to its receiver on the data transfer path. When a node receives a chunk, it checks if it is the final destination of the chunk. If not, it copies the chunk to the *forward* queue, from which messages are injected into the network in order. The Scheduler component checks both queues and selects a chunk from either queue to transfer. The chunk size of 64 KB was empirically found to result in good performance on average. In all the experiments, the reported results include the overhead of queueing, copying etc. in the implementation.

For transferring data, the framework employed PAMI in the BG/Q. PAMI is a low-level communication library for BG/Q (32). PAMI provides low-overhead communication by using various techniques such as accelerating communication using threads, scalable atomic primitives, and lockless algorithms to increase the messaging rate. Since MPI is implemented on top of PAMI, direct use of PAMI would provide higher messaging rates as well as lower latencies in comparison with MPI. The framework used PAMI_Put for large messages and PAMI_Send_immediate for control messages.

PAMI supports both one-sided and two-sided communication. It supports both small immediate communication and rendezvous large-message communication. In this implementation, the framework uses one-sided communication for message transfer and immediate send for con-

trol data. In sending and receiving data events, PAMI supports a callback function to let a sender and receiver know whether the event of sending/receiving has been done at either side.

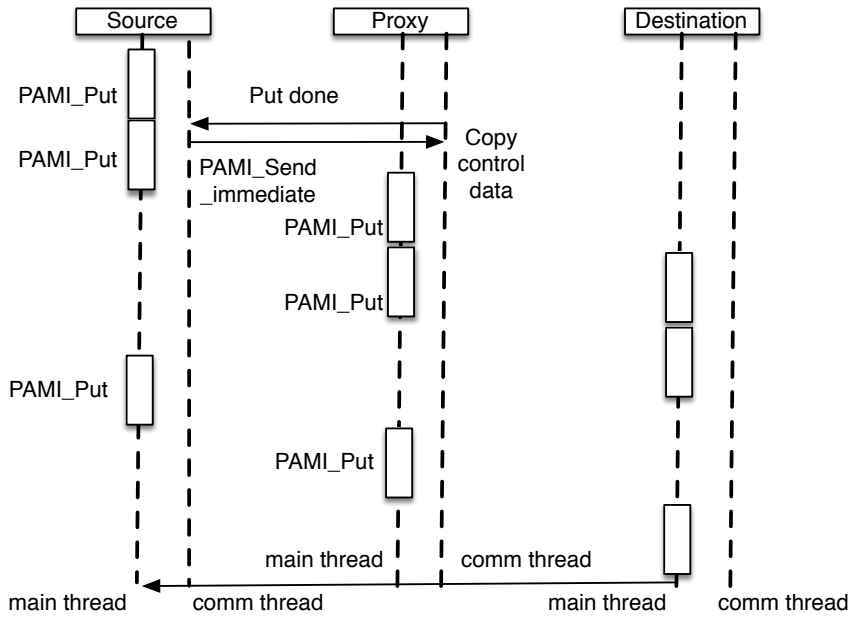


Figure 5: Using pipelining technique with PAMI to eliminate the waiting time at proxy and reduce control overheads

As depicted in Figure 5, in the main thread of the source, the framework will keep transferring data to windows of its proxies using *PAMI_Put*. Its comm thread running in the background is notified whenever the data is completely put on a proxy's side. The comm thread then uses *PAMI_Send_immediate* to let the proxy know that the data is ready. It also sends the control

data of where and how to process the data with the *PAMI_Send_immediate*. Each proxy needs to set up a callback function to process the control data sent to it. The callback function copies control data to a queue and informs the main thread. The main thread at each proxy plays the same role as the main thread of the source node. The size of the window for each message size is also determined empirically.

6.4 Evalution with Synthetic Benchmarks

In this section, the dissertation demonstrates the efficacy of the proposed approaches through multiple experiments on a leadership-scale system. The desseration describes the system, the implementation details, the experimental setup and present the results in the following sections.

6.4.1 Setup

The dissertation shows evaluations of the proposed approaches on Mira by varying the partition size from 512 nodes to 4096 nodes, varying the number of sources and destinations, and the average distance between them. There are experiments with different data sizes to be transferred, various combinations of source-destination pairs. The number of shortest paths used by our approaches was 50. The maximum load *maxload* for our heuristic approach was 16.

6.4.2 Communication Patterns

The dissertation demonstrates the data movement performance of the OPTIQ framework and existing MPI routines on three communication patterns – *disjoint*, *overlap* and *subset*, illustrated in Figure 6. In this figure, m and n refer to set of source or destination nodes. These

patterns are owing to different possible relationships between source and destination nodes as described below:

- Disjoint: There are distinct sources and destinations. It is a common data movement pattern present in many applications.
- Overlap: The sources and destinations are overlapped sets. Some applications like CESM uses this communication pattern for coupling.
- Subset: Either the set of source nodes is subset of the destination nodes or vice versa. This pattern can be found in CESM and in collective I/O aggregation phase.

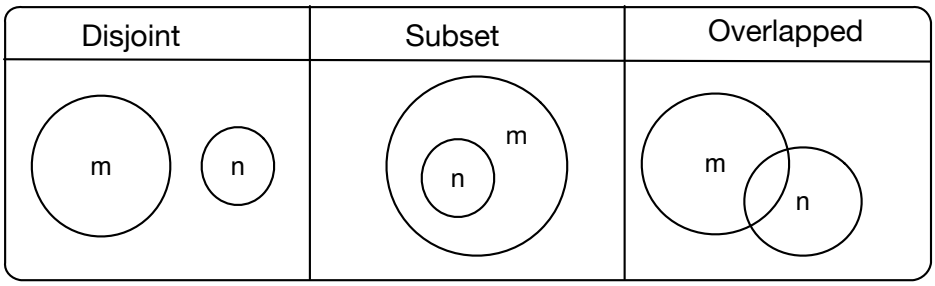


Figure 6: Communication patterns

The dissertation demonstrates throughput improvement in a complex network like the 5D torus through the above diverse communication patterns. It compares the efficacy of the pro-

posed algorithms with MPI_Alltoallv, which is the most commonly used MPI collective for the data movement patterns considered in this research.

6.4.3 MPI Path Reconstruction

Several network-related metrics are measured such as load on physical links, and the number of hops per data-transfer path, for the above benchmarks, using the proposed approaches and MPI_Alltoallv. The load and number of hops highlight performance differences between MPI and OPTIQ. The performance metrics are output directly in case of OPTIQ. However, MPI_Alltoallv does not output all of these performance metrics. Thus, we need to reconstruct the data-transfer paths taken by MPI collectives. The MPI paths are reconstructed based on the routing algorithms described in (15). For each pair of source and destination nodes, the reconstruction starts at a source node and traces the route taken by the MPI message according to the rules of the routing algorithm. The paths are recorded for all source-destination pairs and are used to calculate load and number of hops for MPI_Alltoallv.

6.4.4 Experimental Results

The following values are measured to evaluate the efficacy of the framework and different approaches: throughput, total number of paths, maximum and average values for number of paths per job, number of paths per link, and total amount of data per link for various communication patterns, source destination pairing, distance and sizes of sources and destinations, partition sizes, message sizes, and chunk size. In the next subsections, the dissertation presents the results and a detailed study of system behavior and network performance for the benchmarks.

6.4.4.1 Overall Throughput Improvement

In order to demontrate the efficacy of the framework, 91 experiments were performed on 512, 1024, 2048 and 4096 nodes of Mira. The overlall performance for 2048-node partition is shown in Figure 7.

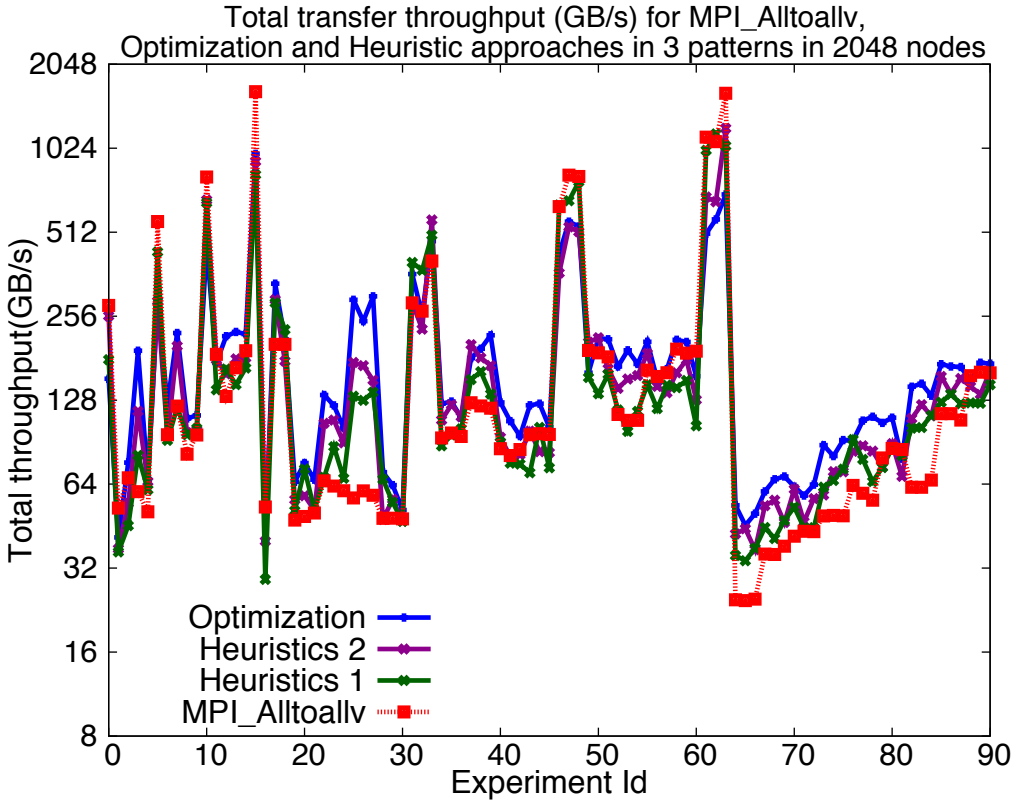


Figure 7: Total throughput from 91 cases for 3 patterns in 2048-node partition.

Figure 7 shows the overall performance of Optimization, Heuristics and MPI_Alltoallv from 91 experiments on 2048-node partition. For this, Optimization, Heristic 1 and Heuristic 2 showed 45.38%, 20.28% and 18.32% improvement over MPI_Alltoallv on average. The average improvement at other scales is shown in Table VI.

Approaches	Partition size (Number of nodes)			
	512	1024	2048	4096
Optimization	50.63	67.32	45.38	43.81
Heuristic 2	31.61	43.19	20.28	17.43
Heuristic 1	29.98	27.61	18.32	13.20

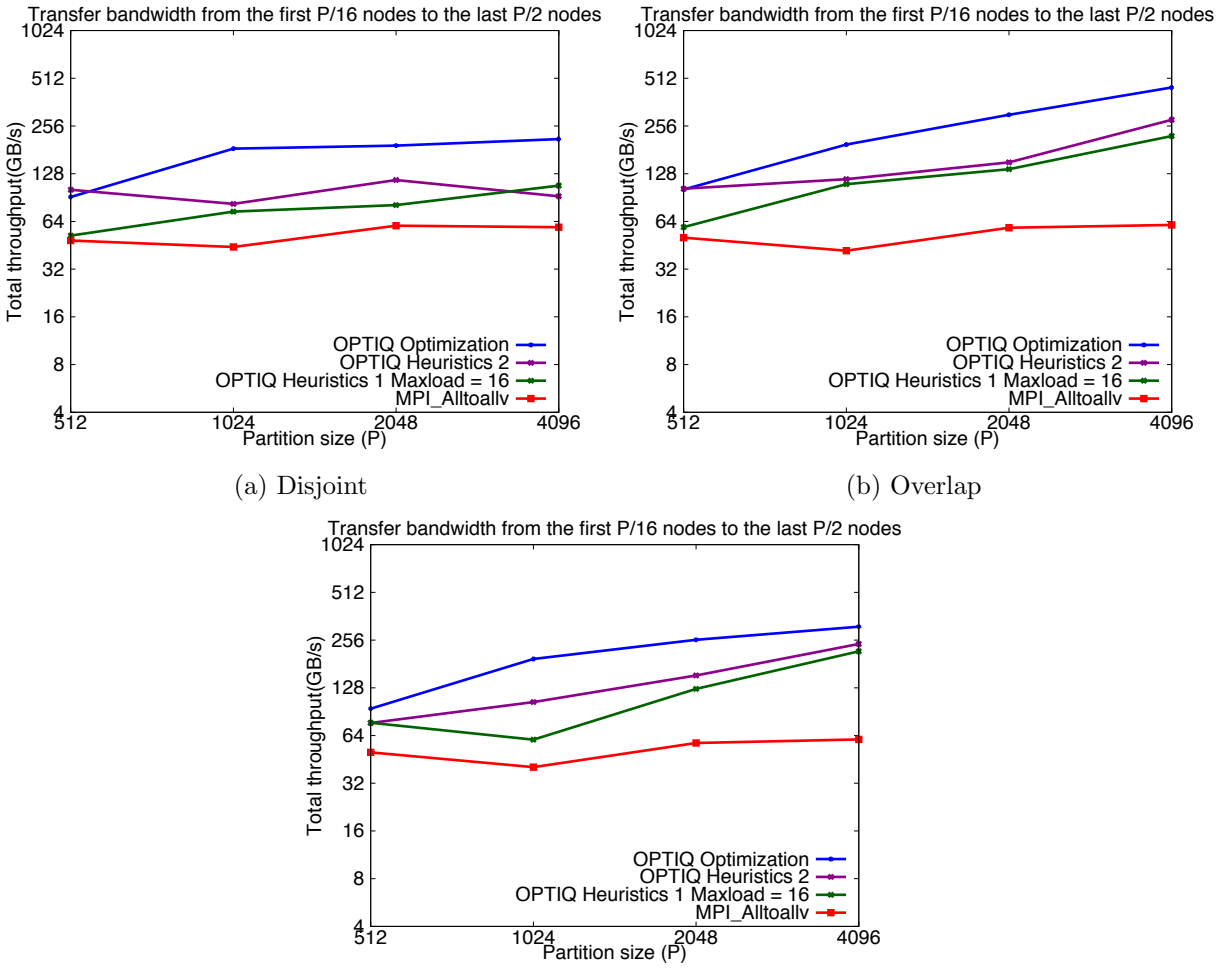
TABLE VI: Overall throughput improvement (%) of 3 approaches over MPI_Alltoallv in 91 experiments with different partition sizes.

6.4.4.2 Scaling Number of Nodes, Keeping Source/Destination Ratio Constant

In this experiment we vary the number of sources and destinations, together with the total number of nodes, while keeping a constant ratio of 1:8 between the number of source and destination nodes. We increase the total number of nodes P from 512 to 4096, The first P/16 nodes send data to the last P/2 nodes. Each source has 8 destinations e.g. node 0 sends data to nodes P/2, P/2+1, ... P/2+7. We present results for *disjoint*, *overlap* and *subset* communications using OPTIQ Optimization (OPT), OPTIQ Heuristic 1 (HEU 1), OPTIQ Heuristic 2 (HEU 2) and MPI_Alltoallv (MPI). We use 1 MPI/PAMI rank/node. The data size is 8 MB per source-destination pair.

Figure 8 shows the throughput for Optimization, Heuristic 1 and 2, and MPI. As shown in the figure, the Optimization approach has the highest throughput, followed by the Heuristic 2 approach then the Heuristic 1 approach and MPI_Alltoallv at the bottom. This is because Optimization and both heuristic approaches use multiple paths for data transfer. Additionally, Optimization globally balances the load for all source-destination pairs. Heuristic 2 usually has better performance than Heuristic 1 due to its capability of distributing data better on physical links.

Table VII shows the throughput, total number of paths, maximum and average number of paths per job, maximum and average number of paths per physical link and the total data per link for 1024 nodes. In all three cases, the greedy Heuristic 2 is able to find the highest number of paths for the entire data flow, following by Heuristic 1, Optimization, and MPI. MPI has only 1 path for 1 pair of data movement. A similar trend is also observed with the maximum and average number of paths per job. However, the paths found by both Heuristic approaches are based on local optimization of load on the physical links, as explained in Section 5.3.1 and Section 5.3.2. Therefore, there are a higher number of paths per physical link, and a higher amount of data per physical link in both Heuristics approaches as compared to the Optimization approach. Optimization has a lower number of paths per link because Optimization globally load balances the data transfer on the physical links. The most important value that affects the data movement throughput is the maximum amount of data per physical link. MPI has the highest maximum amount of data per link, leading to the most consgested links. Heuristic



(a) Disjoint

(b) Overlap

(c) Subset

Figure 8: Varying the number of sources and destinations and total number of nodes while keeping the ratio constant (1:8).

1 has the second highest value, next is Heuristic 2. Optimization has the lowest maximum amount of data per link, thus lowest congestion, contributing to its highest throughput.

Pattern	Type	BW (GB/s)	Num. of Paths			Num of paths Per Link		Max amt. of data / link (MB)
			Total Paths	Per Job				
				Max	Avg	Max	Avg	
Disjoint	OPT	188.62	1169	6	2.28	11	2.53	18.28
	HEU 2	84.50	3723	25	7.27	11	6.00	24.31
	HEU 1	74.88	3146	23	6.14	16	4.94	63.04
	MPI	45.18	512	1	1.00	16	3.07	134.21
Overlap	OPT	200.03	1303	6	2.54	13	2.74	16.97
	HEU 2	121.05	5991	33	11.70	10	5.49	22.21
	HEU 1	113.17	3273	26	6.39	16	5.17	38.66
	MPI	42.84	512	1	1.00	16	3.38	134.21
Subset	OPT	199.20	1269	6	2.48	11	2.79	17.10
	HEU 2	106.56	9350	38	18.26	10	5.70	21.75
	HEU 1	61.71	3238	26	6.32	16	5.28	45.08
	MPI	41.37	512	1	1.00	16	3.52	134.21

TABLE VII: Throughput, total number of paths, number of paths per job, maximum and average values number of paths per link and max amount of data per link for 3 patterns in 1024 nodes experiments.

Figure 9 shows the distribution of data per link for OPT, HEU 2, HEU 1 and MPI. The optimization approach considers the global load on the network and hence data is distributed among the paths in a more balanced way such that the physical links are load-balanced. Thus, OPT has the lowest maximum amount of data per physical link as shown in Figure 9. MPI_Alltoallv has the lowest number of paths. With one path per pair of communication, the entire data is

transferred using the single path, without utilizing the other idle links. MPI has the highest amount of data per physical link as can be observed in Figure 9. Thus MPI has the lowest throughput. Heuristic 1 and 2 are in the middle of the range with Heuristic 2 having better data per link distribution. This leads to higher throughput of Heuristic 2 in comparison to Heuristic 1.

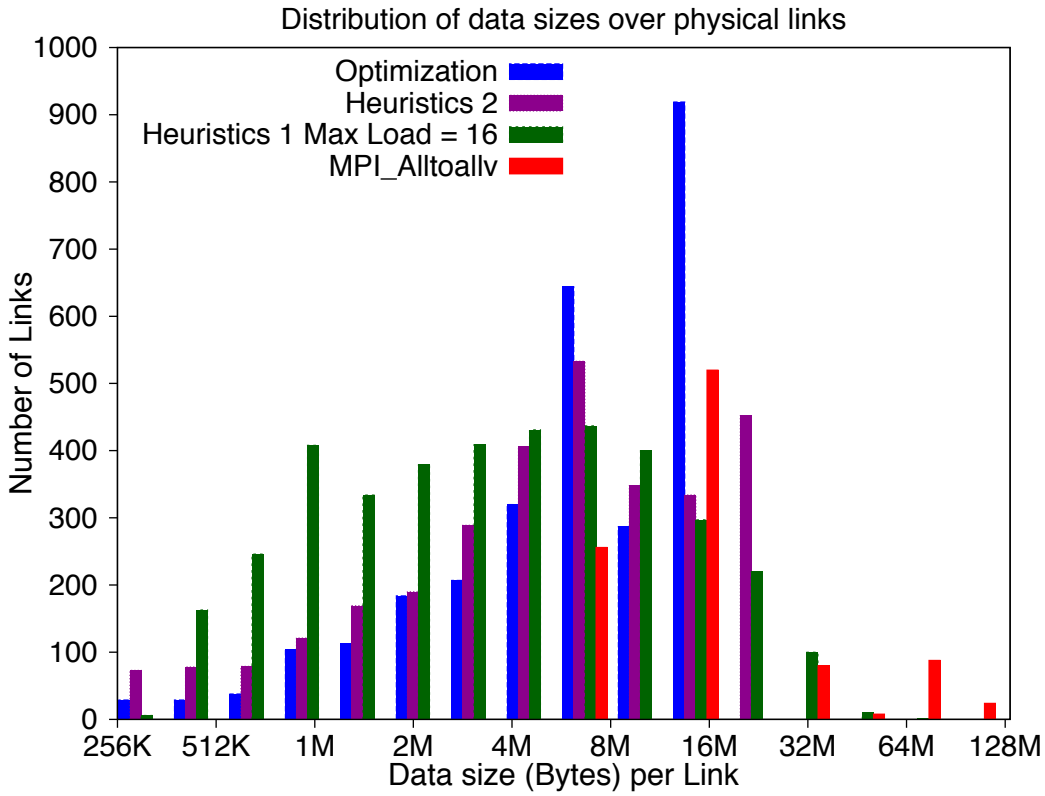


Figure 9: Distribution of total amount of data per link for Disjoint pattern in 1024-node partition.

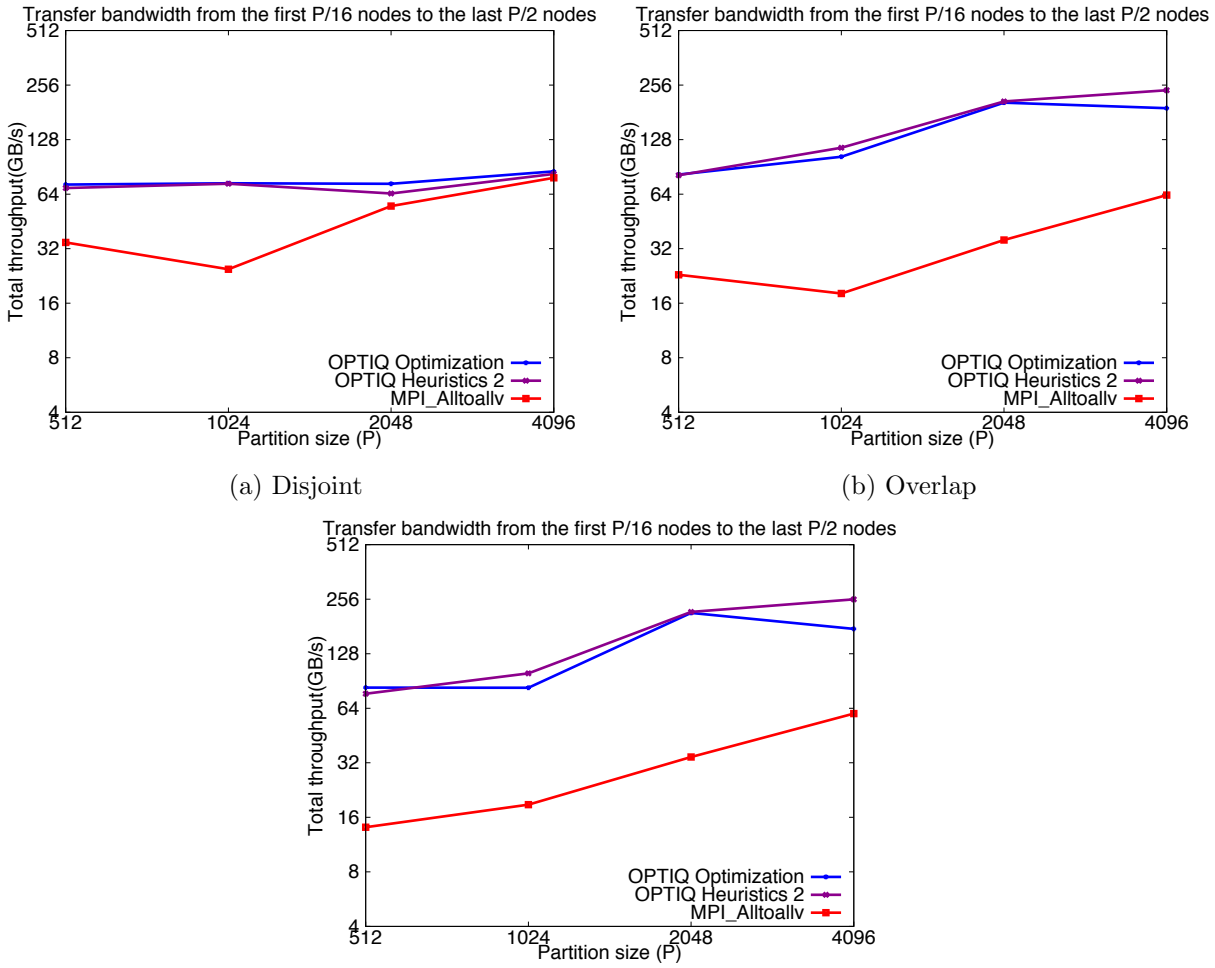
6.4.4.3 Varying Messages Sizes

In this experiment, we varied the number of source nodes, number of destination nodes, and total number of nodes (partition sizes) but kept the ratio between the source nodes and the destination nodes constant. With P as size of the partition, we choose the first P/16 nodes as the source nodes, and N/2 last nodes as the destination nodes. Each node in the set of source nodes communicates with 8 nodes in the set of destination nodes. The pairing is aligned i.e. node 0 communicates with nodes (P/2, ..., P/2/7). We randomly chose the data size for each pair of communication from 64 KB up to 8 MB of data. We experimented with three communication patterns: Disjoint, Overlap, Subset and three approaches: Optimization, Heuristic, and MPI_Alltoallv. We used only one MPI/PAMI rank per node. The total number of nodes P varied from 512 up to 4096 nodes. The communication throughputs are shown in Figure 10.

As shown in the Figure 10, in all three communication patterns, the Optimization approach and Heuristic 2 approach have similar performance and both are significantly higher than MPI_Alltoallv in most of the experiments. In the Disjoint pattern, the performance gap is close at 4096 nodes. With the other two patterns the gap is quite constant showing the scalability of our approaches.

6.4.4.4 Varying Sources-Destinations Distance

Figure 11 shows the throughput of OPTIQ Optimization (OPT), OPTIQ Heuristic 1 (HEU 1), OPTIQ Heuristic 2 (HEU 2), and MPI_Alltoallv (MPI) for disjoint, overlap and subset configurations. The experiment was performed in a 2048-node partition with 256 source nodes



(a) Disjoint

(b) Overlap

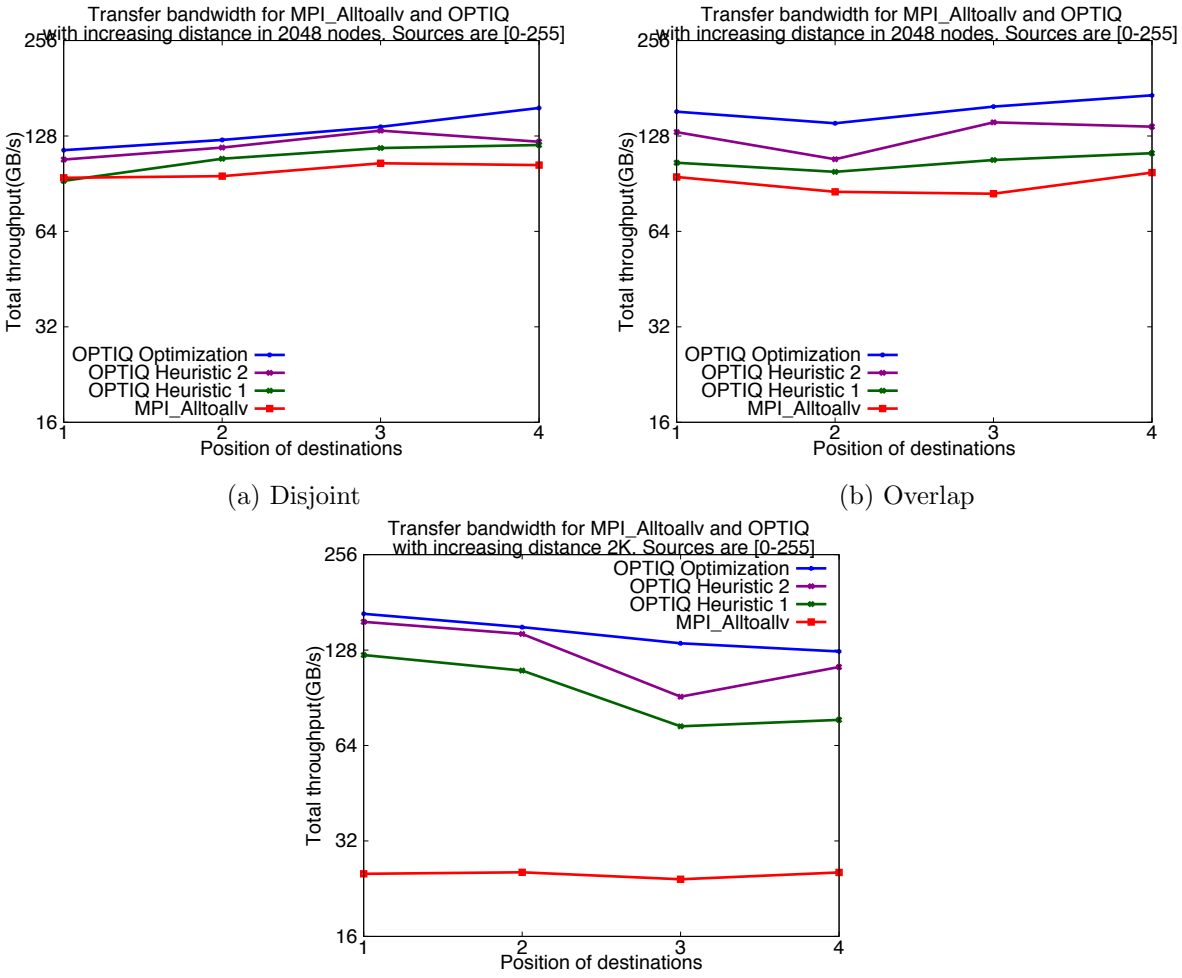
(c) Subset

Figure 10: Varying the number of sources and destinations and total number of nodes with constant ratio

communicating to 512 destination nodes. The set of destination nodes was chosen such that the average distance between the source and destination nodes increases. The distance between two nodes implies the number of hops between them. The x-axes in 11a, 11b and 11c represent the different destination node positions. For example, in 11a, the sources are from 0–255 and destination position 1 refers to destination node set from node 256–767, position 2 refers to node 512–1023, position 3 refers to node 1024–1535, and destination position 4 refers to 1536–2047. The y-axes shows the throughput. We observe that the throughput of OPT and HEU 1 and 2 increase with increasing distance in case of disjoint. This is because OPT and both HEU are able to find more paths with increasing distance without increasing the maximum load on the physical links. Both the Optimization and Heuristics approaches outperform MPI_Alltoallv which uses a single path for communication.

Positions	1					2				
	Distance		Number of Paths			Distance		Number of Paths		
Patterns	Max	Avg	OPT	HEU 1	HEU 2	Max	Avg	OPT	HEU 1	HEU 2
Disjoint	14	7.50	1105	2822	4751	14	7.50	1372	2887	4809
Overlap	13	7.25	2085	6460	5695	14	7.69	2152	3671	6782
Subset	20	8.56	1840	3422	7276	21	8.56	1639	3364	8203
Positions	3					5				
	Distance		Number of Paths			Distance		Number of Paths		
Patterns	Max	Avg	OPT	HEU 1	HEU 2	Max	Avg	OPT	HEU 1	HEU 2
Disjoint	15	8.50	1547	3668	5429	15	8.50	1672	3834	5170
Overlap	15	7.88	2337	6548	5140	18	9.59	2399	7010	7338
Subset	22	9.06	1594	3119	7310	23	9.06	1477	3087	5874

TABLE VIII: Maximum (Max) and average (Avg) distance between sources and destinations and number of paths for OPT and HEU for *disjoint*, *overlap* and *subset* on 2048 Mira nodes.



(a) Disjoint

(b) Overlap

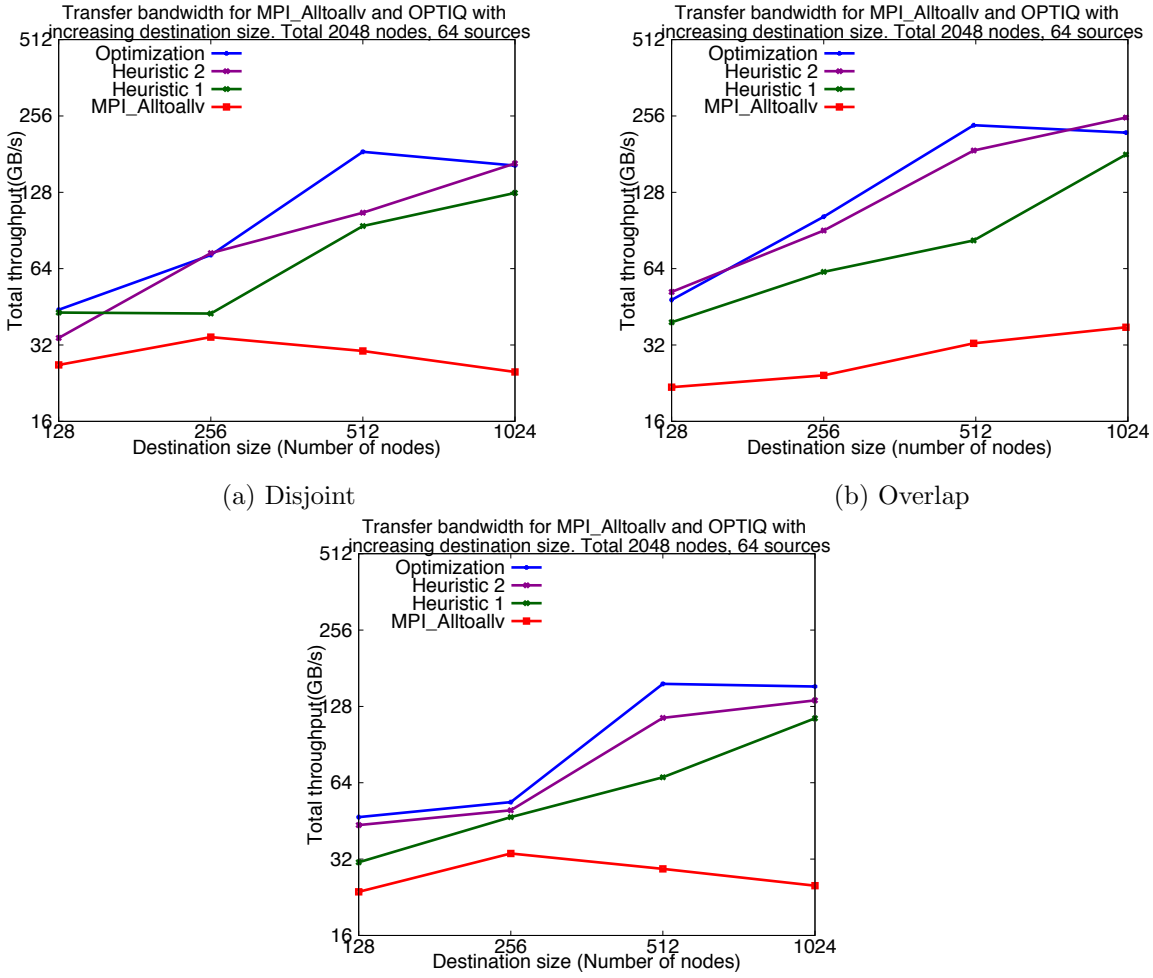
(c) Subset

Figure 11: Total data movement throughput with increasing distance between sources and destinations.

Table VIII shows the corresponding maximum and average distances between source and destination nodes, and the number of paths for OPT and HEU 1 and 2 for the configurations in Figure 11. The number of paths for MPI is 512. In general, the performance of OPT improves with higher number of paths as seen in the first row for disjoint. It can also be seen that the number of paths decreases for OPT in the case of subset which leads to a decrease in throughput. HEU 1 and 2 have more paths than OPT but due to imbalanced data distribution, the throughputs of both heuristic approaches are lower than OPT.

6.4.4.5 Varying Sources-Destinations Ratio

In this experiment we use a partition of 2048 nodes. We keep the number of source nodes constant (64 nodes) and increase the number of destination nodes from 128 to 256, 512 and 1024 nodes. Each source node communicates with k destination nodes where k = 2, 4, 8 and 16 respectively. Node x communicates with nodes $k \cdot x$, $k \cdot x + 1$, ..., $k \cdot x + (k - 1)$. There is 1 MPI/PAMI rank per node. Each pair of communication involves 8 MB of data transfer. The performance of OPT, HEU 2, HEU 1 and MPI for disjoint, overlap, and subset patterns is shown in Figure 12. With an increase in the number of destination nodes, OPT and both HEU yield better performance than MPI_Alltoallv. The throughput of OPT increases for destination sizes of 256 and 512 but slightly reduces at 1024 nodes, whereas the throughput of both HEU increase as the destination size increases. This is because OPT tries to globally balance load while distributing data among all paths, whereas both HEU ensure the load limit per path but distribute data for each communication pair, oblivious of the global load. With a higher number of destination nodes, the number of paths with overlapping links increases.



(a) Disjoint

(b) Overlap

(c) Subset

Figure 12: Total data movement throughput with increasing number of destination nodes.

Another reason for OPT's drop in performance is because we do not consider the underlying synchronization overhead in the data transfer optimization formulation.

6.4.4.6 Random Sources-Destinations Pairing

In contrast to the previous subsections, in this experiment, we randomized the pairing between sources and destinations. We did experiments for all three patterns on 1024 nodes, with 1 MPI/PAMI rank per node, 8 MB of data per communication. We used source to destination ratio of 1:8, i.e. first 64 nodes (nodes 0–63) communicate with last 512 nodes (nodes 512–1024). However, we randomly selected the pairs of sources and destinations. The results are presented in Table IX.

	Config 1				Config 2				Config 3			
Patterns	OPT	HEU 2	HEU 1	MPI	OPT	HEU 2	HEU 1	MPI	OPT	HEU 2	HEU 1	MPI
Disjoint	198	167	74	50	171	214	124	54	203	204	135	55
Overlap	206	207	95	52	205	221	133	60	206	227	134	64
Subset	221	210	118	55	185	216	135	60	219	203	132	52
	Config 4				Config 5				Average			
Patterns	OPT	HEU 2	HEU 1	MPI	OPT	HEU 2	HEU 1	MPI	OPT	HEU 2	HEU 1	MPI
Disjoint	198	233	134	62	194	224	115	59	193.21	206.81	116.92	56.00
Overlap	214	194	125	57	207	203	133	59	208.29	210.88	124.62	58.40
Subset	186	211	123	51	222	197	112	57	207.10	207.92	124.58	55.00

TABLE IX: Throughput (GB/s) for Optimization (OPT), Heuristics (HEU 1 and 2) and MPI_Alltoallv (MPI) for 5 different random pairings between sources and destinations in 1024-node partition.

Optimization and both Heuristic approaches outperform MPI in all cases due to better data distribution and load-balance. OPT and HEU 1 & 2 result in 50% and 36% better performance respectively in the case of disjoint.

6.4.4.7 Efficacy of Chunk Size

To transfer a message from a source to a destination through intermediate nodes we split the message into smaller chunks and keep sending the chunks into the network. This is to reduce the waiting time at the intermediate nodes, and thus reduce total transfer time. We carried out an experiment to show optimal chunk sizes for different message sizes. In this experiment we varied the message sizes from 8 KB up to 8 MB. The chunk sizes also varied from 4KB up to 1MB. The experiment was carried out in a 512-node partition using subset pattern in which the first 32 nodes send data to the last 256 nodes. The results are shown in Figure 13.

As shown in the Figure 13, for messages with sizes less than 16 KB, we should transfer the entire message. With message size 32 KB we can use a chunk-size of 16 KB. With message sizes 64 KB or 128 KB we can use a 32 KB chunk-size. With larger message sizes we can use a 64 KB chunk size. Similar trends are found in the disjoint and overlap patterns. For the experiments in this dissertation we use a 64 KB chunk size.

6.4.4.8 Efficacy of Message Size

In this experiment we show the effect of varying message sizes. The experiment is carried out in a 512-node partition with 1 MPI/PAMI rank per node, 8 MB message size for all three patterns. The results for OPT and MPI are shown in Figure 14.

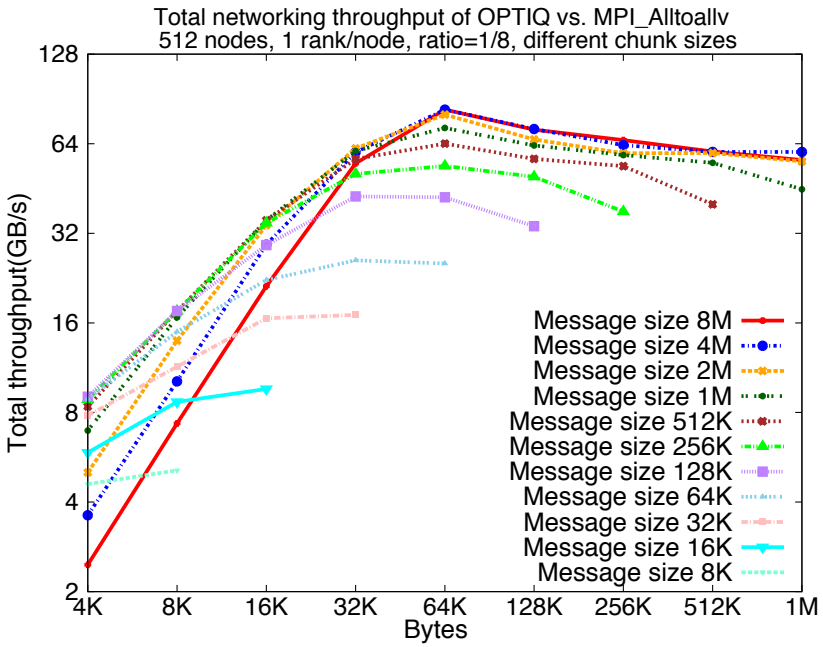


Figure 13: Chunk sizes and their performance in 512-node partition, subset pattern.

The throughput of transferring small messages in OPTIQ is lower than the default MPI_Alltoallv due to overhead in data transfer by OPTIQ. MPI_Alltoallv has better performance than OPTIQ when the data size is less than 512 KB. When the message size is greater than 512 KB, OPTIQ has better performance. The lower performance in OPT at smaller message sizes is due to overhead caused by additional messages, *send* and *forward* queueing, chunking of messages, and time to copy and inject messages at the intermediate nodes.

6.4.4.9 Efficacy of *maxload* Value on Heuristic 1 Approach

For the heuristic approach, we use k shortest paths and a *maxload* value to select the number of paths used for data transfer, as described in 5.3.1. Depending on the *maxload* value, the

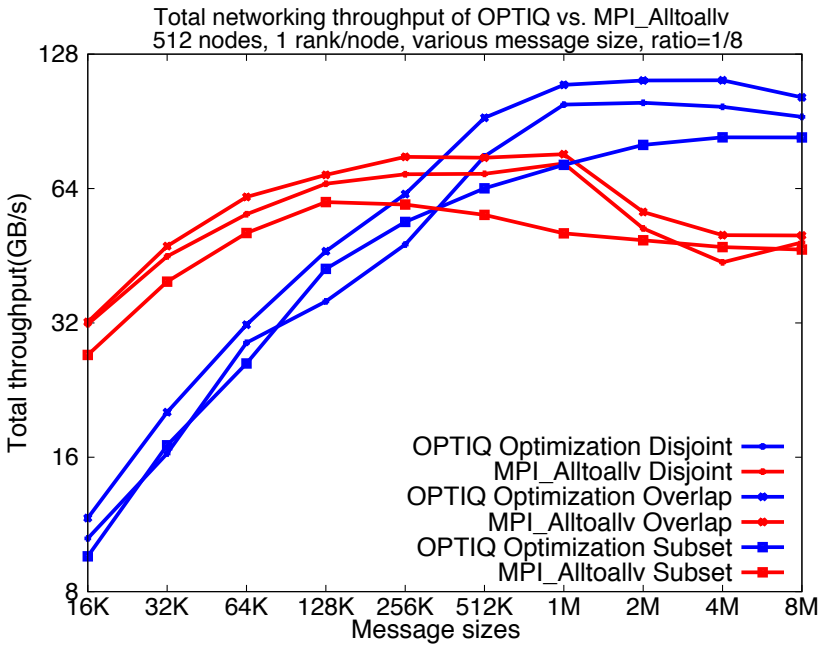


Figure 14: Total throughput with different message sizes from 16 KB – 8 MB in disjoint, overlap and subset for OPT and MPI.

heuristic may select a different set of paths, which can affect performance. In this experiment, we show the effect of choosing the *maxload* value and time to select paths based on the *maxload* value. The experiment was carried out in a 1024-node partition, with 1 MPI/PAMI rank per node, 8 MB message size, source to destination ratio of 1:8 for all 3 patterns. The results for *maxload* values of 1, 2, 4, 8, 16, and 32 are shown in Table X. The first column shows the pattern type, the second column shows the MPI performance and the remaining columns show Heuristic 1 performance for various *maxload*.

Patterns	MPI	Maxload					
		1	2	4	8	16	32
Disjoint	45	31	32	32	63	75	78
Overlap	42	66	66	66	125	112	89
Subset	74	69	70	69	114	110	96

TABLE X: Throughput (GB/s) with different *maxload* values for Heuristic 1 approach.

The performance of Heuristic 1 with *maxload* value of 1, 2 or 4 is similar, and lower than MPI_Alltoallv in the disjoint and subset patterns. This is because with lower *maxload* values, the heuristic is not able to find enough paths to transfer data, leading to fewer number of physical links being used, thus higher load on those physical links. When the *maxload* value is set to 32, performance starts to degrade because the heuristic finds too many paths, leading to many paths sharing a physical link, thus leading to higher load on those physical links. The best performance is achieved with *maxload* value of 8 and 16 because of better load distribution on the physical links. For the experiments in this paper we set *maxload* value to 16.

When we increase the *maxload* value, it also takes more time to select paths from the k shortest paths. Table XIII shows the time for different *maxload* values in different patterns.

Pattern	Time for Different Max Load (s)					
	1	2	4	8	16	32
Disjoint	1.958	1.961	1.917	1.956	2.002	2.164
Overlap	1.923	1.890	1.801	1.929	1.993	2.082
Subset	1.907	1.870	1.891	1.955	2.024	2.223

TABLE XI: Search time with different max load in 1024 nodes partition.

The search time is short and thus can be amortized over time when a pattern is used repeatedly.

6.4.4.10 Efficacy of Number of Shortest Paths Feeding into Solvers

For the Optimization approach we need to input k shortest paths for the solvers to search for an assignment of flow values on the k paths. In this experiment we show the relationship between the number of paths input to the model, the corresponding data transfer throughput, and the elapsed time for the AMPL model and solvers. We carried out the experiment in a 2048-node partition for all three patterns with source to destination ratio of 1:8, where 128 nodes communicate with 1024 nodes. We used 1 MPI/PAMI rank per node and 8 MB per communication. We varied the number of paths fed into the solvers from 4 to 16, 32 and 50. The performance is shown in Table XII.

Patterns	MPI	Number of paths			
		4	16	32	50
Disjoint	61	29	84	104	197
Overlap	59	82	192	224	308
Subset	111	99	163	168	172

TABLE XII: Throughput (GB/s) with different number of paths input to the solvers.

As we increase the number of paths, the performance improves. This is because with more paths the solvers have a larger search space, and thus can produce more optimal results to be

used for data transfer. However, with increasing number of paths, we also increase the time for AMPL model to prepare and solvers to search for flow values for paths as shown in Table XIII. AMPL time is the total amount of time that AMPL needs to check conditions, input data, to load input data and to generate an instant of a model. The solving time is the actual amount of time used by an employed solver to solve a problem. The AMPL environment provides the amounts of time via 2 built-in timing parameters: $_{a}mpl_{e}lapsed_{t}ime$ and $_{t}otal_{s}olve_{e}lapsed_{t}ime$.

Pattern	AMPL time (s)				Solve time (s)			
	4	16	32	50	4	16	32	50
Disjoint	13.9	187.7	123.0	224.0	0.06	6.6	4.4	84.0
Overlap	13.6	51.9	134.6	198.7	0.09	16.6	179.4	530.3
Subset	14.4	50.6	134.9	217.3	0.85	111.3	173.2	939.6

TABLE XIII: AMPL and solving time.

6.4.4.11 Efficacy of Solvers

In this section, the dissertation presents the efficacy of solvers on solving time as well as the quality of results via the througput of data movement with outputs from different solvers. The framework used two solvers in the experiments: Sparse Nonlinear Optimization (SNOPT) and CPLEX. SNOPT is a optimization solver for large-scale nonlinear problems. CPLEX is optimization solver for very large linear programming problems. The problem described in 5.4.1 is a linear problem, so it is expected to be solved faster with CPLEX than solved with SNOPT.

There were 91 experiments carried out in 2048 nodes for three data patterns using the two solvers. For each experiment, there were 50 shortest paths per pair of communication fed into the solvers. The total AMPL time, total solving time, total of AMPL time, and solving time and throughput were measured and reported in Figure 15.

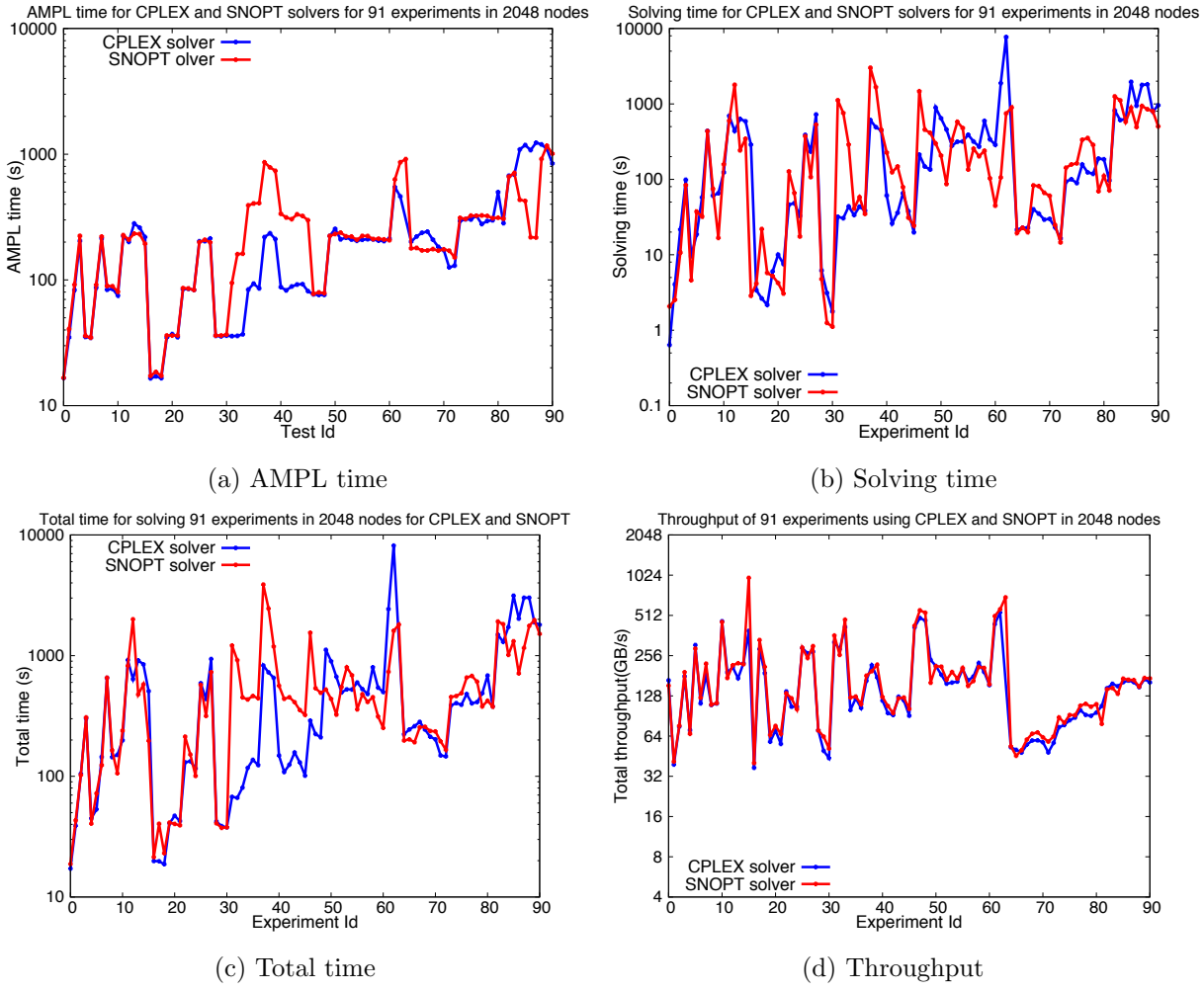
Table XIV presents the time and performance for 91 experiments.

Solvers	AMPL time	Solving time	Total time
CPLEX	22332.8	33973.2	56306.0
SNOPT	24578.2	29728.5	54306.6

TABLE XIV: Time and throughput comparision between 2 solvers CPLEX and SNOPT over 91 cases in 3 patterns.

In terms of total time, in 91 experiments, the total AMPL time for CPLEX is shorter than SNOPT, but the total solving time is surprisingly higher than SNOPT. This leads to the total amount of time for the AMPL model and solvers of CPLEX being higher than SNOPT's total time.

In term of performance, CPLEX and SNOPT demonstrated comparable performance. The difference is insignificant i.e. SNOPT has 7% higher performance on average over 91 experiments.



(a) AMPL time

(b) Solving time

(c) Total time

(d) Throughput

Figure 15: AMPL time, Solving time, Total time and Throughput for CPLEX and SNOPT solvers of 91 cases in 3 patterns in 2048-node partition

6.4.4.12 Paths Searching Time

As the dissertation presents in the previous sections, the performance of Optimization approach is highest, followed by the performance of Heuristic 2 and Heuristic 1. However the performance gained comes with a trade-off of path searching time. Figure 16 shows the paths searching time for three approaches for 91 experiments of three patterns.

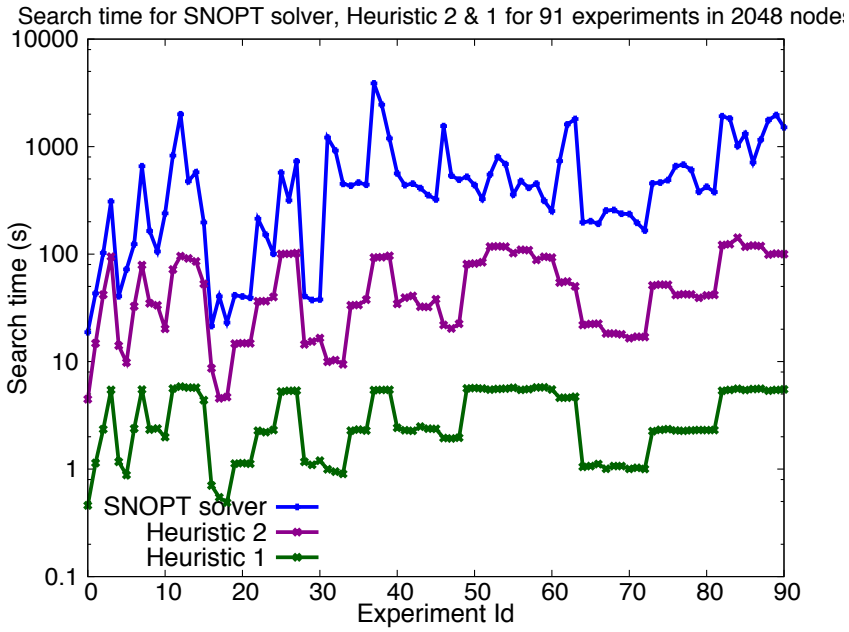


Figure 16: Paths searching time for Optimization approach used SNOPT solver, Heuristic 2, Heuristic 1 with $maxload = 16$

As shown in Figure 16, the search time of Heuristic 1 is lowest, then Heuristic 2. The Optimization approach has the highest search time. In general, Heuristic 1 can finish the search within a few seconds, sometimes less than 1 second. The Heuristic 2 can complete the search in the time from 8 seconds up to 100 seconds. With the Optimization approach, the search time goes up to a few thousands seconds. Thus, it is recommended to use Optimization approach offline for most of cases. Heuristic 1 and 2, on the other hand, can be used on-the-fly for instant path searching.

In the next section, we demonstrate the efficacy of our approaches through an experiment with communication patterns and data from two real applications: Community Earth System Model (CESM) and Hardware/Hybrid Accelerated Cosmology Code (HACC).

6.5 Evaluation with Applications

In these applications, the pairing between sources and destination are random-like. Multiple ranks, different message sizes. Each ranks/nodes can talk to different number of nodes.

6.5.1 Community Earth System Model (CESM)

6.5.1.1 Introduction

Community Earth System Model (CESM) is a coupled global climate model simulating the earth system consisting ice, land, ocean, atmospheric and other components (33). In order to provide flexibility in developing, models in CESM do not communicate directly with each other but via a coupler. The Figure 17 shows the communication between 4 models: Atmosphere (ATM), Ice (ICE), Land (LND) and Ocean (OCN) with the Coupler (CPL).

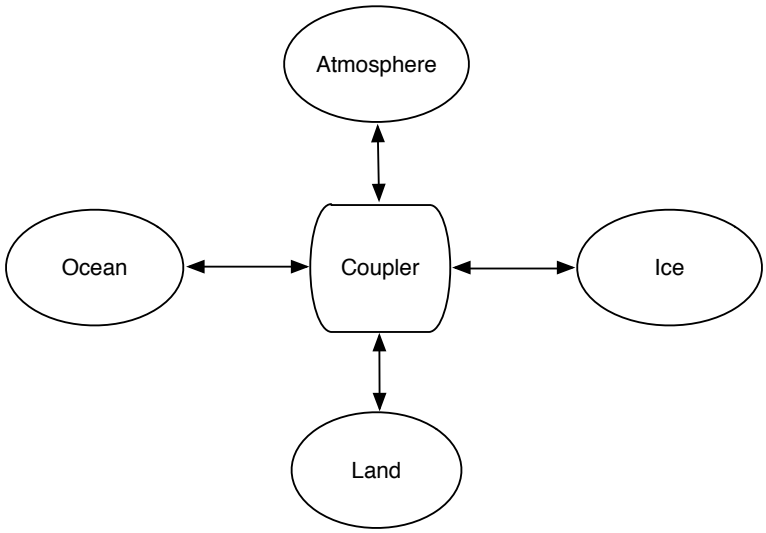


Figure 17: Model communicate with each other via a coupler

In this dissertation, we demonstrate the efficacy of our framework via communication between the models and the coupler extracted from a real run of CESM.

6.5.1.2 Experiments and Results

Our experiment was carried on a partition of 512 nodes with 4 MPI/PAMI ranks per node. The positions of each component, and the number of pairs of communication between ranks and between nodes are shown in Table XV

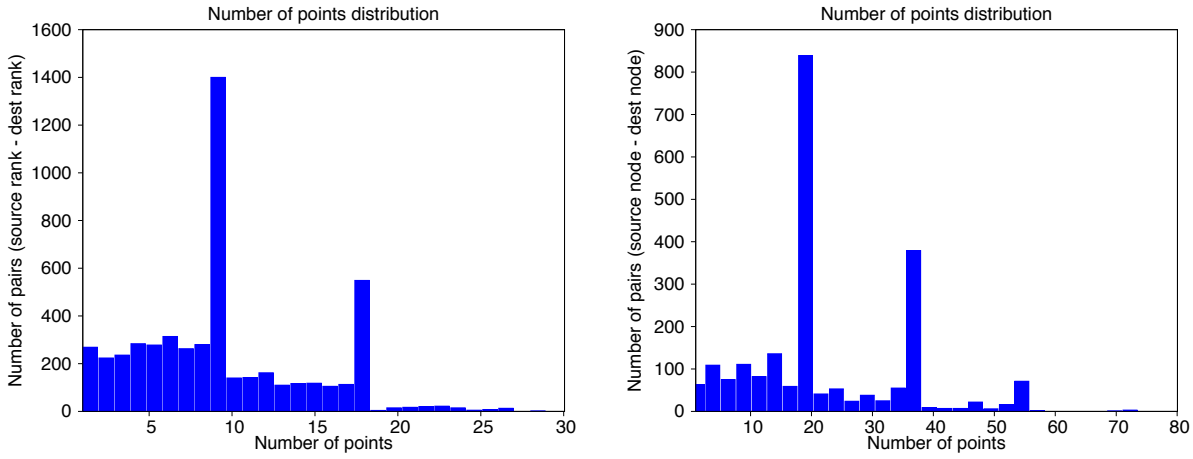
Table XV shows the ranges of ranks (start-end) that host the models and the coupler. It also shows the number of pairs of communication between the models and the coupler. The number of pairs of communication are counted by MPI/PAMI rank or by node Id. If pairs of communication with source ranks in the same source node and destinations in the same

Model	Location	Num. of pairs of communication with Coupler	
		Between Ranks	Between Nodes
ATM	0 - 1791	5227	2233
LND	0 - 515	10390	2911
ICE	516 - 1791	3018	765
OCN	1792 - 2047	2001	502
CPL	0 - 1791		

TABLE XV: Locations and number of pairs of communication between models in CESM

destination node then we count them as one pair of communication between the source and the destination nodes with the data to transfer as the total data of all the pairs. As the Table XV shows, the number of pairs counting by node Id is much lower than the number of pairs counting by MPI/PAMI rank. This shows that the MPI/PAMI ranks in the same source node tend to communicate with MPI/PAMI ranks the same destination node. We gather data of pairs with the same source and destination nodes and let only one pair transfer data. This helps to increase the data size per transfer and to reduce the number of pairs of communication that we need to compute paths, thus speeding up path searching. Communication between MPI/PAMI within a node is carried out using OpenMP.

In the given communication patterns of CESM, each rank talks to 0 to 4 other ranks. The message sizes also vary from 1 data point to 26 data points. The distribution of data points per pair of ranks is shown in 18a. Similar distribution of the number of data points per pair of nodes is shown in 18b.



(a) By Rank Id (b) By Node Id

Figure 18: Points distribution over pairs of source-rank/dest-rank and pairs of source-node/dest-node

Figure 18 shows that by gathering data of communications with the same pair of source node and destination node, the framework can increase the data size significantly. Most of the pairs by rank have less than 20 data points per pair, while most of the pairs by node have more than 20 data points per pair. In this experiment, each point of data is converted into 32 KB of data for communication.

We carried out the experiment's communication between the Coupler with Atmosphere, Land, and Ocean. The result of the experiment is shown in Table XVI for three pairs of communication between CPL-ATM, CPL-LND and CPL-OCN. The throughputs are shown for different approaches: Optmization (OPT), Heurisic (HEU) and MPI_Alltoallv (MPI). We also

show the total number of paths and the maximum and average number of paths per job for each approach.

Coupling	Type	BW (GB/s)	Num. of Paths		
			Total Paths	Per Job	
				Max	Avg
CPL-ATM	OPT	352.24	3987	5	1.37
	HEU 2	350.62	13789	15	6.18
	MPI	241.04	2911	1	1.00
CPL-LND	OPT	343.5	4004	7	1.38
	HEU 2	332.40	15107	14	5.19
	MPI	278.90	2911	1	1.00
CPL-OCN	OPT	135.16	987	5	1.97
	HEU 2	136.06	5924	35	11.80
	MPI	104.44	502	1	1.00

TABLE XVI: Throughput, total num of paths, number of paths per job for 3 couplings in 512 nodes (4 ranks/node) experiments.

As shown in the Table XVI, the Optimization approach usually has the highest throughput, except for the CPL-OCN case, where Optimization approach and Heuristic 2 approach have similar performance. The Heuristic 2 approach usually has the second highest throughput. MPI has the lowest throughput. This is because MPI uses only one path to transfer data between two nodes. This leads to sharing links between pairs of communication coming from the same source node to the same destination nodes. Also the routing policy does not consider idle links and the load on neighbors to balance the load. Both factors lead to low performance in MPI. The

heuristic approach employs the largest number of paths, but due to the data assignment is not as optimal as the Optimization approach, it still has lower performance. With the performance improvement of 30% we can see that by using multipath we can rebalance network load and improve communication throughput.

6.5.2 Hardware/Hybrid Accelerated Cosmology Code (HACC)

6.5.3 HACC I/O Application Benchmark

HACC (Hardware/Hybrid Accelerated Cosmology Code) (1) is a large-scale cosmology code suite that simulates the evolution of the universe through the first 13 billion years after the Big Bang. The application simulates trillions of particles, their movement, collison, as well as interactions while forming structures that transform into galaxies. During runtime, HACC writes data periodically to the storage system both for checkpoints and for I/O of the in situ analysis performed at simulation time.

In this benchmark, we use HACC I/O data to evaluate the performance of data aggregation of the system for HACC. The data was generated before in experiments done on a real simulation. In the simuation, there were 8 MPI ranks per node. The data that we used is around 700 KB to 800 KB per rank or around 6 MB per node. In Mira, I/O data is moved from compute nodes to bridge nodes before moving to I/O nodes. There are two bridge nodes per 128 compute nodes. Each compute node has one default bridge node. In this experiment, we aggregate data from compute nodes to their default I/O nodes by using OPTIQ Heuristic 2 and MPI_Alltoallv.

We evaluate the data movement performance from compute nodes to bridge nodes. We compare the aggregation throughput of OPTIQ Heuristic 2 to default MPI_Alltoallv in aggregating the data to the bridge nodes of Mira. In this experiment, we scale our experiments from 2,048 up to 32,768 compute cores to aggregate data. We aggregate data from 8 ranks per node to a single rank and let the single rank to aggregate data to bridge nodes. We collect bandwidth information and report the average of 10 runs. Figure 19 depicts the achievable performance of aggregating the data to bridge nodes on Mira.

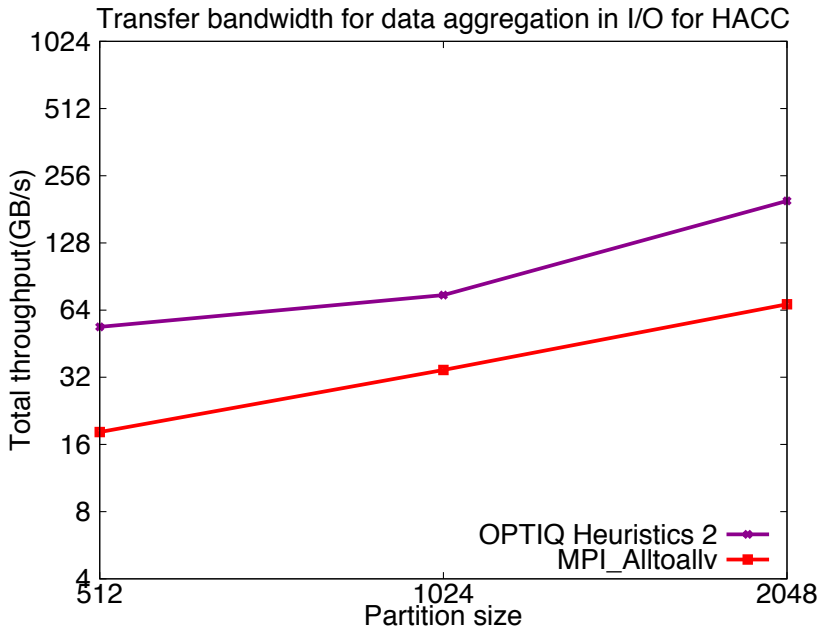


Figure 19: Aggregation achievable throughput for in situ analysis of HACC I/O

From the figure we observe an overall improvement of 60% up to 3 times throughput improvement for data aggregation. OPTIQ framwork can improve performance for data movement due to its capability of searching multiple paths and assigning data appropriately to reduce congestion on physical network links.

CHAPTER 7

CONCLUSION

This chapter summarizes the contributions of the dissertation and proposes some future directions.

7.1 Thesis Contribution

7.1.1 Holistic Approach and Data Movement Optimization Framework

This dissertation is among the first to discuss a holistic approach to improve data movement performance for data-centric applications on supercomputers. Most of the previous research focuses on optimization at the system layer for generic communication patterns. Other research focuses on optimizing data movement for specific communication patterns on specific systems. Research has been done at the application level, but they do not include the system routing and topologies as ours does. The holistic approach presented in this dissertation takes into account the interconnection network topology and the system routing and communication patterns of the applications to improve data movement performance. The scale at which this dissertation is presented is also significant compared to other works.

The dissertation also presents a data movement optimization framework named OPTIQ that realizes this holistic approach. The framework is designed to provide necessary functionalities to allow applications use it to improve data movement. The research also provides a application programming interface (API) that allows the framework to read system information such

as topology, torus, and size, to compute neighbor nodes and routing information of a given partition.

7.1.2 Multi-path Data Movement

Other important contributions of the dissertation are a set of approaches including two greedy heuristic approaches and a model-based optimization approach. All the approaches leverage multiple paths to balance the load on the physical network links on supercomputers. While the two heuristic approaches provide a faster but local optimization solutions, the model-based approach requires longer solving time but outputs globally optimal solutions.

7.2 Future Work

There are several directions that one could take to extend this research from here.

7.2.1 Improving performance by investigating different solutions

In this disseration, the OPTIQ framework only generates one solution from solvers. The solvers, however can generate different solutions, i.e different sets of paths with different data assignments for the solutions. The different solutions may have different numbers of hops and intermediate nodes and thus can have different performance due to the number of copies that are needed. Investigating different paths from different solutions is one way of extending of this research.

7.2.2 Expanding the Work to Other Supercomputers

In this dissertation, the framework is designed and built to work on different supercomputers. The experiments demonstrated the efficacy of the framework on the Blue Gene/Q

supercomputer. The work can be extended to other system with different interconnection network topology such as Cray supercomputers like Edison, Hopper or Stampede, or commodity clusters to see how the design works. Such as study can also help to modify the design of the framework to make it work better on different computing systems.

7.2.3 Providing Quality of Service for Data Movement on Supercomputers

In several supercomputing systems, all flows share the same interconnect network infrastructure. Current systems route all data flows with the same priority i.e. first come first serve. The middleware layer also processes different data flows with the same priority at best effort. This does not provide an opportunity for application writers/scientists to optimize the data flows based on their understanding of the data movement, i.e. some data flows are for computation and thus are more urgent than data flows for I/O which can be temporarily stored at the burst buffer and the transfer can be delayed until the resource contention is lower. With programmable optical switches and software defined networking, it is possible to control the data flows at the switch level. Thus, it is possible to provide quality of service for flows of data on supercomputers. Controlling data flows for QoS purposes will bring to the framework the capability of assigning different priorities for different data flows. The framework can compute how many paths are available at runtime and reserve an appropriate portion of resources for data flows corresponding to their priorities. With support from the framework, application developers can assign priorities for data flows that can result in lowest time-to-solution for their applications.

7.2.4 Reducing Solving Time for Optimial Solutions

Currently in the experiments presented in this dissertation, it could take a few hours to solve an optimization at 4096-node partition scale. It would take much longer time at a larger scale for the same communication patterns. As future supercomputers are expected to have a much larger number of compute nodes, it is necessary to reduce the solving time to a reasonably acceptable level. One approach is to use graph partitioning. Recent research on graph partition is presented in (34). One potential approach is multi-level graph partitioning. In this approach, a graph is first contracted/coarsened in to a much smaller graph by matching/grouping a number of nodes into a single node repeatedly until we reach to a small enough graph. Solvers can be used to solve the problem for the contracted/coarsened graph problem. The next step is to uncontract/uncoarsen the contracted graph to return it to the original state. At this state we have the flow values assigned for groups of nodes, which were single nodes in the contracted graph. We can use local improvement on the subgraphs. This approach can be applied at multiple levels. In case we have multiple solutions for a contracted graph, we can execute the uncontracting/uncoarsening and local improvement in parallel.

APPENDIX

IEEE POLICIES ON REUSE LICENSE

Reprinted from the official policies related to the IEEE copyright procedures, given under the subsection 8.1.4.1 (last verified Feb 14, 2014)

8.1.4 IEEE Copyright Policy and Procedures (from the PSPB Operations Manual)

B. Ownership and rights of IEEE copyrighted material

3. Prior to publication by the IEEE, all authors or their employers shall transfer to the IEEE in writing any copyright they hold for their individual papers. Such transfer shall be a necessary requirement for publication, except for material in the public domain or which is reprinted from a copyrighted publication.

4. In return for the transfer of authors rights, the IEEE shall grant authors and their employers permission to make copies and otherwise reuse the material under terms approved by the Board of Directors.

CITED LITERATURE

1. Habib, S., Morozov, V., Finkel, H., Pope, A., Heitmann, K., Kumaran, K., Peterka, T., Insley, J., Daniel, D., Fasel, P., Frontiere, N., and Lukić, Z.: The universe at extreme scale: multi-petaflop sky simulation on the BG/Q. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, pages 4:1–4:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

2. Collins, W. D., Bitz, C. M., Blackmon, M. L., Bonan, G. B., Bretherton, C. S., Carton, J. A., Chang, P., Doney, S. C., Hack, J. J., Henderson, T. B., et al.: The Community Climate System Model version 3 (CCSM3). Journal of Climate, 19(11), 2006.

3. Leighton, T., Stein, C., Makedon, F., Tardos, E., Plotkin, S., and Tragoudas, S.: Fast Approximation Algorithms for Multicommodity Flow Problems. In Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing, STOC '91, pages 101–111, New York, NY, USA, 1991. ACM.

4. Racke, H.: Minimizing Congestion in General Networks. In Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science (FOCS), pages 43–52, 2002.

5. Azar, Y., Cohen, E., Fiat, A., Kaplan, H., and Racke, H.: Optimal Oblivious Routing in Polynomial Time. In Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing, STOC '03, pages 383–388, New York, NY, USA, 2003. ACM.

6. Applegate, D. and Cohen, E.: Making Routing Robust to Changing Traffic Demands: Algorithms and Evaluation. IEEE/ACM Trans. Netw., 14(6):1193–1206, December 2006.

7. Kinsy, M. A., Cho, M. H., Wen, T., Suh, E., van Dijk, M., and Devadas, S.: Application-aware Deadlock-free Oblivious Routing. SIGARCH Comput. Archit. News, 37(3):208–219, June 2009.

8. Rodriguez, G., Minkenberg, C., Beivide, R., Luijten, R. P., Labarta, J., and Valero, M.: Oblivious routing schemes in extended generalized Fat Tree networks. In CLUSTER, pages 1–8. IEEE, 2009.

9. Prisacari, B., Rodriguez, G., Minkenberg, C., and Hoefler, T.: Bandwidth-optimal All-to-all Exchanges in Fat Tree Networks. In Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS '13, pages 139–148, New York, NY, USA, 2013. ACM.

10. Prisacari, B., Rodriguez, G., Minkenberg, C., and Hoefler, T.: Fast Pattern-specific Routing for Fat Tree Networks. ACM Trans. Archit. Code Optim., 10(4):36:1–36:25, December 2013.

11. Agarwal, T., Sharma, A., and Kalé, L. V.: Topology-aware task mapping for reducing communication contention on large parallel machines. In Proceedings of the 20th International Conference on Parallel and Distributed Processing, IPDPS'06, pages 145–145, Washington, DC, USA, 2006. IEEE Computer Society.

12. Valiant, L. G. and Brebner, G. J.: Universal Schemes for Parallel Communication. In Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, STOC '81, pages 263–277, New York, NY, USA, 1981. ACM.

13. Singh, A., Dally, W. J., Gupta, A. K., and Towles, B.: GOAL: a load-balanced adaptive routing algorithm for torus networks. ACM SIGARCH Computer Architecture News, 31(2):194–205, 2003.

14. Pifarré, G. D., Gravano, L., Felperin, S. A., and Sanz, J. L. C.: Fully-adaptive Minimal Deadlock-free Packet Routing in Hypercubes, Meshes, and Other Networks. In Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '91, pages 278–290, New York, NY, USA, 1991. ACM.

15. Chen, D., Eisley, N., Heidelberger, P., Kumar, S., Mamidala, A., Petrini, F., Senger, R., Sugawara, Y., Walkup, R., Steinmacher-Burow, B., Choudhury, A., Sabharwal, Y., Singhal, S., and Parker, J. J.: 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, SC '12, pages 69:1–69:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

16. Garcia, M., Vallejo, E., Beivide, R., Odriozola, M., and Valero, M.: Efficient Routing Mechanisms for Dragonfly Networks. In Parallel Processing (ICPP), 2013 42nd International Conference on, pages 582–592. IEEE, 2013.

17. Yelick, K., Bonachea, D., Chen, W.-Y., Colella, P., Datta, K., Duell, J., Graham, S. L., Hargrove, P., Hilfinger, P., Husbands, P., Iancu, C., Kamil, A., Nishtala, R., Su, J.,

Welcome, M., and Wen, T.: Productivity and Performance Using Partitioned Global Address Space Languages. In Proceedings of the 2007 International Workshop on Parallel Symbolic Computation, PASCO '07, pages 24–32, New York, NY, USA, 2007. ACM.

18. Kumar, S. and Faraj, D.: Optimization of MPI_Allreduce on the Blue Gene/Q Supercomputer. In Proceedings of the 20th European MPI Users' Group Meeting, EuroMPI '13, pages 97–103, New York, NY, USA, 2013. ACM.

19. Fryxell, B., Olson, K., Ricker, P., Timmes, F. X., Zingale, M., Lamb, D. Q., MacNeice, P., Rosner, R., Truran, J. W., and Tufo, H.: FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. The Astrophysical Journal Supplement Series, 131(1):273, 2000.

20. Khanna, G., Catalyurek, U., Kurc, T., Kettimuthu, R., Sadayappan, P., Foster, I., and Saltz, J.: Using overlays for efficient data transfer over shared wide-area networks. volume 0, pages 1–12, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

21. Zerola, M., Sumbera, M., Lauret, J., and Barták, R.: Efficient multi-site data movement in distributed environment. In GRID, pages 171–172, 2009.

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

23. Bui, H., Finkel, H., Vishwanath, V., Habib, S., Heitmann, K., Leigh, J., Papka, M., and Harms, K.: Scalable Parallel I/O on a Blue Gene/Q Supercomputer Using Compression, Topology-Aware Data Aggregation, and Subfiling. In Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on, pages 107–111, Feb 2014.

24. Bui, H., Jung, E., Vishwanath, V., Leigh, J., and Papka, M.: Improving data movement performance for sparse data patterns on blue gene/q supercomputer. In 7th International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2) held in conjunction with the 43rd International Conference on Parallel Processing, September 2014.

25. Even, S., Itai, A., and Shamir, A.: On the complexity of time table and multi-commodity flow problems. In Foundations of Computer Science, 1975., 16th Annual Symposium on, pages 184–193. IEEE, 1975.

26. Yen, J. Y.: An algorithm for finding shortest routes from all source nodes to a given destination in general networks. Quart. Applied Math, 27:526–530, 1970.

27. Fourer, R., Gay, D. M., and Kernighan, B. W.: AMPL: A Modeling Language for Mathematical Programming. The Scientific Press, 1993.

28. AMPL: A Modeling Language for Mathematical Programming.

29. Chen, D., Eisley, N. A., Heidelberger, P., Senger, R. M., Sugawara, Y., Kumar, S., Salapura, V., Satterfield, D. L., Steinmacher-Burow, B., and Parker, J. J.: The IBM Blue Gene/Q Interconnection Network and Message Unit. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, pages 26:1–26:10, New York, NY, USA, 2011. ACM.

30. Chen, D., Eisley, N. A., Heidelberger, P., Senger, R. M., Sugawara, Y., Kumar, S., Salapura, V., Satterfield, D., Steinmacher-Burow, B., and Parker, J.: The IBM Blue Gene/Q Interconnection Fabric. IEEE Micro, 32(1):32–43, 2012.

31. Gilge, M. et al.: IBM System Blue Gene Solution Blue Gene/Q Application Development. IBM Redbooks, 2013.

32. Kumar, S., Mamidala, A. R., Faraj, D. A., Smith, B., Blocksome, M., Cernohous, B., Miller, D., Parker, J., Ratterman, J., Heidelberger, P., Chen, D., and Steinmacher-Burrow, B.: PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer. In Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS '12, pages 763–773, Washington, DC, USA, 2012. IEEE Computer Society.

33. Hurrell, J. W., Holland, M. M., Gent, P. R., Ghan, S., Kay, J. E., Kushner, P., Lamarque, J.-F., Large, W. G., Lawrence, D., Lindsay, K., et al.: The community earth system model: a framework for collaborative research. Bulletin of the American Meteorological Society, 94(9):1339–1360, 2013.

34. Buluç, A., Meyerhenke, H., Safro, I., Sanders, P., and Schulz, C.: Recent advances in graph partitioning. CoRR, abs/1311.3144, 2013.

35. Faanes, G., Bataineh, A., Roweth, D., Court, T., Froese, E., Alverson, B., Johnson, T., Kopnick, J., Higgins, M., and Reinhard, J.: Cray Cascade: A Scalable HPC System Based on a Dragonfly Network. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, pages 103:1–103:9, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

36. Chen, S. and Nahrstedt, K.: An Overview of Quality of Service Routing for Next-generation High-speed Networks: Problems and Solutions. Netwrk. Mag. of Global Internetwkg., 12(6):64–79, November 1998.

37. Subramanian, L., Stoica, I., Balakrishnan, H., and Katz, R. H.: OverQos: An Overlay Based Architecture for Enhancing Internet QoS. In Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1, NSDI'04, pages 6–6, Berkeley, CA, USA, 2004. USENIX Association.

38. Yuan, X.: Heuristic Algorithms for Multiconstrained Quality-of-service Routing. IEEE/ACM Trans. Netw., 10(2):244–256, April 2002.

39. Lin, X. and Shroff, N. B.: An Optimization-based Approach for QoS Routing in High-bandwidth Networks. IEEE/ACM Trans. Netw., 14(6):1348–1361, December 2006.

40. Xue, G., Zhang, W., Tang, J., and Thulasiraman, K.: Polynomial Time Approximation Algorithms for Multi-constrained QoS Routing. IEEE/ACM Trans. Netw., 16(3):656–669, June 2008.

41. Misra, S., Xue, G., and Yang, D.: Polynomial Time Approximations for Multi-Path Routing with Bandwidth and Delay Constraints, 2009.

42. Mondal, A., Sharma, P., Banerjee, S., and Kuzmanovic, A.: Supporting application network flows with multiple QoS constraints. In Quality of Service, 2009. IWQoS. 17th International Workshop on, pages 1–9, July 2009.

43. Jeyakumar, V., Alizadeh, M., Mazières, D., Prabhakar, B., Kim, C., and Greenberg, A.: EyeQ: Practical Network Performance Isolation at the Edge. In Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, nsdi'13, pages 297–312, Berkeley, CA, USA, 2013. USENIX Association.

44. Jacob, R., Larson, J., and Ong, E.: M× N communication and parallel interpolation in Community Climate System Model Version 3 using the model cou-

pling toolkit. International Journal of High Performance Computing Applications, 19(3):293–307, 2005.

45. Fan, Z., Cao, Z., Su, Y., Liu, X., Wang, Z., Liu, X., Zang, D., and An, X.: HiNetSim: A Parallel Simulator for Large-Scale Hierarchical Direct Networks. In Network and Parallel Computing, volume 8707 of Lecture Notes in Computer Science, pages 120–131. Springer Berlin Heidelberg, 2014.

VITA

Huy A. Bui

Email: abui4@uic.edu

Research Interest

Current research interests include the research and development of scalable system software for data movement optimization in high performance computing systems.

Education

Ph.D. in Computer Science 2009 - 2015
University of Illinois at Chicago (UIC), Chicago, IL.

Master of Science in Computer Engineering 2007 - 2009
Politecnico di Milano (Polimi), Como, Italy.

Bachelor of Science in Computer Science 2001 - 2006
Hanoi University of Science and Technology, Hanoi, Vietnam.

Working Experience

Graduate Research Assistant, Argonne National Lab, Lemont, IL 2012 - 2015

- **OPTIQ:** Designing and implementing a framework for **OPTI**mization and **Q**uality of service for data-centric applications on supercomputers. Build model to capture interconnection network topologies, system routing policies, application data movement patterns (flows) and QoS constraints. Optimize data movement to get maximum throughput while satisfying QoS constraints at scale. Written in C/C++.
- **BIOSAL:** Designed and implemented a part of transport layer in BIOSAL (a distributed BIOlogical Sequence Actor Library) to move data using Parallel Active Message Interface (PAMI) in the Blue Gene/Q supercomputer Mira. Written in C/C++.
- **Multi-path data movement:** Designed and implemented multi-path data movement for sparse data pattern on Blue Gene/Q (BG/Q) supercomputer Mira. Used Ford-Fulkerson algorithm to discover multiple paths available to move data between sources and destinations. Also used pipeline technique and low-level networking library to move data faster. Improved throughput up to 8 times for data movement. Demonstrated performance improvement for I/O on several applications up to 5 times. Written in C/C++.
- **Compressed Generic I/O:** Designed and implemented an I/O approach that leveraged topology information, subfiling mechanism and used a compression library BLOSC (BLOcking, Shuffling and lossless Compression) to improve I/O performance on BG/Q up to 2-3 times. Written in C/C++.

- **Low level communication libs:** Designed and implemented an API in C/C++ using low-level libraries (PAMI, uGNI) for communication in BG/Q and Cray systems for different communication modes such as one-sided (RDMA), two-sided, inter-node, intra-node, achieve up to 2-3 times higher performance.
- **Technology roadmaps:** Have been collecting and composing technology roadmaps every year for processors, GPU, memory, interconnect, storage, programming environments and tools for high performance computing (since 2011).

Graduate Assistant, University of Illinois at Chicago, Chicago, IL Summers '10,'11

- Designed, developed and maintained web applications for College of Medicine and Department of Disability and Human Development at UIC. Used MS SQL Server as backend database and ASP.NET with C# code behind as front end.
- Helped to administrate and manage computers, users of two above organizations.

Teaching Assistant, University of Illinois at Chicago, Chicago, IL 2009 - 2012

- Assisted teaching several courses at UIC including Software Design, Video Game Design and Development, Software Engineering I & II, Distributed Object Programming Using Middleware. Lectured, graded and helped students with their projects and lab sessions.
- Designed classes' projects such as a project to introduce cloud computing, MapReduce and Hadoop framework using Illinois Cloud Computing Testbed.

Skills

Languages: C/C++, Java, C#, ASP.Net, SQL, Linux Shell Script.
Libraries: MPI, OpenMP, Pthread, POSIX, HDF5.
Tools and OSes: Eclipse, Visual Studio, MS SQL Server, MySQL, Windows, Mac, Linux.
Others: Parallel and Distributed Computing, Hadoop, Network Programming.

Publications

[6] **H. Bui**, E. Jung, V. Vishwanath, A. Johnson, J. Leigh, M. E. Papka. Improving Sparse Data Movement Performance Using Multiple Paths on the Blue Gene/Q Supercomputer. *International Journal of Parallel Computing (PARCO) (submitted).*

[5] V. Vishwanath, **H. Bui**, M. Hereld, M. E. Papka. High Performance Parallel I/O *(Chapter 18 (GLEAN)) Oct. 2014.*

[4] **H. Bui**, E. Jung, V. Vishwanath, J. Leigh, M. E. Papka. Improving Data Movement Performance for Sparse Data Patterns on Blue Gene/Q Supercomputer, *Proc. of the 43nd International Conference on Parallel Processing Workshops (ICPPW) 2014. Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2), 2014.*

[3] **H. Bui**, V. Vishwanath, H. Finkel, K. Harms, J. Leigh, S. Habib, K. Heitmann, M. E. Papka. Scalable parallel I/O on Blue Gene/Q supercomputer using compression, topology-aware data aggregation, and subfiling, *The 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2014)*.

[2] **H. Bui**, V. Vishwanath, J. Leigh, M. E. Papka. Improving I/O Performance for Sparse Data Patterns on Leadership Systems, *The 8th Parallel Data Storage Workshop, PDSW13 [Poster]*.

[1] **H. Bui**, V. Vishwanath, J. Leigh, M. E. Papka. Evaluating Communication Performance in Supercomputers Blue Gene/Q and Cray XE6, *The International Conference for High Performance Computing, Networking, Storage and Analysis, SC12 [Abstract]*.

Honors and Awards

Student Travel Award for ICPP 2014

Graduate Student Council and Graduate College Travel Awards for traveling to SC12 conference, UIC, 2012.

Honorable Mention Teaching Assistant Award at Computer Science Dept, UIC, 2011-2012.

ICE Scholarship for Master of Science, Politecnico di Milano, Italy, 2007-2009.