High-performance Communication in MPI through Message Matching and Neighborhood Collective Design

by

Seyedeh Mahdieh Ghazimirsaeed

A thesis submitted to the
Department of Electrical and Computer Engineering
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
March 2019

Abstract

Message Passing Interface (MPI) is the de facto standard for communication in High Performance Computing (HPC). MPI Processes compute on their local data while extensively communicating with each other. Communication is therefore the major bottleneck for performance. This dissertation presents several proposals for improving the communication performance in MPI.

Message matching is in the critical path of communications in MPI. Therefore, it has to be optimized given the scalability requirements of the HPC applications. We propose clustering-based message matching mechanisms as well as a partner/non-partner message queue design that consider the behavior of the applications to categorize the communicating peers into some groups, and assign dedicated queues to each group. The experimental evaluations show that the proposed approaches improve the queue search time and application runtime by up to 28x and 5x, respectively.

We also propose a unified message matching mechanism that improves the message queue search time by distinguishing messages coming from point-to-point and collective communications. For collective elements, it dynamically profiles the impact of each collective call on message queues and uses this information to adapt the queue data structure. For point-to-point elements, it uses partner/non-partner queue design. The evaluation results show that we can improve the queue search time and application runtime by up to 80x and 5.5x, respectively.

Furthermore, we consider the vectorization capabilities of used in new HPC systems

many-core processors/coprocessors to improve the message matching performance. The evaluation results show that we can improve the queue search time and application runtime by up to 4.5x and 2.92x, respectively.

Finally, we propose a collaborative communication mechanism based on common neighborhoods that might exist among groups of k processes. Such common neighborhoods are used to decrease the number of communication stages through message combining. We consider two design alternatives: topology-agnostic and topology-aware. The former ignores the physical topology of the system and the mapping of processes, whereas the latter takes them into account to further optimize the communication pattern. Our experimental results show that we can gain up to 8x and 5.2x improvement for various process topologies and a sparse matrix-matrix multiplication kernel, respectively.

Acknowledgments

I would like to thank my supervisor, Prof Ahmad Afsahi, for his priceless guidance and support throughout my PhD research studies. Prof. Afsahi's enthusiasm for high quality research helps me realize how enjoyable research could truly be. I would also like to thank Dr. Ryan Grant from Sandia National Laboratories for his endless support and insightful guidances. It was an honor to collaborate with him and his colleagues, Dr. Matthew G. F. Dosanjh from Sandia National Laboratories and Dr. Patrick G. Bridges from University of New Mexico. Thanks to my PhD Supervisory Committee for their valuable feedback regarding the research conducted in this dissertation.

I would like to acknowledge the Natural Science and Engineering Research Council of Canada (NSERC), the school of graduate studies at Queen's University as well as the Electrical and Computer Engineering (ECE) Department of Queen's University for their financial support. I would also like to thank Compute Canada for access to their large-scale resources such as GPC, Niagra in SciNet, and Graham in SHARCNET. Special thanks to Dr. Scott Northrup and Dr. Grigory Shamov for their technical support.

Many thanks to all my current and former colleagues at Parallel Processing Research Laboratory, Dr. Judicael Zounmevo, Dr. Iman Faraji, Dr. Hessam Mirsadeghi, Kashual Kumar, Mac Fregeau, Pedram Alizadeh, Leila Habibpour and Yiltan Temucin. Additional thanks to Hessam for his collaboration and his unconditional help whenever I needed. Thanks to Judicael, Iman, Kaushal and Mac for the constructive discussions. I would also like to thank Pedram, Leila and Yiltan for bringing more joy to the lab by their presence.

Finally, I would like to thank my family for all their love and support. My gratitude goes to my parents, Fatemeh and Mostafa, who mean the world to me. To my brother, Mojtaba, and his lovely family, Arezo and Hannah. I thank him for always being my role model in life. My thanks also go to my sister, Samaneh, and her beloved daughter, Silvana. I thank her for the infinite care and support. I would also like to thank her husband, Dr. Marzband, for all his guidances. Last but not least, my thanks go to my understanding boyfriend, Mohammadreza, for all his care, love and patience.

Contents

Abstract **i**

Acknowledgments **iii**

Contents **v**

List of Tables **ix**

List of Figures **x**

Glossary **xv**

1 Introduction **1**
1.1 Motivation . . . 2
1.2 Problem Statement . . . 4
1.3 Contributions . . . 5
1.4 Organization of the Dissertation . . . 8

2 Background **10**
2.1 Multi-core and Many-core Processors . . . 11
2.1.1 Intel Xeon Phi . . . 11
2.2 Message Passing Interface (MPI) . . . 13
2.2.1 Groups and Communicators . . . 14
2.2.2 Point-to-Point Communications . . . 15
2.2.3 Collective Communications . . . 15
2.2.4 One-sided Communications . . . 18
2.2.5 MPI Message Queues . . . 18
2.2.6 MPI Topology Interface . . . 20
2.2.7 Neigborhood Collective Communications . . . 22
2.3 InfiniBand: A High Performance Interconnect . . . 24

3 Clustering-based Message Matching Mechanism **26**
3.1 Related Work . . . 27
3.2 Motivation . . . 29
3.3 The Proposed Clustering-based Message Matching Design . . . 32

3.3.1 Heuristic-based Clustering Algorithm . . . 36
3.4 Performance Results and Analysis . . . 37
3.4.1 Experimental Platform . . . 37
3.4.2 Message Queue Traversals . . . 38
3.4.3 Queue Search Time . . . 42
3.4.4 Application Runtime . . . 44
3.5 Summary . . . 48

4 Partner/Non-partner Message Queue Data Structure 49
4.1 Motivation . . . 50
4.2 The Proposed Partner/Non-partner Message Queue Design . . . 53
4.2.1 Metrics for Selecting Partner Processes . . . 53
4.2.2 The Static Approach . . . 54
4.2.3 The Dynamic Approach . . . 57
4.3 Complexity Analysis . . . 62
4.3.1 Runtime Complexity . . . 62
4.3.2 Memory Overhead Complexity . . . 67
4.4 Performance Results and Analysis . . . 69
4.4.1 Selecting the Partners . . . 69
4.4.2 Queue Search Time . . . 70
4.4.3 Application Execution Time . . . 74
4.4.4 Number of Levels and Partners . . . 75
4.4.5 Overhead of Extracting Partners . . . 76
4.5 Related Work . . . 77
4.6 Summary . . . 79

5 A Unified, Dedicated Message Matching Engine for MPI Communications 81
5.1 Motivation and Related Work . . . 82
5.2 The Proposed Unified Message Queue Design . . . 84
5.2.1 The COL Message Queue Design for Collective Elements . . . 85
5.2.2 The Unified Queue Allocation Mechanism for Collective and Point-to-Point Elements . . . 88
5.2.3 The Unified Search Mechanism for Collective and Point-to-Point Elements . . . 90
5.3 Experimental Results and Analysis . . . 92
5.3.1 Experimental Platform . . . 92
5.3.2 Microbenchmark Results . . . 93
5.3.3 Application Queue Search Time . . . 95
5.3.4 Number of Dedicated Queues for the Applications in COL+PNP Approach . . . 101
5.3.5 Application Execution Time with COL+PNP and CPL+LL approaches 105
5.3.6 Runtime Overhead of the Message Queue Design . . . 106

5.4 Summary . . . 106

6 Message Queue Matching Improvement on Modern Architectures 108
6.1 Motivation and Related Work . . . 109
6.2 Techniques to Improve MPI Message Matching . . . 111
6.2.1 Linked List of Arrays Queue Data Structure . . . 112
6.2.2 Linked List of Vectors Queue Data Structure . . . 115
6.3 Complexity Analysis . . . 117
6.3.1 Runtime Complexity . . . 117
6.3.2 Memory Overhead Complexity . . . 118
6.4 Performance Results and Analysis . . . 119
6.4.1 Experimental Platform . . . 119
6.4.2 The Size of Array in each Linked List Element . . . 120
6.4.3 Microbenchmark Results . . . 122
6.4.4 Application Results . . . 126
6.5 Summary . . . 128

7 MPI Neighborhood Collective Optimization 130
7.1 Related Work . . . 132
7.2 Preliminaries . . . 133
7.2.1 Common Neighbors and Friend Groups . . . 133
7.2.2 Number of communications . . . 134
7.2.3 Main Steps of the Design . . . 135
7.3 Communication Pattern Design . . . 136
7.3.1 The Collaborative Mechanism . . . 136
7.3.2 Building the Friendship Matrix . . . 140
7.3.3 Mutual Grouping of the Processes . . . 142
7.3.4 Complexities . . . 147
7.4 The Proposed Communication Schedule Design . . . 148
7.5 Design Parameters Selection . . . 150
7.5.1 Topology-agnostic Approach . . . 150
7.5.2 Topology-aware Approach . . . 151
7.6 Experimental Results and Analysis . . . 152
7.6.1 Moore Neighborhoods . . . 153
7.6.2 Random Sparse Graph . . . 160
7.6.3 Application Results and Analysis . . . 166
7.7 Summary . . . 168

8 Conclusions and Future Work 170
8.1 Conclusion . . . 170
8.2 Future Work . . . 174

References 180

1 Appendix: K-means Clustering 194

List of Tables

3.1 Different combination of groups of processes assigned to each queue 31

3.2 Maximum number of (UMQ/PRQ) queue searches in the applications . . . 46

4.1 List of parameters and their definitions 63

5.1 List of parameters used for collective queue allocation and search mechanism 89

5.2 Overhead/search time ratio in COL+PNP 106

7.1 The list of parameters 137

List of Figures

2.1 An example of an HPC cluster 11

2.2 Intel Xeon Phi Knight Corner coprocessor silicon (redrawn from [65]) 13

2.3 Software stack in a parallel system 14

2.4 A few examples of algorithms for different collective operations 17

2.5 Linked list data structure in MPICH/MVAPICH 20

2.6 Open MPI message queue data structure 21

2.7 An example of process topology with the status of buffers for neighbor allgather and neighbor alltoall operations from the viewpoint of process P_7 . . 24

3.1 Average UMQ search time for the reverse search on Cluster A 30

3.2 Average UMQ search time for the reverse search on Cluster A 32

3.3 The first phase of gathering information and clustering in the proposed message matching mechanism 34

3.4 Clustering-based message queue structure 35

3.5 Average number of traversals for AMG2006 in different approaches on Cluster A with 1k processes 40

3.6 Average number of traversals for LAMMPS in different approaches on Cluster A with 240 processes 41

3.7 Average PRQ and UMQ search time in the AMG2006 application on Cluster A 42

3.8 Average PRQ and UMQ search time in the LAMMPS application on Cluster A 44

3.9 PRQ and UMQ search time for process 0 in the FDS application with K-means and heuristic algorithms on Cluster A 45
3.10 PRQ and UMQ search time for process 0 in the FDS application with K-means algorithm on Cluster A 46
3.11 Application runtime in AMG2006, LAMMPS and FDS applications on Cluster A 47

4.1 Number of elements sent to UMQ/PRQ from different processes in the AMG2006 and LAMMPS applications on Cluster A 52
4.2 Static partner/non-partner message queue design 55
4.3 Dynamic partner/non-partner message queue design 59
4.4 Selected partners using the average metric in AMG2006 and LAMMPS with threshold = 100 in the Dynamic approach on Cluster A 71
4.5 PRQ and UMQ search time speedup of partner/non-partner design over linked list in AMG2006, LAMMPS and FDS applications on Cluster A . . . 72
4.6 FDS application runtime speedup over linked list with unbounded memory on Cluster A 75
4.7 FDS application runtime speedup over linked list with bounded memory and using average as the partner extraction metric on Cluster A 75
4.8 Number of levels in FDS application on Cluster A 76
4.9 Number of extracted partners in FDS application for different number of processes and $t = 100$ on Cluster A 76
4.10 Partner extraction overhead/search time ratio on Cluster A 77

5.1 Average number of elements in the queues from collective and point-to-point communications across all processes in different applications (512 processes) in Cluster A 84
5.2 The proposed unified message matching mechanism 85

5.3 Proposed message matching mechanism for collective elements 87
5.4 Latency improvement in MPI_Gather, MPI_Allreduce and MPI_Iallgather, for k=1 on Cluster A 94
5.5 Average UMQ and PRQ search time speedup for collective elements with COL+LL approach and different k values in Radix, Nbody, MiniAMR and FDS in Cluster A 96
5.6 Average UMQ and PRQ search time speedup for collective, point-to-point and total elements with COL+PNP and COL+LL approaches and $k = 16$ in Radix, Nbody, MiniAMR and FDS in Cluster B 99
5.7 Number of dedicated UMQs and PRQs for collective operations with COL+PNP approach and $k = 16$ in Radix, Nbody, MiniAMR and FDS in Cluster B . . 103
5.8 FDS runtime speedup over MVAPICH in Cluster B 105

6.1 Queue search time in different data structures (MVAPICH2 and OPEN MPI) and hardwares (Intel Xeon as host, and Xeon Phi KNC) on cluster C 110
6.2 Linked list of arrays queue data structure 112
6.3 Managing the holes and incoming messages in the linked list of arrays queue data structure 114
6.4 Managing the holes and incoming messages in the linked list of vectors queue data structure with vec_size 16 116
6.5 Packing data structures into 64 byte cache lines 120
6.6 Different match order in microbenchmark tests 121
6.7 Queue Search Time in forward search, middle search and backward search with Xeon Phi KNC coprocessor on Cluster C 123
6.8 Queue search time with 50% wildcard receive in backward search with Xeon Phi KNC coprocessors on Cluster C 125

6.9 Queue Search Time in forward search, middle search and backward search with Xeon processor on Cluster B 126
6.10 PRQ and UMQ search time speedup of linked list of arrays approach over linked list in AMG2006 and FDS on Cluster B 127
6.11 FDS application runtime on Cluster B 128

7.1 An example of process topology graph: processes cn_1, cn_2, ..., cn_m are common neighbors of processes P_1, P_2, ..., P_k 134
7.2 Matrices used in the distributed message combining mechanism 140
7.3 An example of generating a row of matrix T from matrix A 142
7.4 A sample Moore neighborhood with d = 2 and r = 1, 2. The neighbors are shown in green for node P 154
7.5 Number of hyperedges for Moore neighborhood for 4K processes on Cluster D. Missing bars represent a zero value 155
7.6 MPI_Ineighbor_allgather speedup for Moore neighborhood for 4K processes- 4 byte message size on Cluster D 157
7.7 MPI_Ineighbor_allgather speedup for Moore neighborhood with d=2, r=4 for different message sizes and 4K processes on Cluster D 159
7.8 The overhead of Algorithm 1 for Moore neighborhood with 4K processes on Cluster D 160
7.9 Number of hyperedges in Random sparse graph with topology-aware design and 1K processes on Cluster B. Missing bars represent a zero value 162
7.10 MPI_Ineighbor_allgather speedup in Random sparse graph with topology-aware design and 1K processes- 4 bytes message size on Cluster B 163
7.11 Neighbor allgather speedup for Random Sparse Graph with topology-aware design for different message sizes and 1K processes on Cluster B 164

7.12 The overhead of Algorithm 1 for random sparse graph with topology-aware design for 1K processes on Cluster B 165
7.13 An example of building the process topology graph for the SpMM kernel based on the non-zero elements of the input matrix A 167
7.14 The speedup of SpMM for various input matrices with neighborhood collectives over ordinary collectives on Cluster B 168
7.15 The speedup of SpMM for various input matrices with optimized topology-aware neighborhood collectives over the default neighborhood collectives on Cluster B 169

Glossary

API Application Programming interface.

AVX Advanced Vector Extensions.

CA Channel Adapter.

CFD Computational Fluid Dynamic.

FDS Fire Dynamic Simulator.

FLOPS FLoating-point Operations Per Second.

GDDR Graphics Double Data Rate.

GPC General Purpose Cluster.

HCA Host Channel Adapters.

HPC High-Performance Computing.

IBA InfiniBand Architecture.

IMIC Intel Many Integrated Core.

JCAHPC Joint Center for Advanced High Performance Computing.

KNC Knights Corner.

KNL Knights Landing.

MCDRAM Multi-Channel Dynamic Random Access Memory.

MIC Many Integrated Core.

MPI Message Passing Interface.

MPP Massively Parallel Processors.

MPSS MIC Platform Software Stack.

NERSC National Energy Research Scientific Computing Center.

NIC Network Interface Card.

NUMA Non-Uniform Memory Access.

OS Operating System.

PCIe Peripheral Component Interconnect express.

PGAS Partitioned Global Address Space.

PQ Profiling Queue.

PRQ Posted Receive Queue.

QE Queue Element.

RDMA Remote Direct Memory Address.

RMA Remote Memory Access.

SE Searching Element.

SMP Symmetric Multi-Processor.

SpMM Sparse Matrix-Matrix Multiplication.

TD Tag Directory.

UMQ Unexpected Message Queue.

VPU Vector Processing Units.

Chapter 1

Introduction

High Performance Computing (HPC) is the key in tackling the problems in different application domains such as Fire Dynamic Simulation in chemistry [89], Computational Fluid Dynamic (CFD) [92] and thermodynamics in physics [78], N-Body simulations in cosmology [103, 22, 57] and deep learning applications [38, 21]. These are just a few representative examples of HPC application domains. These applications require an enormous computational capability that significantly exceeds what is provided by ordinary computers. The HPC community investigates novel hardware and software computing architectures and techniques to provide high-end computing power to scientific domain and engineering.

The key approach to satisfy the ever increasing demands for more computational power in HPC systems is parallel processing. In parallel processing, an application is decomposed into a number of tasks that can be executed simultaneously. Each task is assigned to a process/thread which is in turn executed by one of many processing elements. To enable the parallel execution of an application's tasks, HPC systems are designed with an ever-increasing number of processors to provide substantial computing power. Such increases in computational power has brought HPC to the petascale era with systems capable of performing operations at the rate of 10^{15} FLoating-point Operations Per Second (FLOPS). Currently, Summit, Sierra and Sunway TaihuLight are the top publicly known supercomputers in the world [18], each having millions of processors. However, this computational

power is still not sufficient for the size and complexity of the current challenges and emerging problems. Therefore, the HPC community has been aiming to move towards the exascale era (10^{18} FLOPS) by expanding the computational boundaries even further.

To take advantage of hardware resources in HPC systems, the software layers should deliver as much performance to the applications as the system has to offer. This is challenging since the HPC systems are deployed at large scales and in a variety of complex ways.

One of the most important bottlenecks in achieving high performance in HPC systems is the communication between processing elements. The processes in an HPC system are distributed and they require a collaborative mechanism to exchange data or intermediate results with each other. As the number of processing elements in HPC systems increases, the speed gap between communication and computation is widening, making the communication the performance bottleneck. As a result, enhancing communication performance is highly crucial for the performance of parallel applications. In order to support explicit communication between the processes, the Message Passing Interface (MPI) [8] programming model is mostly used in high-performance applications. There are other programming paradigms available such as OpenMP [17] and Partitioned Global Address Space (PGAS) [15], however, the main focus of this dissertation is on MPI since it is the implementation vehicle of massive amount of existing HPC applications and is widely supported on existing supercomputers.

1.1 Motivation

Delay in any single peer-to-peer communication can propagate to subsequent communications and computations, impacting all other processes of a job. This harmful effect of latency propagation becomes more of an issue at large scale where a large number of processes are involved. One of the main sources of delay in MPI communication is *message queue* operations. Message queues are in the critical path of MPI communications and thus, the

performance of message queue operations can have significant impact on the performance of applications. They are used in MPI libraries to cope with the unavoidable out-of-sync communications between the processes. It is impossible for a process to instantly consume all the messages sent by other processes running on the same or remote nodes. Therefore, the messages must be queued at the receiving side.

The usage of message queues is not specific to MPI; it is a well-known mechanism to deal with the limited resources, such as memory buffers, that are required for message reception and processing [26]. Message queues are featured as the most critical data structure in MPI because of its frequent usage [23, 70]. In other words, message queue operations determine the performance of communication-intensive HPC applications. Underwood and Brightwell [110] show that message queue operations can account for up to 60% of the communication latency. On the other hand, MPI message queues are growing proportionally because of two main reasons. The first reason lies in the behavior of HPC applications that are increasing in scale [30, 31, 33, 70]. The second reason is the growing number of CPU cores and consequently processes in the new parallel computing systems with many-core processors and coprocessors. Considering the scale of the current and emerging applications and systems, it is crucial to design novel message queue data structures and improve the message matching performance so that each process can communicate with other processes in an efficient way. This efficiency is defined not only based on the speed of operation, but also in terms of memory footprint of the new message queue data structures. In essence, the new message queue design should account for the amount of memory resource that each CPU core can afford to reserve for an exponentially growing number of CPU cores in modern parallel computing systems.

Although various message matching mechanisms have been proposed by researchers [116, 46, 27, 71], there is still a high demand to come up with new designs considering the properties of the emerging HPC systems with their multi-core and many-core processors/coprocessors, and also the scalability requirements of the HPC applications. In fact,

we require new message matching mechanisms that consider the application behavior as well as the features available in hardware architectures to efficiently improve the speed of operation and memory requirements of the HPC applications on the target system.

Another source of latency in MPI is the global communications among the parallel processes of an application. With the increasing scale of supercomputers, new parallel algorithms are required at the application level that can scale with the increase in the number of cores. These algorithms try to limit communications to a sparse neighborhood of each process. Therefore, they inherently incur less latency and provide more scalability. In order to support such algorithms, MPI-3.0 [8] recently introduced the neighborhood collective communications support on top of the process topology interface of the standard. Neighborhood collectives provide another opportunity to enhance the communication performance by exploiting the virtual topology of the processes. Virtual topology represents the way MPI processes communicate.

1.2 Problem Statement

In this dissertation, we seek to address the following questions:

- How the *clustering* approaches can be used to improve the message queue operations in MPI? How can we design a heuristic-based clustering algorithm to improve message matching performance? What is the impact of the heuristic-based algorithm as compared to K-means clustering on message queue performance?

- How can we define the term *partnership* between peer processes and how can we use it to improve the performance of message queue operations? How can we design an asymmetric message queue subsystem that more communication resources or faster communication paths are assigned to partner processes?

- What is the impact of different types of communications on MPI message matching?

How can we improve message matching performance by designing a message matching mechanism that considers the type of communications?

- What is the performance of message queue operations on the new HPC architectures with their multi-core and many-core processors/coprocessors? How can we take advantage of the features available in such new HPC systems to improve the message matching performance?

- How can we improve the performance of sparse neighborhood collective communications in MPI? How to extract information from a distributed process topology and use it to design an efficient communication schedule for MPI neighborhood communication?

1.3 Contributions

This dissertation presents several proposals to improve message-passing performance and scalability in MPI and applications that use them. It contributes by addressing challenges in MPI that impact the performance of MPI communication. These challenges include MPI message queue operations and sparse neighborhood communications.

1. Clustering-based Message Queue Data Structure

In Chapter 3, we propose an MPI message matching mechanism that considers the behavior of the applications to categorize the communicating peers into some groups and assign a dedicated message queue to each group [48]. Grouping the processes is done based on the number of queue elements each communicating process adds to the message queue at runtime. The proposed approach provides an opportunity to parallelize the search operation for different processes based on the application's message queue characteristic. For grouping the processes, two different algorithms are used: *K-means* clustering and a heuristic algorithm. The evaluation results show that the proposed algorithm can reduce the

number of traversals significantly. It can also improve the queue search time and application runtime by up to 2.2x and 1.33x, respectively.

2. Partner/Non-partner Message Queue Data Structure

In Chapter 4, we propose a new message matching mechanism for MPI that can speed up the search operation by allocating dedicated queues for processes with high frequency of communications [52]. One way to reduce the queue search time is to design an asymmetric message queue subsystem where faster communication paths would be allocated to busier processes. Processes in MPI applications typically communicate more with a subset of other processes. These processes are called *partners*. The information about partner processes such as their dedicated queue number is saved in a hash table.

Two types of design are proposed: static and dynamic. While the Static approach works based on the information from a profiling stage, the Dynamic approach utilizes the message queue characteristics at runtime. The experimental evaluations show up to 5x speedup for the FDS application [89] which is highly affected by the message matching performance. It is also shown that the proposed design can provide as high as 25x reduction in queue search time for long list traversals without degrading the performance for short list traversals.

3. A Unified, Dedicated Message Matching Engine for MPI Communications

Chapter 5 considers the type of communication to improve the queue search time. MPI provides support for different types of communications such as *point-to-point* and *collective* communication. Point-to-point communication represents the basic communication mechanism in MPI that allows sending and receiving messages between two individual processes. On the other hand, collective communications provide an abstraction for communications among a group of processes rather than just two. The incoming messages to the message queues could be due to a point-to-point or collective communication operations. In this chapter, we propose a communication optimization that dynamically profiles the impact of

different types of communications on message matching performance and uses this information to allocate dedicated message matching resources to both point-to-point and collective communications [50, 49]. We demonstrate that our dynamic MPI communication optimizations accelerate the collective and point-to-point queue search time up to 80x and 71x, respectively.

4. Message Matching Improvement on Modern Architectures

Chapter 6 addresses performance of MPI message queues on many-core systems. Unfortunately, the performance of current MPI libraries on these systems is significantly worse than on traditional systems. Many-core processors and coprocessors such as Intel Xeon Phi [65, 66] are used in supercomputers because of their energy efficiency and massive parallelism. However, well-known MPI libraries are designed for traditional heavy-weight cores with large amount of serial compute power. Recent studies have shown that system message rates that were previously bottlenecked by networking overheads are now instead limited by compute core performance on many-core systems [25]. It is therefore vital to optimize message matching performance for emerging many-core systems to enable scalability in future machines.

To this aim, we propose techniques to take advantage of vectorization capabilities on modern processor architectures to enhance message matching performance [40]. First, we explore spatial locality by combining multiple entries into a single linked list element. We also expand on this idea by rearranging data into AVX vectors [83] to take advantage of Intel Xeon Phi intrinsic functions. The experimental results show that these techniques are effective with both Intel Xeon Phi and traditional Intel Xeon processors and we can gain up to 4.5x and 2.92x performance for applications with extreme message matching requirements, respectively.

5. MPI Neighborhood Collective Optimization

In Chapter 7, we propose an algorithm to improve the performance of neighborhood collectives in MPI by designing efficient communication schedules. More specifically, we propose a distributed algorithm that can be used to extract message-combining communication patterns and schedules for neighborhood collectives [51]. We show that part of the problem falls within the scope of maximum matching in weighted hypergraphs, where we seek to find a mutual pairing of the processes that have neighbors in common. We consider two design alternatives: topology-agnostic and topology-aware. The former ignores the physical topology of the system and the mapping of processes, whereas the latter takes them both into account to further optimize the communication pattern. The physical topology represents the connections between the cores, chips, and nodes in the hardware. Our experimental results show that we can gain up to 8x reduction in the communication latency of neighborhood communication, and around 5x speedup for a sparse matrix-matrix multiplication kernel.

1.4 Organization of the Dissertation

As discussed earlier, the proposals in this dissertation are on two main research fronts in MPI: message matching and sparse neighborhood communication. Chapter 2 provides some background information related to these research fronts. It discusses MPI and its specific features that are used in these proposals. It also briefly discusses parallel computers and HPC network technologies. In Chapter 3, we propose message queue mechanisms based on clustering algorithms to improve MPI message matching. Chapter 4 proposes partner/non-partner message queue data structure to further improve the performance. In Chapter 5, we look at the message matching issue from a different perspective and propose a new message matching mechanism based on the type of communication. Chapter 6 addresses improving the performance of MPI message queues considering the hardware/software features of the new many-core processors and coprocessors. Chapter 7 discusses our proposed approach

for designing optimized communication schedules for MPI neighborhood collective communications. Finally, Chapter 8 concludes this dissertation and outlines some future research directions.

Chapter 2

Background

Parallel computers utilize the processing power of multiple processors/computing nodes and coordinate their computational efforts to deal with the limitations of the sequential computers that cannot provide the computing power required by most applications. In other words, they divide the computational tasks among multiple processors to enhance performance. One of the main classes of parallel computers is cluster computers. As shown in Figure 2.1, a cluster consists of compute nodes that are connected to each other through a high-performance interconnect. The nodes in a cluster can have either a symmetric multiprocessor (SMP) or Non-Uniform Memory Access (NUMA) configuration. An SMP provides symmetric access to the shared memory for all nodes. However, NUMA provides non-uniform memory access to the local and remote parts of the memory. Another important class of parallel computers is Massively Parallel Processors (MPP). In MPPs, processing units are tightly interconnected together usually with a custom-designed interconnect. MPPs are more expensive than clusters due to their custom and proprietary design.

Clusters are more popular than other parallel architectures since they provide high availability and performance-cost ratio. Moreover, they are flexible in configuration and support a wide range of available software. At the time of writing this document, 86.4% of the top 500 supercomputers in the world are clusters [18]. Clusters can be either homogeneous with just the main processors, or heterogeneous that include the coprocessors (such as Intel Xeon

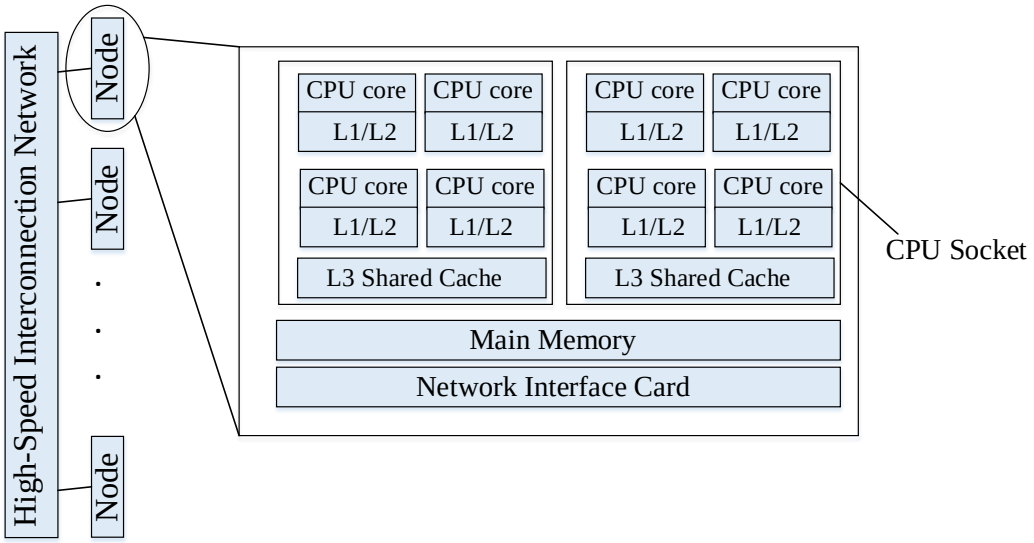


Figure 2.1: An example of an HPC cluster

Phi [65, 66]) or accelerators (such as GPUs) along with the processors.

2.1 Multi-core and Many-core Processors

Nowadays, it is end of the "era of higher processor speeds", and microprocessor industry is finding the way to the "era of higher processor parallelism". This way, higher parallelism is provided in computer design which leads to a better performance for applications. To this aim, multi-core and many-core processors/coprocessors are designed and utilized in HPC systems. Multi-core processors contain a few simple, independent cores. On the other hand, many-core processors/coprocessors such as Intel Xeon Phi have a large number of cores. Figure 2.1 shows a simplified architecture of an HPC cluster with multi-core processors.

2.1.1 Intel Xeon Phi

Intel Xeon Phi [65, 66] is a many-core processor/coprocessor with Many Integrated Core (MIC) architecture that is known for its high computing capacity, low power consumption, x86 instruction set support and ability to work as accelerators for conventional processors.

It is used in some of the world's largest supercomputers including Cori [1] at the National Energy Research Scientific Computing Center (NERSC) and Oakforest-PACS [12] at Joint Center for Advanced High Performance Computing (JCAHPC).

There are two main generations of Xeon Phi: Knights Corner (KNC) [65] and Knights Landing (KNL) [66]. The Intel Xeon Phi KNC coprocessors can connect to the main processor by a PCI Express (PCIe) [34] bus. They can also communicate with each other using the PCIe peer-to-peer interconnect or other networks such as InfiniBand [4] or Ethernet directly. There is a virtualized TCP/IP stack implemented over the PCIe bus that gives the opportunity to the users to access the coprocessor as a network node. Intel distributed a MIC Platform Software Stack (MPSS) [6], which consists of an embedded Linux, a minimally modified GCC, and driver software. In this Linux environment, different programming models such as MPI and OpenMP are supported.

Figure 2.2 shows the main components of the Intel Xeon Phi KNC coprocessor including cores, caches, memory controllers, PCIe client logic, and a very high bandwidth, bidirectional ring interconnect. Each private L2 cache is dedicated to one core and can be accessible from other cores through global distributed Tag Directory (TD). The cores can access the PCIe bus through PCIe client logic. Moreover, the GDDR Memory Controller provides an interface to the GDDR on the coprocessor. KNC has up to 61 CPU cores.

The next-generation Intel Xeon Phi (KNL) provides greater enhancement compared to KNC. KNL is built with up to 72 CPU cores and is shown to deliver 3 times the performance of KNC chip. The basic KNL component is called a tile. Each tile consists of two cores along with two Vector Processing Units (VPUs). The cores in a tile share 1MB of L2 cache. The tiles are linked to each other over a 2D mesh which provides the cache coherency between the L2 caches on the die.

One interesting thing about KNL is that it is designed to self-boot the operating system as a native processor. Another difference between the KNC Xeon Phi generation and the KNL generation is the use of Intel Advanced Vector Extensions (AVX). Intel AVX is a set

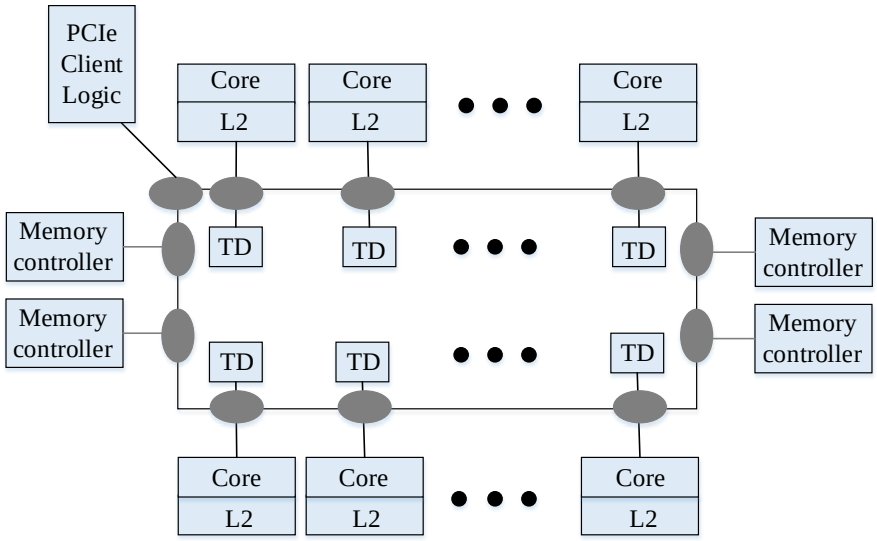


Figure 2.2: Intel Xeon Phi Knight Corner coprocessor silicon (redrawn from [65])

of instructions for doing the same operation on multiple data points simultaneously. KNL supports AVX-512 instructions [100] (e.g., Intel AVX intrinsics) which makes it binary compatible with Intel Xeon processor instructions. However, KNC does not support AVX and has its own instruction set known as Intel Many Integrated Core (IMIC).

2.2 Message Passing Interface (MPI)

The MPI forum [8] has introduced MPI as a standard that improves execution of parallel applications by exchanging data. In other words, MPI extends languages like C, C++ and Fortran and it is known as one of the most commonly-used programming paradigms in HPC. As can be seen in Figure 2.3, MPI decouples parallel applications from the underlying communication layers. It can be used in both distributed-memory and shared-memory architectures.

It should be noted that MPI is not a library implementation; it only specifies the features the library should have to provide scalability, portability and efficiency. MPICH [10], MVAPICH [11] and Open MPI [14] are example of some open source implementations of

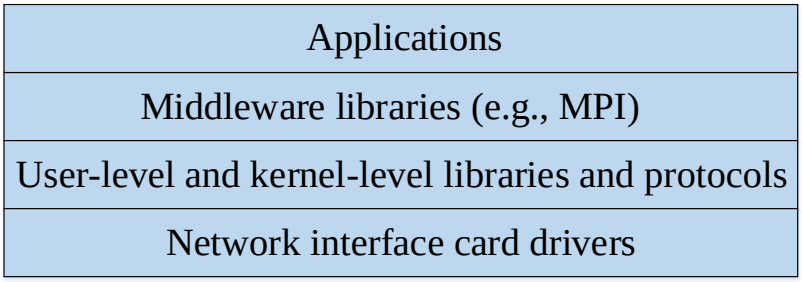


Figure 2.3: Software stack in a parallel system

MPI. MPICH is a reference implementation of MPI from Argonne National Laboratory that is used as source code for other implementations like MVAPICH, maintained by Ohio State University. The difference between MVAPICH and MPICH is that MVAPICH is optimized over high speed interconnection networks like InfiniBand [4] and iWARP [16]. Another open source implementation of MPI used by many Top500 supercomputers is Open MPI. Intel MPI [5], IBM Spectrum MPI [3] and Cray MPI [53] are examples of commercial implementations of MPI.

In order to execute a job in MPI, processes should be able to communicate with each other. In other words, the data should move from address space of one process to the address space of another process. In the following sections, we review the key concepts related to communications in MPI.

2.2.1 Groups and Communicators

The concept of groups and communicators in MPI is used to define the scope and context of all communications. A group is an ordered number of processes, each associated with a unique rank which is an integer number between zero and $N-1$ (N is the number of ranks). When a program initializes, the system assigns the rank number to each process. Communicators use groups to represent the participants in each communication. The standard defines a predefined communicator MPI_COMM_WORLD that includes all the processes of an application. The standard provides various Application Programming interface (APIs)

that can be used to duplicate a communicator or create sub-communicators. Each process can be a member of more than one communicator, having a separate rank with respect to each.

2.2.2 Point-to-Point Communications

There are different types of communications supported in MPI. Point-to-point or two-sided communication is the basic communication mechanism in which both sender and receiver take part in the communication. It is guaranteed by the MPI standard that the messages are received in the order they are sent. The sender has a buffer that holds the message and an envelop containing information that will be used by the receiver side. The receiver uses the information in the envelop to select the specified message and stores it into its receiver buffer. Researchers have proposed different approaches to improve point-to-point communication performance [73, 95, 117, 113, 99, 55].

In MPI, the point-to-point communication can be either blocking or non-blocking. In blocking send operation, the calling process is blocked until its communication buffer can be used again. The blocking receiver blocks the receiver until the receive operation is completed. On the other hand, the non-blocking send operation returns as soon as the data is copied into the communication buffer, and non-blocking receive returns as soon as the receive request is posted. The advantage of non-blocking communication is that they provide the opportunity to overlap communication and computation. However, they require polling or waiting mechanisms to verify the completion of the message transfer.

2.2.3 Collective Communications

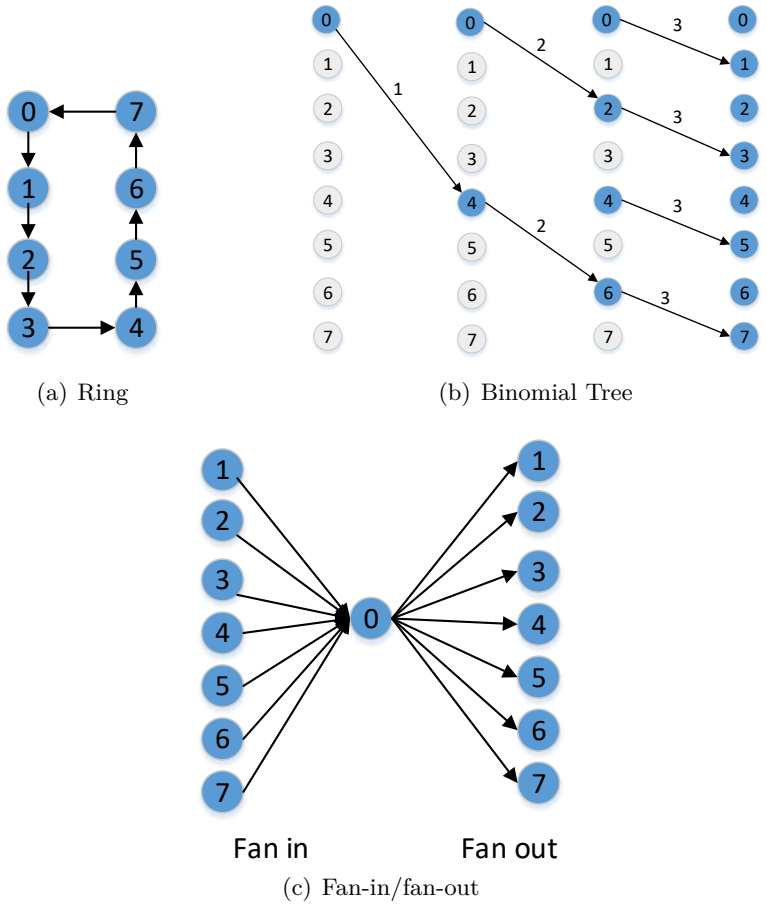
In collective communication, the messages can be exchanged among a group of processes rather than just two of them. Collective communications provide this opportunity for the processes to perform one-to-many and many-to-many communications in a convenient, portable, and optimized way. Some examples of collective communications include barrier,

broadcast, allgather, alltoall and allreduce. The barrier operation enables explicit synchronization across the processes, whereas other operations enable a certain type of collective data exchange among the processes. MPI allgather is a many-to-many collective communication where each process gathers data from every other process in the communicator. In other words, each process has a message that has to be sent to every other process.

Collective communications are extensively used by the MPI applications. Therefore, the performance of collective operations can significantly impact the performance of MPI applications. Consequently, there have been numerous research to improve the performance of collective communications in MPI [44, 90, 64, 20, 43, 76, 80, 105].

There are two different approaches to design collective communications: unicast-based algorithms and hardware-based implementations. In the unicast-based algorithms, MPI collectives are implemented as a series of point-to-point communications. On the other hand, hardware-based implementations exploit special hardware supports such as hardware multicast. In the unicast-based collective algorithms, the communication is scheduled over a sequence of stages. The union of these stages makes the desired collective operation. At each stage, a permutation of source-destination processes communicate with each other. The internal communication pattern of each collective operation depends on the permutation used in each stage by the collective algorithm. Note that a single collective operation might have different collective communication algorithms which results in different internal communication patterns.

There are various algorithms in literature for designing each collective operations. The MPI libraries usually use a combination of such algorithms and choose one of them based on some parameters such as message size and communicator size. The Fan-in/fan-out, binomial tree and ring algorithms are some examples of such algorithms. These algorithms are shown in Figure 2.4. In the ring algorithm [105], the data is sent around a virtual ring of processes as can be seen in Figure 2.4(a). In each step, process i sends its data to process $i + 1$ and receives a new data from process $i - 1$. The ring algorithm is used



(a) Ring (b) Binomial Tree

(c) Fan-in/fan-out

Figure 2.4: A few examples of algorithms for different collective operations

in MPI collective operations such as MPI_Allgather. Figure 2.4(b) shows the steps of the binomial tree algorithm [105]. In the first step, the data is sent from the root process to process $root + n/2$, in which n is the total number of processes. In the second step, both this and the root processes act as new roots in their own subtrees. The same procedure is done in subsequent steps of the algorithm, recursively. The binomial tree algorithm can be used in various collective operations such as MPI_Bcast, MPI_Gather, and MPI_Reduce. Figure 2.4(c) shows the fan-in/fan-out algorithms. In the fan-in algorithm [54], a single process receives data from all other processes. It is used for collective operations such as MPI_reduce and MPI_gather. In the fan-out algorithm, all processes read from a single

process. This algorithm can be used in collective operations such as MPI_Scatter and MPI_Bcast. The combined fan-in/fan-out algorithm can be used in collective operations such as MPI_Allreduce.

Some research studies [105, 96] discuss a comprehensive set of algorithms for various MPI collective operations (such as broadcast, allgather, alltoall, etc.). These algorithms are the basis for many other tuned collective communication algorithms proposed in the literature [69, 80, 87, 44, 90, 64].

2.2.4 One-sided Communications

In one-sided communication, also called Remote Memory Access (RMA), one process (origin) sends/ receives data to/from the address space of another process (target) without explicit participation of the remote process. In other words, the process can directly read or write from/into an exposed memory window of the other side. In one-sided communication, all the required communication parameters for both sending and receiving sides are provided at the origin process. We do not use one-sided communications in this dissertation. Therefore, we do not discuss it further.

2.2.5 MPI Message Queues

The MPI libraries typically maintain two queues, a *Posted Receive Queue* (PRQ) and an *Unexpected Message Queue* (UMQ) to deal with unavoidable out-of-sync communication in MPI. When a new message arrives, the PRQ must be traversed to locate the corresponding receive queue item, if any. If no matching is found, a Queue Element (QE) is enqueued in the UMQ. Similarly, when a receive call is made, the UMQ must be traversed to check if the requested message has already (unexpectedly) arrived. If no matching is found, a new QE is posted in the PRQ. Message queues are in the critical path of communication in MPI, and its search time affects the performance of applications that perform extensive communications.

In MPI message queues, the search is done based on the tuple <context_id, rank, tag>. Context_id is an integer value representing the communicator. The rank represents the source process rank in a PRQ request, and the receive process rank in an UMQ request. Tag is also an integer value that specifies the message id. The receiver can provide MPI_ANY_SOURCE or MPI_ANY_TAG instead of providing a specific source rank or tag. This is called wildcard communication and it means that the queue element can be matched with any sender process in the same communicator or any tag.

MPI implementations use different data structures to support message queue operations. MPICH and MVAPICH both use the linked list data structure depicted in Figure 2.5. Based on this data structure, these libraries provide a straightforward implementation of the MPI message queue that searches linearly for the key tuple in $O(n_{\mathrm{q}})$ in which $\mathrm{n_q}$ is the number of elements in the queue. When $\mathrm{n_q}$ is small, the linear search is acceptable. However, at large-scale, traversing a long queue is computationally intensive and requires a large number of pointer operations (e.g., access and dereferencing). The advantage of the linked list data structure is that the QEs are saved in the order of their arrival, and this conforms with the MPI point-to-point ordering semantic.

In order to manage MPI_ANY_SOURCE and MPI_ANY_TAG in MPICH/MVAPICH, Besides tag, rank and context_id, three other elements called *mask_tag*, *mask_rank* and *mask_context_id* are also saved in PRQ. Whenever a wildcard receive operation is called, the UMQ is searched. If the element is not there, it is saved in PRQ. The corresponding mask bits of the wildcard element is also set to zero. For example, if the wildcard operation was MPI_ANY_TAG, mask_tag would be set to zero and if it was MPI_ANY_SOURCE, mask_rank would be set to zero. Otherwise, if there was no wildcard operation, the mask bits are all set to 1. At the time of searching the PRQ, the searching element and the queue elements are masked. This way, we can skip searching the tag or the rank if the queue element was MPI_ANY_TAG or MPI_ANY_SOURCE, respectively.

Figure 2.6 shows message queue data structure in Open MPI. As can be seen in this

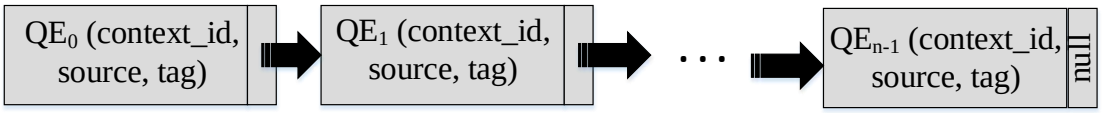


Figure 2.5: Linked list data structure in MPICH/MVAPICH

figure, Open MPI makes the search hierarchical by considering context_id as the first level and source as the second level. Each context_id, associated with a communicator of size n, has an array of size n representing the source ranks. Within each source rank, there is a linked list of tags. In this approach, requests bearing rank i are positioned in the ith element of the array. The advantage of Open MPI message queue data structure is that its queue search time is faster than the linked list data structure. However, contrary to the linked list, it has a large memory footprint as it requires an array of size n for each communicator of size n. Moreover, fragmenting the queues based on the communicator has limited benefit in practice since there are not many applications that use multiple communicators (it might not be the case in future though).

Due to the message queue traversal overhead in the MPI applications that generate long message queues, using a suitable matching algorithm is of great importance. In Chapters 3-6 of this dissertation, we propose new message queue mechanisms to improve the performance of message queue operations. The proposed designs consider the properties of the new HPC systems as well as the features of the HPC applications.

2.2.6 MPI Topology Interface

MPI provides this opportunity to define a logical topology for the processes of an application. This information, which is referred to as process topology or *virtual topology* in MPI, is attached as an additional and optional attribute to a corresponding communicator. Virtual topology provides this opportunity to have a convenient naming mechanism for the set of processes in a communicator. Moreover, it can be used to gain information about the communication pattern of the processes. This information can be leveraged in many

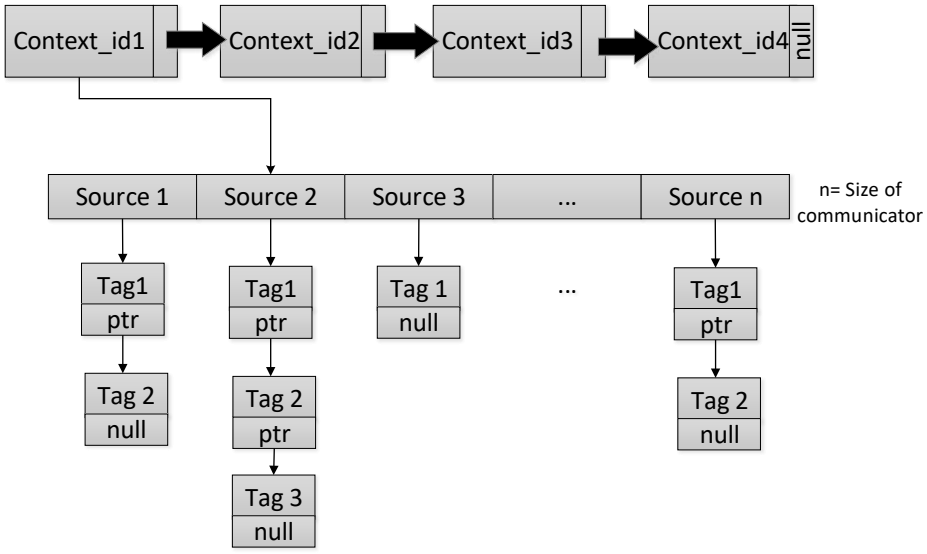


Figure 2.6: Open MPI message queue data structure

different ways to improve the performance. For example, the virtual topology information can be used to conduct topology-aware mapping optimization [91].

Process topologies is described using two main interfaces: (1) the *graph* interface, and (2) the *Cartesian* interface. In the graph interface, each process is represented as a vertex of the graph and the communication relationship among the processes is described as the edges. The MPI-1.0 standard (released in 1994) defines a general graph constructor, MPI_Graph_create(), that can be used to build a directed and unweighted topology graph. As discussed in [23, 29, 107], this interface is not scalable so it cannot be used in practice. To deal with this issue, the MPI-2.2 standard (released in 2009) provides a more scalable and informative topology interface which is defined as *distributed graph topology* functions. In this interface, each process describes only a fraction of the whole communication graph. In other words, the graph topology is distributed between the processes. It also provides this opportunity to assign relative *weights* to the communication edges. Moreover, an additional *info* argument can be passed to the interface. This argument provides the user with more control over further optimizations related to process topologies.

Two distributed graph constructors are defined in MPI-2.2: MPI_Dist_graph_ create_adjacent() and MPI_Dist_graph_create(). In MPI_Dist_graph_create_ adjacent(), each process only specifies its own outgoing and incoming neighbors. On the other hand, MPI_Dist_graph_create() can specify an arbitrary set of edges that may or may not be its own neighbors. The advantage of the adjacent specification is that the neighborhood information is already available locally at each process, while in the non-adjacent specification, processes should communicate with each other to extract neighborhood information. On the other hand, the disadvantage of the adjacent specification is that it supplies each edge twice; once by each of its endpoints.

Although the graph topology functions can be used to describe any virtual process topology, the Cartesian interface provides easier and more efficient way to describe certain process topologies such as n-dimensional grid-based mesh/torus topologies that can be entirely defined by the number of dimensions and the number of processes along each dimension. Accordingly, MPI provides a number of functions that allow for creation and manipulation of Cartesian topologies. For example, the function MPI_Cart_create() creates a new communicator with a Cartesian topology of a desired number of dimensions. The drawback of the Cartesian topology is that it only specifies communications among immediate neighbor processes. For example, it cannot be used to describe communications between diagonal neighbors. Also, it does not support weighted edges.

2.2.7 Neigborhood Collective Communications

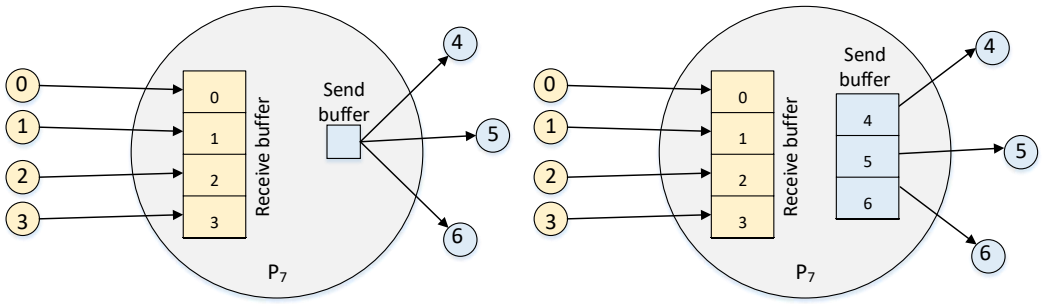
Neighborhood collectives are introduced in the MPI-3.0 standard (released in 2012) to add communication functions to the process topologies. They are similar to conventional collective communications in a sense that they provide an abstraction for communications among a group of processes. However, the problem with conventional collectives is that they are not inherently scalable. That is because of the global nature of their communications that encompasses all the processes in a communicator. This concern becomes an issue at the

exascale level where some collectives (e.g., all-to-all) are too costly to be practical.

To address these issues, neighborhood collectives can be used. In neighborhood collectives, each process only communicates with the processes that are defined as one of its outgoing/incoming neighbors. The neighbors are specified using the communication pattern derived from the topology graph of the processes. Thus, unlike conventional collectives, neighborhood collectives allow users to define their own communication patterns through the process topology interface of MPI. In this sense, neighborhood collectives vastly extend the concept of collective communications in MPI and represent one of the most advanced features of the standard. Moreover, neighborhood collectives are more scalable since they restrict communications to a local neighborhood of each process. Another advantage of neighborhood collectives is that they support *sparse* communication patters [61] used in many applications such as Nek5000 [45], Qbox [56] and octopus [35]. Although point-to-point operations can be used to implement sparse collective communications, the advantage of neighborhood collectives is that they use the information provided by virtual topology to implement such communication pattern more efficiently. Other advantages of using neighborhood collectives include performance portability and higher levels of readability and maintainability of the application code.

Currently, two main neighborhood collective operations are defined in the standard: MPI_Neighbor_allgather() and MPI_Neighbor_alltoall(). In a neighbor allgather operation, each process sends its data to each of its outgoing neighbors designated by the process topology graph, and receives the data from each of its incoming neighbors. The neighbor alltoall operation has the same communication pattern. The difference is that in neighbor allgather the same message is sent to all outgoing neighbors of a process, whereas in neighbor alltoall, a different message is sent to each outgoing neighbor of a process. Figure 2.7 shows an example of process topology with status of buffers for neighbor allgather and neighbor alltoall operations from the viewpoint of process P_7.

Variations of neighborhood collective operations include MPI_Neighbor_allgatherv(),



(a) Neighbor Allgather (b) Neighbor Alltoall

Figure 2.7: An example of process topology with the status of buffers for neighbor allgather and neighbor alltoall operations from the viewpoint of process P_7

MPI_ Neighbor_alltoallv() and MPI_Neighbor_alltoallw(). MPI_Neighbor_allgatherv(), MPI_ Neighbor_alltoallv() extend the functionality of MPI_Neighbor_allgather() and MPI_ Neighbor_alltoall() by allowing a varying count of data from each process. MPI_ Neighbor_alltoallw() allows different datatypes, counts and displacement for each partner.

2.3 InfiniBand: A High Performance Interconnect

InfiniBand (IB) [4] and Gigabit/10Gigabit Ethernet are two main cluster interconnection network technologies in HPC domain. At the time of writing this document, InfiniBand is being used in around 27% of top500 supercomputers in the world. Gigabit/10Gigabit Ethernet are other popular interconnects that have 50.4% share. However, the performance share of InfiniBand is 36.1% compared to 20.7% of Gigabit/10Gigabit Ethernet.

InfiniBand Architecture (IBA) was developed in 1999 by a group of around 180 companies called InfiniBand Trade Association (IBTA). They successfully standardized InfiniBand in October 2000. InfiniBand provides high performance, low latency and efficient communication technology. Unlike traditional networks, InfiniBand does not require Operating System (OS) for transferring the messages. In a traditional network, the TCP/IP protocol stack and the Network Interface Card (NIC) are owned by the operating system. Therefore, applications rely on the OS for message transfer. However, InfiniBand transfers the message

on its own without going through the OS. Switches, routers, Channel Adapters (CAs), cable and connectors are the main components of InfiniBand. CAs can be either Host Channel Adapters (HCA) that connect the host node to IB network or Target Channel Adapters that connect external I/O devices to IB networks. IBA defines two communication semantics: channel semantic and memory semantic. In the channel semantic, both sender and receiver take part in communication. However, in the memory semantic, also called *Remote Direct Memory Address* (RDMA), only the node that initiates the communication is responsible for data communication, and it reads/writes from/to the virtual address of the remote peer.

Chapter 3

Clustering-based Message Matching Mechanism

Message queues are in the critical path of communication in MPI, and its search time affects the performance of applications that perform extensive communications. Due to the message queue traversal overhead in the MPI applications that generate long message queues, using a suitable matching algorithm is of great importance. AMG2006 [114], LAMMPS molecular dynamics simulator [97] and Fire Dynamic Simulator (FDS) [89] are some examples of the applications that generate long message queues.

Although there are various message matching designs used in MPI implementations or proposed in literature [116, 46, 27, 71], non of them takes advantage of clustering mechanisms to improve message queue operations. In this chapter, we propose a mechanism that considers the impact of communication by source/receive processes on message queue length, and uses this information to cluster the processes into different groups [48]. Processes are clustered based on the number of queue elements they add to the PRQ or UMQ. We then allocate a dedicated linked list to each cluster. We study the benefit of the clustering-based approach with two different clustering algorithms. The first algorithm is based on K-means clustering [102] while the second one is a heuristic designed to uniformly assign processes to the clusters based on the number of queue elements they add to the queues.

The advantage of the proposed approach is twofold. First, it will reduce the average number of queue traversals due to the use of multiple queues instead of a single queue.

Secondly, we can further improve the queue search operation by using a proper clustering algorithm in a way that the average number of traversals for each cluster is minimized. In our study with three real applications, LAMMPS, AMG2006 and FDS, on a large-scale cluster, we show that the proposed approach decreases the average queue traversals as well as the queue search time and ultimately improving the application performance.

The rest of this chapter is organized as follows: Section 3.1 presents the related work and distinguishes our work from them. Section 3.2 discusses the motivation behind this work. Section 3.3 describes our message queue approach. The experimental results and analysis are presented in Section 3.4. Finally, we conclude the chapter in Section 3.5.

3.1 Related Work

Several works have studied the impact of the message queue features on the performance of different MPI applications [70, 30, 32, 33]. Keller and Graham [70] show that the characteristics of UMQ, such as the size of UMQ, the required time for searching the UMQ and the length of time such messages spend in these queues, have considerable impact on scalability of MPI applications (GTC, LSMS and S3D). Researches in [30, 32, 33] focus on evaluating the latency and message queue length of some applications. A more recent work [79] proposes a validated simulation-based approach to examine how the performance of several important HPC workloads is affected by the time required for matching. It also provides some guidance on the design and development of MPI message queues.

There have been various works on improving the message queue operations in MPI [116, 46, 27, 71]. The main idea behind most of these approaches is to improve the queue search time by reducing the number of queue traversals. This can be done by taking advantage of multidimensional queues [116], hash tables [46] or parallelizing the search operation [71]. Zounmevo and Afsahi [116] proposed a 4-dimensional data structure to decompose ranks into multiple dimensions. The aim of this data structure is to skip searching a large portion

of the data structure for which the search is guaranteed to yield no result. The disadvantage of this work is that it has some overhead for short list traversals. Moreover, it does not consider the application's characteristics in the design.

Flajslik, et al. [46] use a hash function to assign each matching element to a specific queue. The hash function is based on a full set of matching criteria (context_id, rank and tag). This data structure consists of multiple bins with a linked list within each bin. The number of bins and the hash function can be set at configuration time. The evaluation results show that by increasing the number of bins, the number of traversals and consequently the queue search time would be reduced for long list traversals. The disadvantage of this work is that it does not consider the application characteristics and it has some overhead for short list traversals.

In [27], the authors propose a message matching design that dynamically selects between one of the existing approaches: a defaultt linked list, a bin-based design [46] and a rank-based design in which each source process has its own dedicated message queue. This work starts running the application with the default linked list design. During the application runtime, the number of traversals for each process is measured. If the number of traversals is more than a threshold, they switch to the bin-based design. They can also switch to the rank-based design if it is specified by the user at configuration time. The disadvantage of this work is that it does not consider the application characteristics to allocate the elements to the queues. The clustering-based approach differs from this work in that it profiles the communication traffic between each individual peers and takes advantage of clustering mechanism to improve message matching performance.

Other researches investigate MPI matching list improvement in hardware [71, 111, 26]. Klenk, et al. [71] propose a new message queue design to take advantage of the availability of large number of threads on GPUs. This message matching algorithm searches the queue in two phases (scan and reduce) and provides 10 to 80 times improvement in matching rate. The disadvantage of this work is that it cannot handle more than 1024 elements in the

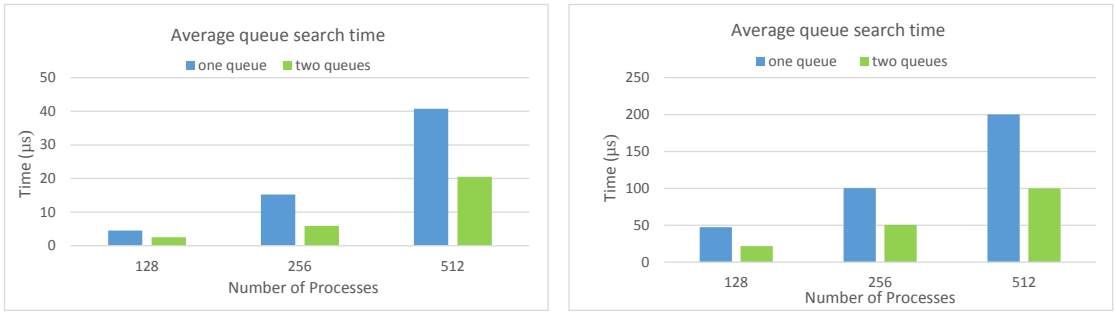
queue. Underwood, et al. [111] accelerate the processing of UMQ and PRQ by offloading the MPI message matching to specialized hardware. By increasing the scale of the current systems, offloading the MPI message matching is gaining attention once again to drive down the MPI messaging latency. The Portals networking API [26] enables such offloads. Several current low-level networking APIs also provide interfaces that include support for message matching [7, 13] to enable the use of specialized hardware and software techniques.

Our work improves the message matching operations in MPI by reducing the number of queue search traversals. To do this, it takes advantage of multiple queues along with clustering. The difference between the proposed work in this chapter and other approaches is that our approach clusters the queue elements wisely, based on the number of queue elements each source adds to the message queues. This way, we can achieve high message matching performance by allocating a small number of queues rather that allocating a large number of queues.

3.2 Motivation

Placing all incoming unexpected messages into the same UMQ, or all early posted receives into the same PRQ, will result in long message queues and traversals when looking for a matching element. Previous research [46] has shown that increasing the number of message queues will potentially increase the performance. This is also used in Open MPI as an extreme case, where there is a dedicated UMQ allocated for each potential source process. The work in [46] naively places incoming messages and posted receives into message queues based on a hashing function without considering the application characteristics. In this chapter, we also use multiple queues to speed up the operation, however we extend our intuition a step further by considering the behavior of the applications in managing parallel message queues in MPI.

Through microbenchmarking, we provide the evidence that increasing the number of



(a) M=10 (b) M=50

Figure 3.1: Average UMQ search time for the reverse search on Cluster A

message queues and clustering groups of processes together and assigning them a dedicated message queue based on their communication intensity will increase the message queue performance. In our *reverse search* microbenchmark, processes P_1 to P_{n-1} (n is the total number of processes) first send m data each to process P_0. Then, the elements are dequeued from the bottom of the queue by P_0. We measure the time P_0 spends in searching each one of the elements in the queue coming from the other processes and report the average queue search time across all incoming messages. We double the number of message queues to two and compare its performance for the reverse search against the default MVAPICH2 implementation that uses a single UMQ. In the two-queue case, we place half of the incoming messages at the end of the queue in the second queue. Our objective in this test is to confirm that increasing the number of message queues indeed improves the performance.

Figure 3.1 shows the results for the reverse search that has been conducted on Cluster A described in Section 3.4.1. As can be seen in the figure, by doubling the number of queues, we can decrease the average queue search time. This improvement is more significant when the number of pending messages per process is larger.

In our second microbenchmark, we revise our previous microbenchmark in a way that the processes P_1 to $P_{\text{n-1}}$ are now grouped into four groups, P_{s}, P_{m}, P_{l} and P_{vl} where each process within a group sends a small (2 messages), medium (10 messages), large (50

Table 3.1: Different combination of groups of processes assigned to each queue

*Clustering*1	Queue 1: P_s & P_m	Queue 2: P_l & P_{vl}
*Clustering*2	Queue 1: P_s & P_l	Queue 2: P_m & P_{vl}
*Clustering*3	Queue 1: P_s & P_{vl}	Queue 2: P_m & P_l
*Clustering*4	Queue 1: P_s & P_m & P_l	Queue 2: P_{vl}
*Clustering*5	Queue 1: P_m & P_l & P_{vl}	Queue 2: P_s
*Clustering*6	Queue 1: P_s & P_m & P_{vl}	Queue 2: P_l
*Clustering*7	Queue 1: P_s & P_l & P_{vl}	Queue 2: P_m

messages), or very large (100 messages) number of messages to process P_0, respectively. We then forward a combination of these messages, such as those from the small and large groups, to one of the two queues for P_0. The remaining (e.g., medium and very large) messages will be sent to the other queue. Table 3.1 shows the different combination of groups of processes that are assigned to each of the two queues for P_0.

Our objective in this experiment is to understand the impact of clustering on multiple message queues and compare its performance against a single queue as well as a four-queue case where the messages from each group are sent to a distinct queue. Figure 3.2 compares the average UMQ search time for our microbenchmark with different clustering of processes shown in Table 3.1 with a single-queue case and a 4-queue case, respectively.

Two conclusions can be derived from these results. First, as shown in the previous microbenchmark results in Figure 3.1, increasing the number of queues improves the queue search time. The second and more important observation from this experiment is that the way we assign the queues to processes has a considerable impact on the queue search time. In other words, if we assign the processes to queues appropriately (Clustering 3 and 4), the two-queue case can be as good as the case with four queues. However, assigning the processes to queues incorrectly would negate the impact of increasing the number of message queues and may make it as worse as the case with a single queue (Clustering 5).

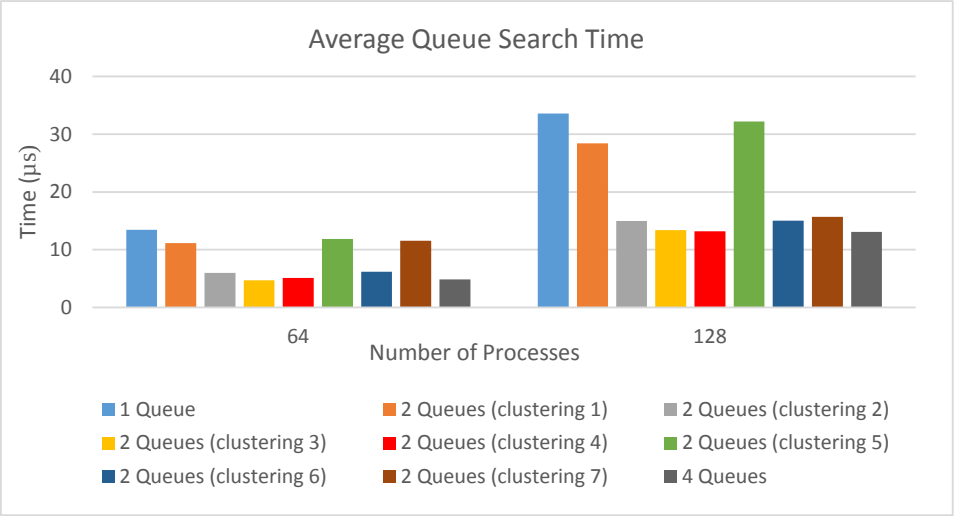


Figure 3.2: Average UMQ search time for the reverse search on Cluster A

3.3 The Proposed Clustering-based Message Matching Design

The motivational results showed that adding a second message queue can reduce the queue search time significantly. In addition, it became possible to reduce the queue search time further by clustering the processes into different groups based on their communication frequency and allocating a dedicated message queue to each group. Motivated by these results, we propose a new message matching structure in MPI with multiple queues in order to support message passing more efficiently based on the communication characteristics of the MPI applications at runtime. The overall process is as follows:

(a) Run the MPI application once to gather information about the number of elements processes add to the queue at runtime.

(b) Use this information to perform a cluster analysis of the MPI application processes. Here, we use two different clustering algorithms: K-means clustering algorithm [102] and a heuristic algorithm.

(c) Redesign the MPI library to have a dedicated queue for each cluster of processes.

(d) Use the clustering information obtained in Step (b) to enqueue the incoming messages or receive calls from the MPI application to their dedicated MPI message queues in the next run.

Figure 3.3 shows the first phase of gathering information and clustering in the clustering-based message queue mechanism for the unexpected message queue. As the figure shows, we first derive the *message queue feature* matrix representing the total number of messages coming to the queues for peer processes. In this matrix, each row, representing a process i, has an array of size n, where n is the number of processes. The element $A_{\mathrm{i,j}}$ indicates the total number of messages that was added to the queue from the source process j to the destination process i. This array is then used to classify the processes into groups or clusters. Any clustering algorithm can be used here for the classification. We discuss two different algorithms. The first algorithm is K-means clustering algorithm That is discussed in Appendix I. We used the Euclidean distance between the data points and the clustering points in K-means clustering. The second algorithm uses a heuristic-based algorithm discussed in Section 3.3.1. The output of the clustering phase is a *clustering identifier* matrix file with elements indicating the cluster number for source-destination pairs.

Figure 3.4 shows how the clustering information gathered in the first phase is used to build the message queues in future runs. The clustering identifier matrix is given as an input to the message queue identifier. Each destination process i in the clustering identifier matrix has an array of size n. For the peer destination process i, the element $B_{\mathrm{i,j}}$ indicates the cluster in which the source rank j belongs to. In this design, each cluster represents one of the queues. When a message arrives (or a receive call is issued), the message queue identifier retrieves the cluster number from the clustering identifier matrix and enqueues it into the corresponding queue, if it cannot be matched with posted receives in PRQ (or unexpected messages in the UMQ).

The new MPI message matching design provides two main advantages. First, leveraging

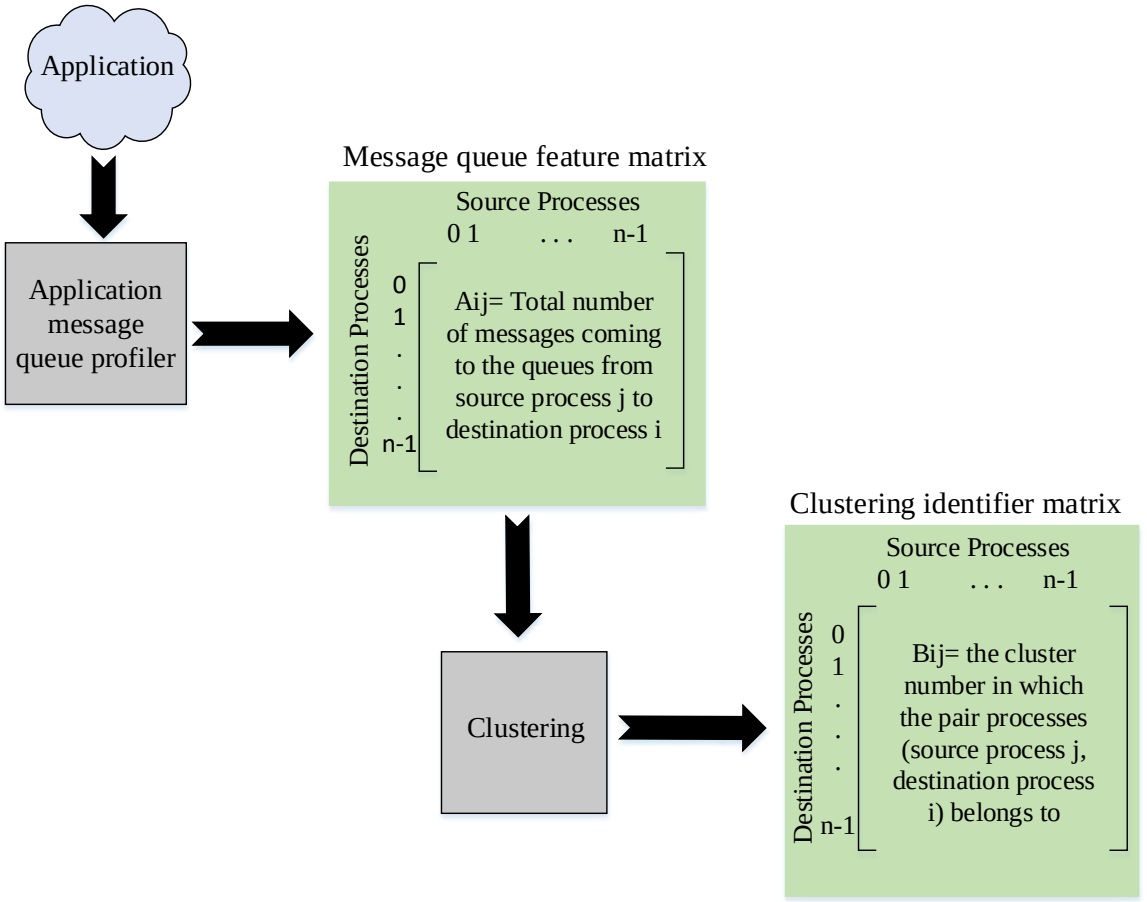


Figure 3.3: The first phase of gathering information and clustering in the proposed message matching mechanism

multiple queues can reduce the number of traversals and consequently, the queue search time. Secondly, clustering the processes based on the queue behavior of the application and assigning a dedicated message queue to each cluster provides the opportunity to search the queues faster and consequently reduce the queue search time and application runtime.

Note that the MPI_ANY_TAG wildcard is automatically supported in the proposed message queue structure, as elements from a source is stored in the same linked list in the order of their arrival. To deal with MPI_ANY_SOURCE wildcard communication, a sequence number is added to each queue element. Requests bearing MPI_ANY_SOURCE are

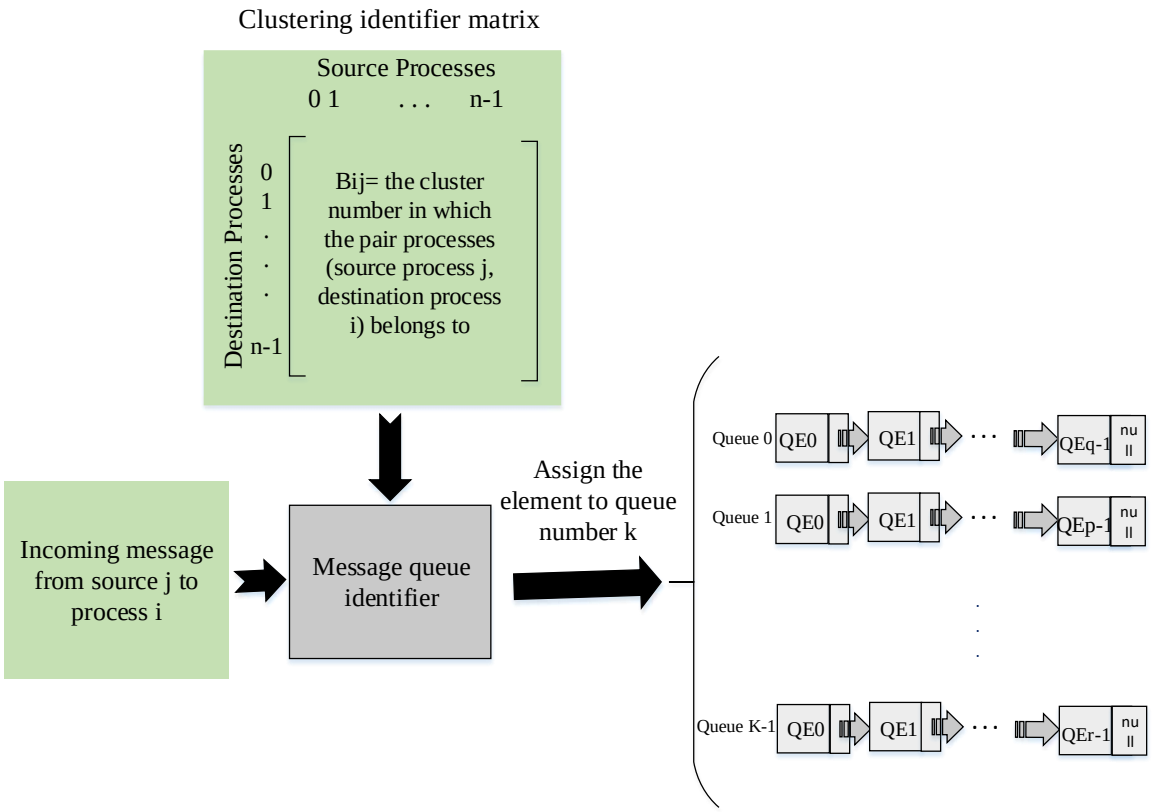


Figure 3.4: Clustering-based message queue structure

allocated to a separate PRQ called PRQ_ANY. When searching rank j in the posted receive queue is required, both the posted receive queue k corresponding to rank j (derived from the K-means clustering or heuristic-based algorithm) and the PRQ_ANY queue are searched. If there are multiple matches, the element whose sequence number is smaller is chosen as the matching element. Similarly, when searching UMQ with MPI_ANY_SOURCE, all the queues k $(0 \leq k < K)$ will be searched, and the element with the smallest sequence number will be selected as the matching element. Note that the user can provide a hint to the runtime library to disable the search mechanism for MPI_ANY_SOURCE if the application does not use any wildcard communication.

3.3.1 Heuristic-based Clustering Algorithm

The problem with K-means clustering is that the queue elements from different processes are not uniformly distributed in the clusters. This makes some queues to be overloaded while others do not have much elements. Moreover, K-means clustering does not recognize the number of clusters k. Therefore, we propose a heuristic-based clustering algorithm to deal with these issues. This algorithm assigns $average+SD$[1] queue elements to each cluster. The term average makes the heuristic-based design to assign at least an $average$ number of elements to each cluster. We also add standard deviation to the average to avoid allocating a dedicated queue for a significant number of processes whose number of queue elements is more than average. We should note that in a normal distribution, 50% of data points are greater than average and only 16% of data points are greater than $average + SD$.

The heuristic-based algorithm is presented in Algorithm 3.1. In this algorithm, first we calculate the average and standard deviation for each row of message queue identifier matrix (Lines 1 and 2). Then, we sort the input array in Line 3. By sorting the elements, we can assure that each cluster has the maximum possible number of processes. In Lines 6 to 12 of the algorithm, we start from the first process in the sorted array and add the processes into the first cluster until the total sum of all messages from these processes is less than $average + SD$. If it becomes greater than this value, the number of clusters is incremented and a new set of processes is added to the second cluster. This trend is continued until all processes are allocated to the clusters. This way, on average we have $average + SD$ elements in each cluster.

The advantage of the heuristic algorithm is that it assigns the processes to the clusters in a way that the communication frequency is almost the same in all clusters. Moreover, this algorithm assures that the total number of queue elements in each cluster will not exceed $average + SD$. Another advantage of this algorithm compared to the K-means algorithm

[1] Standard Deviation

is that it identifies the number of clusters based on the input array without any hint from the user. We should note that the overhead of the clustering algorithm (either K-means clustering or heuristic-based algorithm) occurs only once in the first run.

Algorithm 3.1: Heuristic algorithm for clustering the processes

Input : Total number of elements sent to UMQ/PRQ from each process Num-of-$QE_{0...P}$, Number of processes P

Output: An array containing the cluster number for peer process i and j cluster-$num_{0...P}$, Number of clusters total-cluster-QE

```
1  E=average of Num-of-QE_{0...P};
2  SD=standard deviation of Num-of-QE_{0...P};
3  Sort Num-of-QE_{0...P};
4  k=0;
5  total-cluster-QE=0;
6  for i ∈ 1...P do
7  |  cluster-num_i=k;
8  |  total-cluster-QE+ =Num-of-QE_i
9  |  if total-cluster-QE > (E + SD) then
10 |  |  k++;
11 |  |  total-cluster-QE=0;
12 |  end
13 end
```

3.4 Performance Results and Analysis

This section studies the performance of the proposed clustering-based message queue structure against MVAPICH2 default queue data structure. We present the results for average number of queue traversals, average queue search time, as well as the execution time for three applications. We should note that we run the applications under the same condition (same workload and number of processes) in the first and subsequent runs.

3.4.1 Experimental Platform

The experimental study is done on the General Purpose Cluster (GPC) at the SciNet HPC Consortium of Compute Canada. GPC consists of 3780 nodes, for a total of 30240 cores.

Each node has two quad-core Intel Xeon sockets operating at 2.53GHz, and a 16GB memory. We have used the QDR InfiniBand network of the GPC cluster. The MPI implementation is MVAPICH2-2.0. We refer to GPC as Cluster A in this dissertation.

The applications that we use for the experiments are AMG2006 [114], LAMMPS [97] and FDS [89]. We consider these three applications due to their different message queue behavior. For example, FDS has long list traversals especially at process 0. On the other hand, AMG2006 and LAMMPS are well-designed applications that have short and medium message queue traversals. We present the results for these three applications to show the efficiency of the proposed approach for well-crafted applications with short and medium traversals as well as applications with considerably large number of traversals.

AMG2006 is a parallel algebraic multi-grid solver for linear systems arising from problems on unstructured grids. It uses data decomposition to parallelize the code in MPI. LAMMPS is a classical molecular dynamics code, and an acronym for Large-scale Atomic/ Molecular Massively Parallel Simulator. It has potentials for solid-state materials (metals, semiconductors), soft matter (biomolecules, polymers) and coarse-grained or mesoscopic systems. It can be used to model atoms or, more generically, as a parallel particle simulator at the atomic, meso, or continuum scale. We run rhodopsin protein benchmark in our experiments. This benchmark uses particle-mesh method to calculate long range forces [98]. FDS is a computational fluid dynamics model of fire-driven fluid flow. It solves numerically a form of the Navier-Stokes equations appropriate for low-speed, thermally-driven flow with an emphasis on smoke and heat transport from fires.

3.4.2 Message Queue Traversals

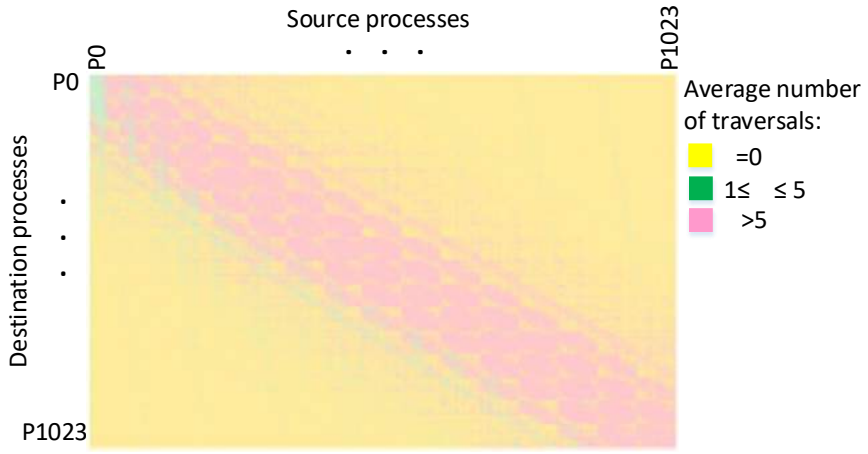
We measure the average number of traversals to find the desired element in UMQ for AMG2006 and LAMMPS applications. Figure 3.5 and Figure 3.6 show the average number of traversals for different sources for each process in a heat map in the proposed approach and the default MVAPICH implementation for the AMG2006 and LAMMPS, respectively.

The horizontal axis shows the source processes and the vertical axis shows the destination processes. The red data points depict high number of traversals while the green and yellow data points show medium and low number of traversals, respectively. The experiments were done on Cluster A.

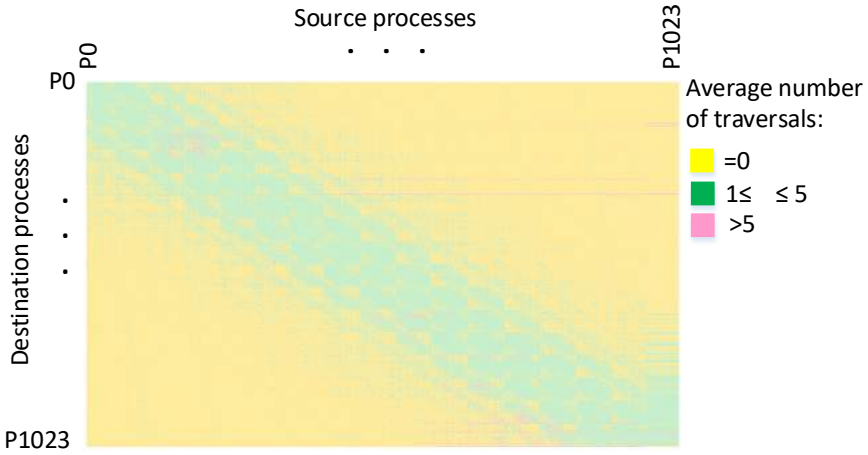
Figure 3.5 shows the average number of traversals for UMQ in AMG2006 application in default MVAPICH implementation (Figure 3.5(a)), the clustering-based approach with K-means algorithm when k=32 (Figure 3.5(b)) and the clustering-based approach with heuristic-based algorithm (Figure 3.5(c)) for 1024 processes. It is obvious from the figures that both algorithms in the clustering-based approach can reduce the number of traversals considerably, effectively reducing the queue search time of the application. Comparing Figure 3.5(b) with Figure 3.5(c), we can observe that the number of traversals in heuristic algorithm is less than that of K-means algorithm.

We ran the same experiment for the LAMMPS application. Figure 3.6 show the average number of traversals in default MVAPICH implementation and in our approaches for 240 processes. Note that LAMMPS generates longer queues with 240 processes than 1024 processes. Therefore, we show the results with 240 processes. Comparing Figure 3.6(a) with Figure 3.6(b) and 3.6(c), it is obvious that the clustering-based approach (with both K-means and heuristic algorithm) can reduce the number of traversals to less than 10 for almost all peer processes. This reduction is more significant in heuristic algorithm.

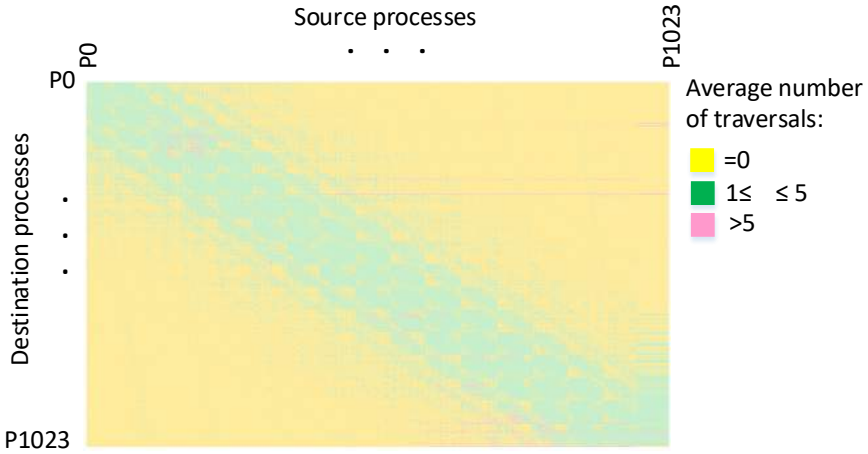
We should note that in the FDS application, process 0 is the only process that has a significant number of communications; the number of traversals in other processes is negligible. Using a figure similar to Figures 3.5 and 3.6 to depict the number of traversals in FDS would not provide much information since the number of traversals in all the rows except the first row would be very small in both MVAPICH and clustering-based approach. In other words, in both approaches we would have a figure which is almost yellow except in the first row. Therefore, we do not show the message queue traversals for FDS. Rather, we will show the performance gain of the FDS application by presenting its queue search time



(a) default MVAICH

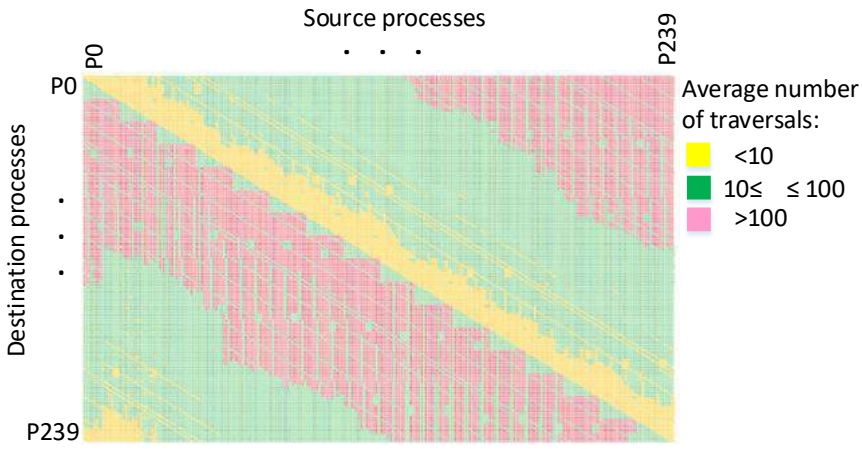


(b) clustering-based (K-means with $k = 32$)

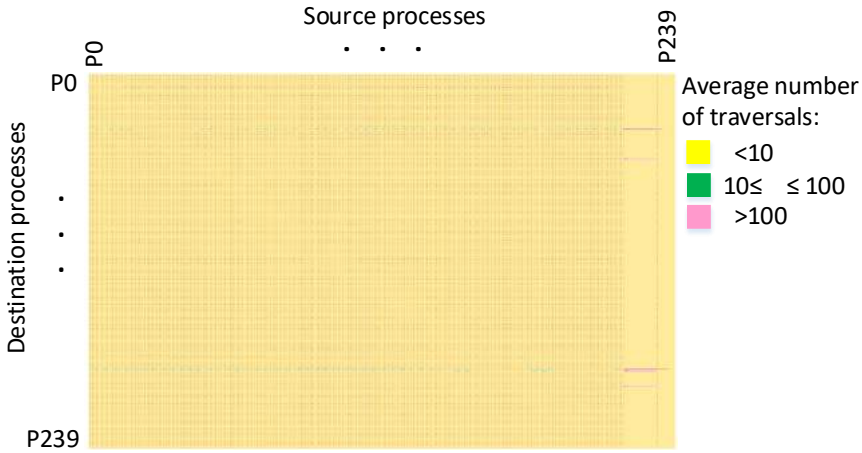


(c) clustering-based (heuristic algorithm)

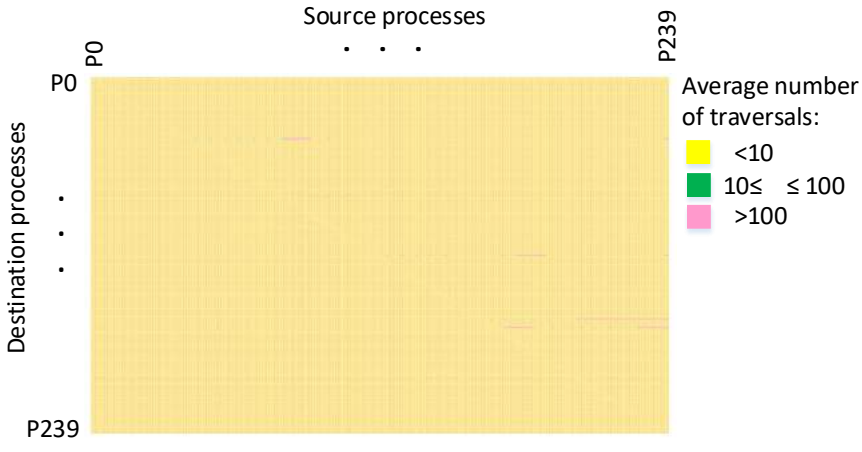
Figure 3.5: Average number of traversals for AMG2006 in different approaches on Cluster A with 1k processes



(a) default MVAICH

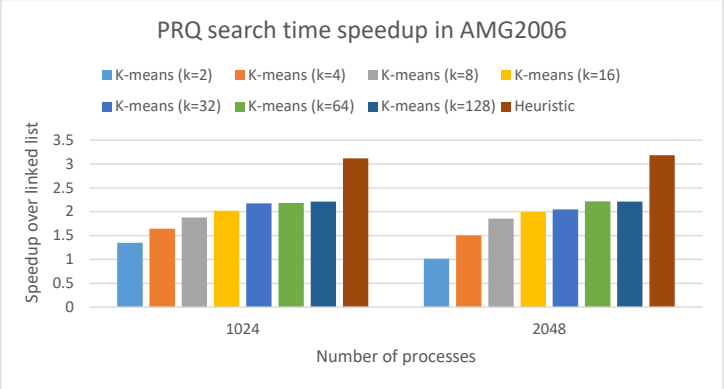


(b) clustering-based (K-means with $k = 32$)

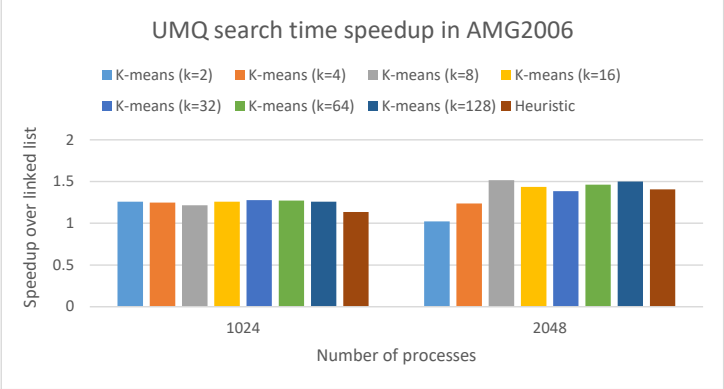


(c) clustering-based (heuristic algorithm)

Figure 3.6: Average number of traversals for LAMMPS in different approaches on Cluster A with 240 processes



(a) PRQ



(b) UMQ

Figure 3.7: Average PRQ and UMQ search time in the AMG2006 application on Cluster A

and application runtime in Section 3.4.3 and Section 3.4.4, respectively.

3.4.3 Queue Search Time

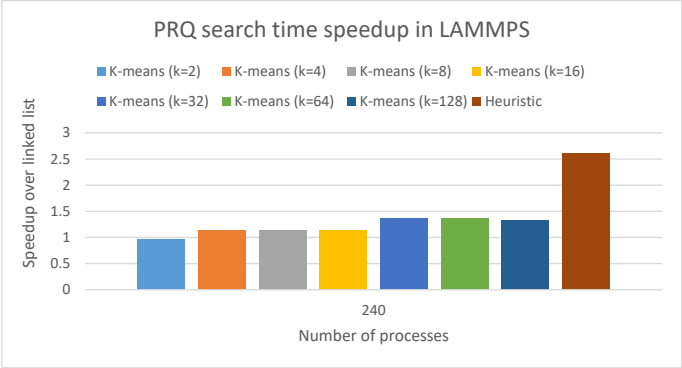
In this section, we compare the PRQ and UMQ search time in linked list data structure used in MVAPICH with the clustering-based approach for AMG2006, LAMMPS and FDS applications. Figure 3.7 through Figure 3.10 show the PRQ and UMQ search time speedup over the linked list for these applications, respectively. We report the results for both K-means and heuristic algorithms.

Figure 3.7 shows the PRQ and UMQ search time for AMG2006. One observation from

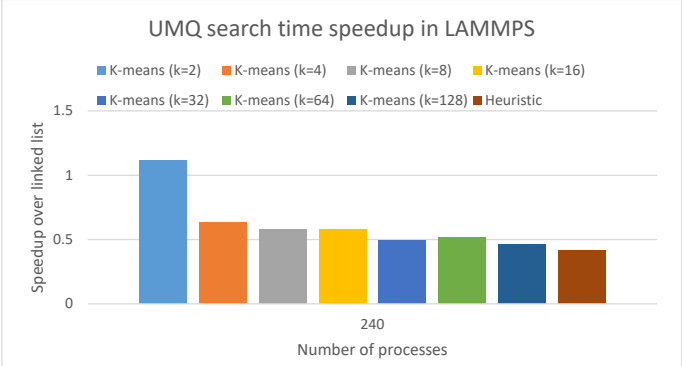
this figure is that the improvement in PRQ search time is more significant compared to UMQ. That is because of the long list traversal of PRQ compared to UMQ. The short list traversal of UMQ in AMG2006 provides no room for performance improvement and the speedup is around 1 in almost all cases. For PRQ, the heuristic algorithm provides a better performance than the K-means algorithm. The reason for this is that in the heuristic algorithm the processes are assigned to the clusters uniformly based on their communication frequency, as discussed in Section 3.3.1. We should note that the average number of clusters derived from the heuristic algorithm is 183 and 236 for 1024 and 2048 processes, respectively. In regard to the K-means results, the average PRQ search time is decreased when the number of clusters is increased from 2 to 16. However, increasing the number of clusters further does not improve the PRQ search time significantly. In other words, the best performance can be roughly achieved with 16 clusters/queues in the K-means algorithm.

Figure 3.8 shows the PRQ and UMQ search time improvement for the LAMMPS application. As can be seen in this figure, the UMQ search time does not have performance improvement with clustering-based design. This is because of the short list traversals of UMQ in LAMMPS that does not compensate the overhead of retrieving the queue number in clustering-based design. On the other hand, PRQ has longer list traversals so its queue search time improvement is more significant with clustering-based design (up to 2.6x). Here, the heuristic-based algorithm performs better than the K-means algorithm. The average number of clusters generated in the heuristic algorithm is 81.

Figure 3.9 shows the PRQ and UMQ search time for FDS. In this application, the majority of communications is done with process 0. Therefore, we show the PRQ and UMQ search time for process 0. Clearly, the heuristic-based algorithm performs better than the K-means algorithm (44x compared to 2x). We enlarge Figure 3.9 in Figure 3.10 to show the results for different k values in the K-means algorithm. For this algorithm, the best performance can be achieved when the number of queues is 8 or 16 for PRQ (the speedup is around 2). For UMQ, the best performance is achieved with k=128 (the speedup is around



(a) PRQ



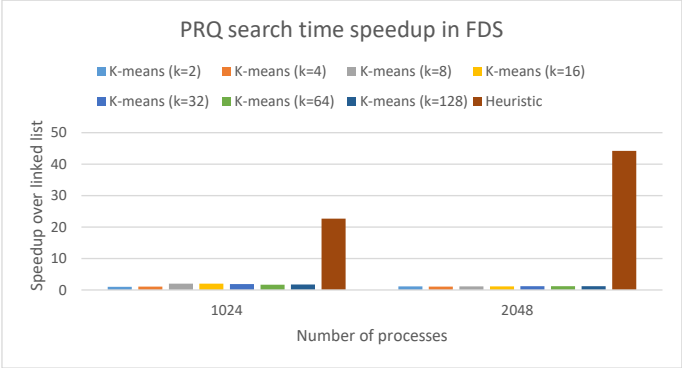
(b) UMQ

Figure 3.8: Average PRQ and UMQ search time in the LAMMPS application on Cluster A

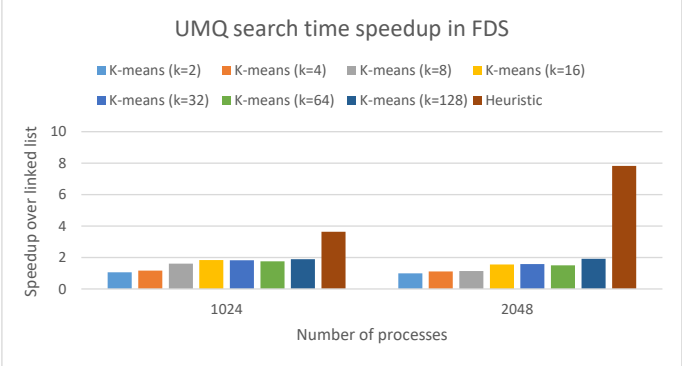
2). In the heuristic algorithm, the average number of clusters is 13 and 14 for 1024 and 2048 processes, respectively.

3.4.4 Application Runtime

In this section, we compare the performance of the AMG2006, LAMMPS and FDS applications in our approach against the linked list data structure in MVAPICH. The results in Figure 3.11 is in concert with the results shown in Figures 3.7, 3.8 and 3.9. We can observe that the improvements in queue search time directly translates to application performance. Figure 3.11 also shows that the heuristic-based algorithm provides a better performance compared to the K-means algorithm for FDS.



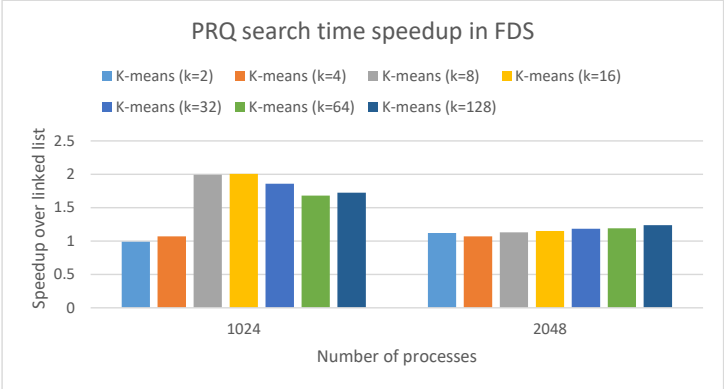
(a) PRQ



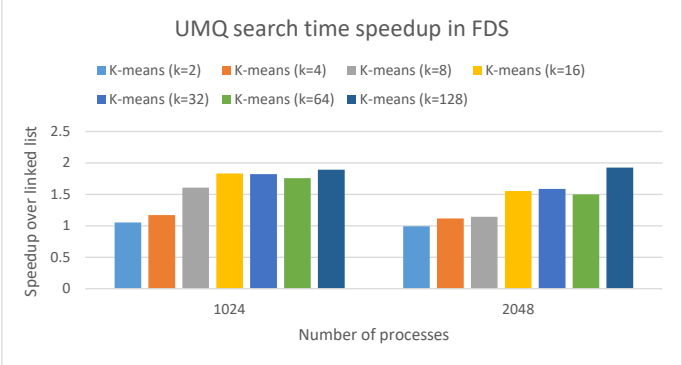
(b) UMQ

Figure 3.9: PRQ and UMQ search time for process 0 in the FDS application with K-means and heuristic algorithms on Cluster A

As can be seen in Figures 3.11(a) and 3.11(b), for AMG2006 and LAMMPS, the application runtime does not have considerable performance improvement (around 1.2x in AMG2006 and 1x in LAMMPS application). On the other hand, FDS performance is improved by 2.4x. The reason for insignificant performance improvement in LAMMPS and AMG2006 is the number of times the queues are searched in these applications. Table 3.2 shows the maximum number of times the UMQ and PRQ are searched in FDS, AMG2006 and LAMMPS applications, One can observe the sharp contrast between AMG2006 and LAMMPS with the FDS application. Reducing the average queue search time in FDS considerably impacts its execution time. However, the improvement is less in AMG2006 and



(a) PRQ



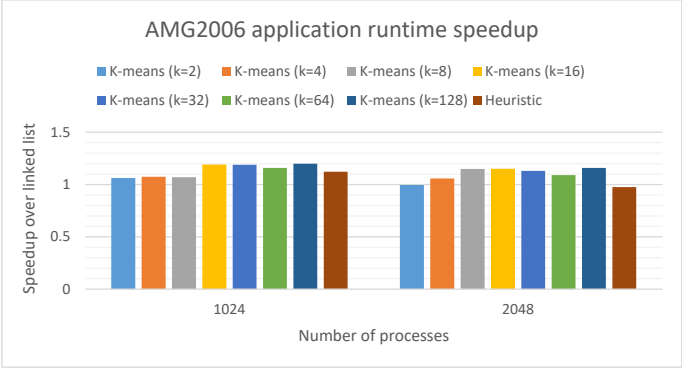
(b) UMQ

Figure 3.10: PRQ and UMQ search time for process 0 in the FDS application with K-means algorithm on Cluster A

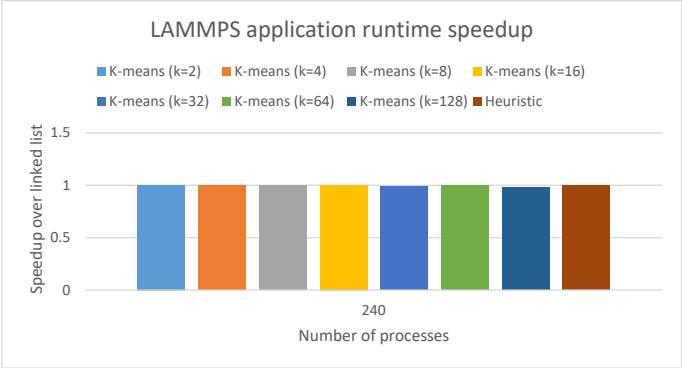
Table 3.2: Maximum number of (UMQ/PRQ) queue searches in the applications

Application	Number of UMQ searches	Number of PRQ searches
LAMMPS	46,673,352	103,613,128
AMG2006	99,929,566	136,665,483
FDS	6,825,587,162	9,706,370,678

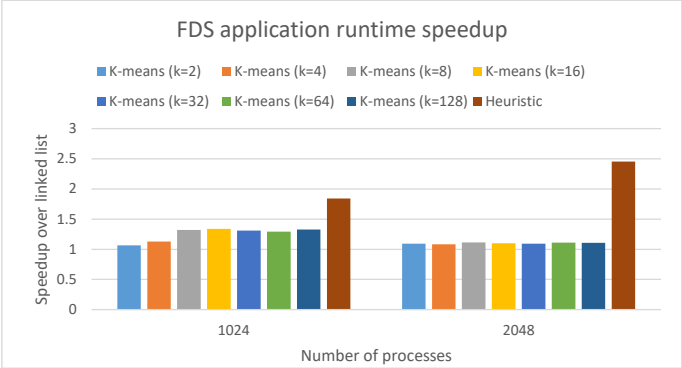
LAMMPS, mainly due to much smaller number of queue searches.



(a) AMG2006 application



(b) LAMMPS application



(c) FDS application

Figure 3.11: Application runtime in AMG2006, LAMMPS and FDS applications on Cluster A

3.5 Summary

In this chapter, we propose a new message matching algorithm that profiles the message queue behavior of applications to categorize the processes into some clusters. It will then assign a dedicated message queue to each cluster. The advantage of this approach is that it speeds up the search operation by using multiple queues. Moreover, unlike other algorithms in this area, it considers the message queue behavior of the peer processes to speed up the search operation. The evaluation results show that the proposed algorithm can reduce the number of traversals significantly. It can also improve the queue search time and application runtime by up to 44.2x and 2.4x, respectively. In Chapter 4, we will improve the proposed design in this chapter by proposing an approach that dynamically captures the application queue characteristics. Moreover, it will provide similar or better performance than the proposed design in this chapter without its memory footprint.

Chapter 4

Partner/Non-partner Message Queue Data Structure

In Chapter 3, we introduced a new message queue data structure that clusters the processes based on the number of queue elements they add to the queue, and allocates a dedicated message queue to each cluster. The clustering-based message queue data structure shows that clustering the processes based on the message queue behavior of the applications and assigning a dedicated message queue to each cluster can improve the message matching performance. However, there are two problems with this design. First, the clustering-based approach maintains information about clusters for each peer process in an array which results in large memory footprint at large scale. The second issue with the clustering-based approach is that it is *static*, meaning that it requires the application to be executed once to gather the profiling information before its actual run.

In this chapter, we propose a new MPI message queue design to deal with the issues in clustering-based approach [52]. The new message matching mechanism is based on *partner/non-partner* message queue traffic. It decreases the queue search time and at the same time maintains a scalable memory consumption. To this end, we utilize information about the dynamics of the message queue characteristics of the application and allocate dedicated queues for messages coming from certain processes. Consequently, in our design we build a collection of message queues that belong to two different classes: partners and non-partners. While each partner queue is used only for messages from a certain partner,

the non-partner queue provides a shared container for messages from all non-partner processes. This approach solves the memory scalability issue of the clustering-based approach by maintaining the information of just partner processes in a hash table rather than saving the information of all processes in an array.

The proposed design in this chapter is *dynamic*, meaning that it builds the queues gradually based on the application runtime characteristics. Obviously, the Dynamic approach is more practical for application users compared to the Static approach. However, we also present a static version of our design to compare its performance against the Dynamic approach.

The rest of the chapter is organized as follows. Section 4.1 discusses the motivation behind this work. Section 4.2 describes the Static and Dynamic approaches in the proposed message queue design. Section 4.3 presents the runtime and memory complexities of our design and compares them with the linked list and Open MPI data structures, respectively. The experimental results are presented in Section 4.4. Section 4.5 presents the related work and distinguishes our work from them. Finally, we conclude the chapter in Section 4.6.

4.1 Motivation

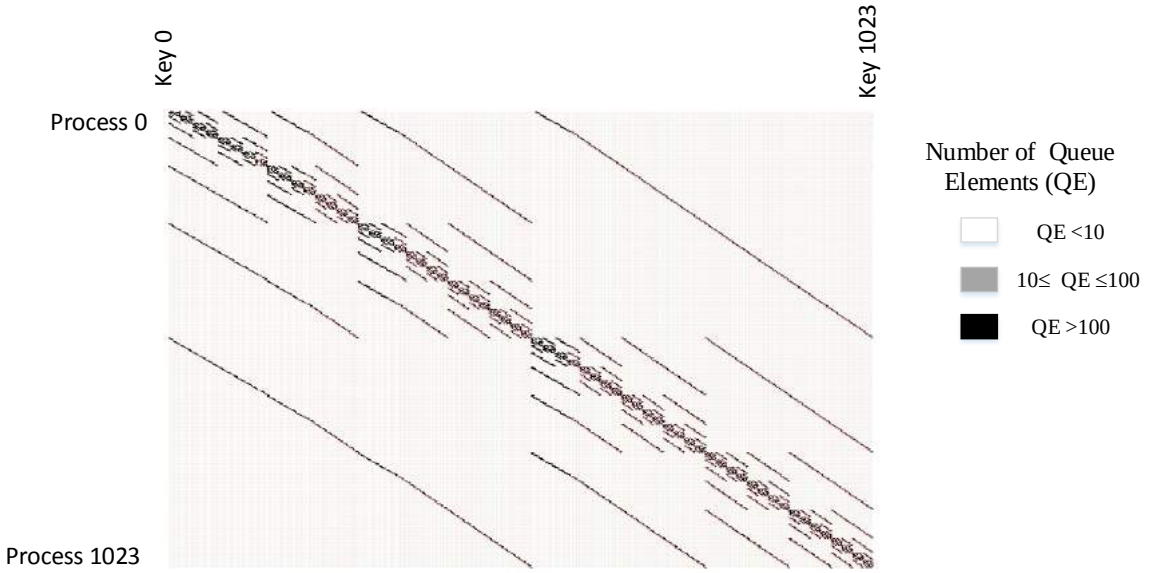
Section 2.2.5 discussed the message queue data structures used in current MPI implementations such as linked list data structure in MPICH and MVAPICH and array-based data structure in OpenMPI. A linear linked list search is acceptable for applications that do not generate long queues. However, for applications with long list traversals, a single linked list imposes a significant overhead due to its computational complexity [24]. Thus, it is important to have a message queue design that is fast in terms of traversing/matching and also scalable in terms of memory consumption.

The array-based data structure in Open MPI highly improves the performance. However, the problem is that it allocates one queue for each peer process. Such an allocation

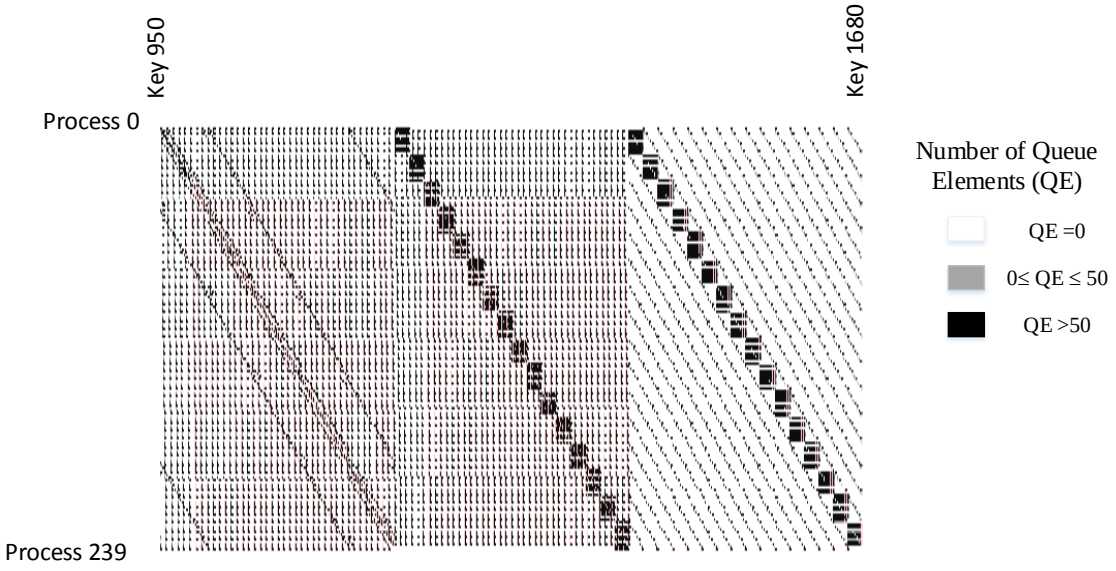
scheme may not incur high memory overheads at small scales. However, the once-for-all allocation scheme causes linear degradation of memory consumption at large scale. Moreover, allocating arrays equal to the size of the communicators will waste high amounts of memory at large scales as many elements of the array might not be even used at all. The reason is that most well-crafted MPI applications avoid the fully connected communication pattern in which all processes communicate with all the other processes. Consider AMG2006 and LAMMPS application as an example. Figure 4.1 shows the total number of elements sent to the queues from different processes in AMG2006 and LAMMPS. In the figure, each row shows the queue profiling information for one process. Each column corresponds to a key value that represents a process with a specific rank and context-id derived from Eq. 4.1. In this equation, P is the number of processes and $Mapped\_context\_id$ is an integer value derived from mapping the context-id to a small integer range between 0 and the total number of communicators minus one.

$$Key = Rank + (P \times Mapped\_Context\_id) \tag{4.1}$$

In Figure 4.1, we only show the queue profiling information for a fraction of the keys. The black and gray data points represent high communicating processes, whereas the white data points show low communicating processes. It can be seen that many processes send a few or no messages to the queue of the other processes. This shows the inefficiency of the Open MPI data structure in terms of allocating unnecessary memory for such processes, which becomes a huge issue at scale. This observation, along with the unscalable performance of the linked list, motivates us to design message matching mechanisms that are scalable in terms of both speed of operation and memory consumption by allocating dedicated message queues only to processes who have large number of messages in the queues. Our proposed message queue designs in this chapter outperform the linked list data structure in queue search time by speeding up the search operation and reducing the number of queue traversals. In



(a) AMG2006



(b) LAMMPS

Figure 4.1: Number of elements sent to UMQ/PRQ from different processes in the AMG2006 and LAMMPS applications on Cluster A

addition, unlike the approach used in Open MPI, our queue designs avoid wasting memory by considering the communication pattern and queue behavior of the applications.

4.2 The Proposed Partner/Non-partner Message Queue Design

The core idea of our proposed design is to allocate a dedicated queue for each process that sends/posts a large-enough number of messages/receives to the UMQ/PRQ. We refer to such processes/queues as partner processes/queues. We design the partner/non-partner message queue data structure using two different approaches: a Static approach and a Dynamic approach. The Dynamic approach is certainly more promising and relevant in practice. However, the Static approach is practical in cases where an application is expected to be run many times and each run is considerably affected by message queue traversal overheads.

4.2.1 Metrics for Selecting Partner Processes

For extracting the partner processes, we first count the number of elements each process sends to the queue. Then, we use this information to calculate an edge point. The edge point is a threshold parameter for selecting the partner processes. Any process whose number of elements in the queue is more than the edge point is chosen as the partner. Three different metrics, average, median, and upper fence outliers, are used to determine the edge point. The advantage of the average metric in selecting the partners is that it considers every value in the data. Moreover, its calculation is less intensive compared to the median and upper fence outliers metrics, since it does not require sorting the elements. However, the problem with the average metric is that it is sensitive to the extreme values. On the other hand, median is more robust to the extreme values, and is a better indicator of the dataset.

Eq. 4.2 shows the upper fence formulation, where Q_3 and Q_1 are the upper and lower quartiles, respectively, and α is a constant coefficient. Using the upper fence outliers as the edge point, only the processes who have considerably larger number of elements in the

queue are chosen as the partners. This would potentially result in extracting fewer partner processes compared to the average and median metrics, leading to fewer partner queues and hence improved memory consumption at the expense of less performance improvement.

$$\text{Upper\_Fence} = Q_3 - \alpha \times (Q_3 - Q_1) \tag{4.2}$$

4.2.2 The Static Approach

Figure 4.2 illustrates the Static partner/non-partner message queue design. In this approach, the application is profiled once to gather the total number of QEs each process sends to the UMQ/PRQ during the entire application runtime. This information is used to identify the partner processes based on one of the metrics discussed in Section 4.2.1. Information about the partner processes is then saved in a hash table to be used in the future runs, where a dedicated message queue is allocated for each partner process. In addition, a single non-partner queue is allocated for the QEs from all non-partner processes. A hash table is used for its $O(1)$ search complexity and less memory overhead over other data structures, as discussed in Section 4.3.2.

Algorithm 4.1 shows the steps involved in extracting the partner processes in the profiling stage. The metrics discussed in Section 4.2.1 are used to determine the edge point for selecting the partners (Line 1). Any process whose number of elements in the queue is greater than the edge point is considered as a partner and its corresponding $< hash\_key, hash\_value >$ pair is added to the hash table (Lines 3 to 10). Eq. 4.1 is used to generate the hash keys; this ensures that the hash key is unique for all the ranks in different communicators. As most applications usually use a small number of communicators, the time for mapping context-id to small integers is negligible.

The hash value is a unique integer for each process, between 0 and $n_{\text{p}} - 1$, where n_{p} is the number of partners for each process. Each hash value indicates one partner queue. We also count the number of selected partners for each process in Line 8.

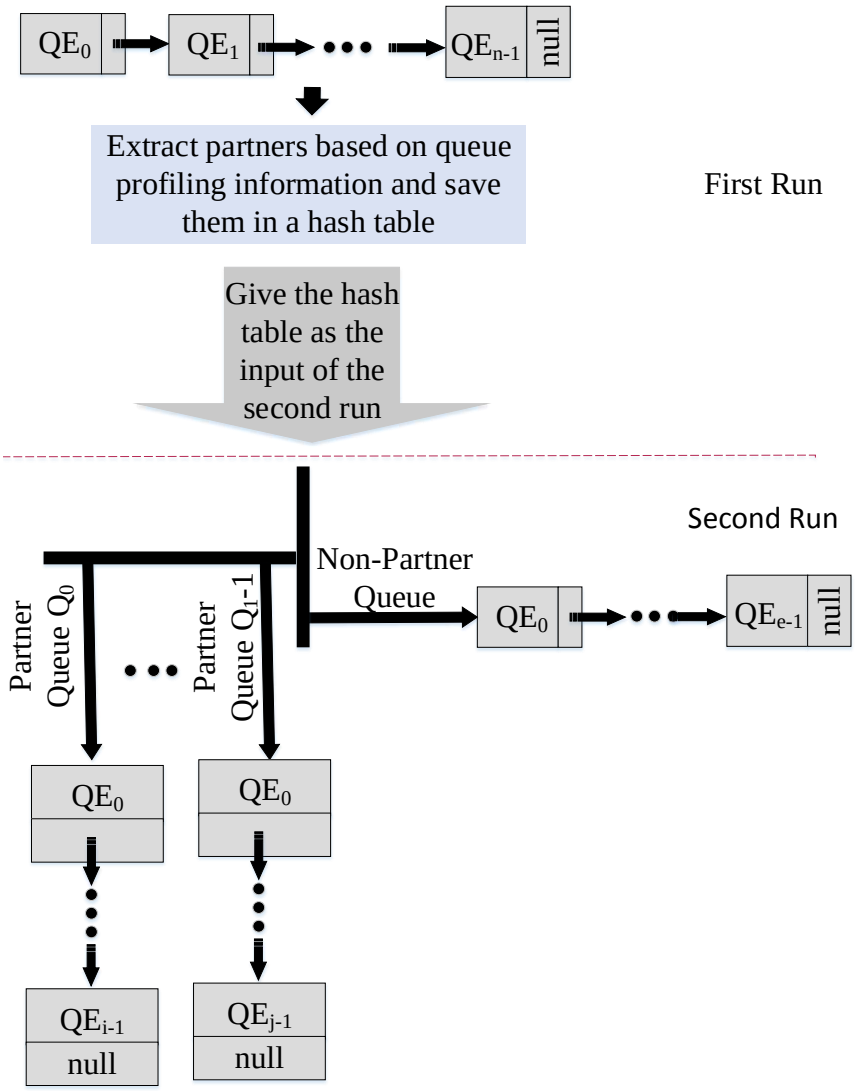


Figure 4.2: Static partner/non-partner message queue design

We should note that any hash function could be used in Algorithm 4.1. However, considering that in the Static approach the partners are selected once in the profiling stage and that they do not change during the future runs of the application, there will be no insertion/deletion to/from the hash table. Therefore, we use a perfect hash function [2] to eliminate hash collisions. For the sake of completeness, we present the implementation of the perfect hash function below. The interested reader is referred to [2] for a detailed

Algorithm 4.1: Partner extraction in partner/non-partner message queue design

Input : Total number of elements sent to UMQ/PRQ from each process Num-of-QE$_{0...P}$, Number of processes P

Output: A hash table containing all partner processes HT, The number of partners for process p n_p

```
1  E = average/median/upper fence outlier of Num-of-QE_{0...P};
2  n_p = 0;
3  for i ∈ 1...P do
4      if Num-of-QE_i > E then
5          Generate hash_key for process i;
6          Determine hash_value(s);
7          Insert < hash_key, hash_value(s) > to the hash table;
8          n_p++;
9      end
10 end
```

description of the perfect hash function. Assuming that the keys are saved in array S, the algorithm to generate perfect hash function is as follows:

1. Find the parameter m such that $(m \times m) \geq max(S)$. Each key is to be mapped into a matrix of size $m \times m$.

2. Place each key k in the matrix at location (x, y), where $x = k \div m$, $y = k \mod m$.

3. Slide each row of the matrix to the right a number of times so that no column has more than one entry.

4. Collapse the matrix down into a linear array.

5. The hash function uses m and the displacements from steps 3 to locate k.

When the application runs a second time, the output of the profiling stage (the hash table and the number of partners for each process n_p) is used to improve the search operation. For that, n_p partner queues and one non-partner queue are created at initialization for each process. Algorithm 4.2 shows the message matching mechanism in the Static approach that

is used in the second run. To search for an element in the queue, we first use Eq. 4.1 to generate the search key based on the rank and context-id of the corresponding element (Line 1). Then, the search key is given as the input of the hash function to derive the hash table index (Line 2). If the hash key corresponding to the derived hash table index was equal to the search key, it means that the corresponding process is a partner and hence, we should search the corresponding (dedicated) partner queue specified by the hash value. Otherwise, we should search for the desired element in the non-partner queue (Lines 4 to 8).

Algorithm 4.2: The searching mechanism in the Static approach (Second run)

```
   Input : The hash table HT, The partner and non-partner queues, The searching
           element (Context-id, rank, tag)
1  Generate the search_key from rank and context_id;
2  hash_table_index = hash_function(search_key);
3  < hash_key, hash_value > ⇐ HT_Hash_table_index;
4  if hash_key == search_key then
5  |  Search the partner queue specified by hash_value;
6  else
7  |  Search the non-partner queue;
8  end
```

4.2.3 The Dynamic Approach

Unlike the Static approach, the Dynamic approach identifies the partner processes dynamically at runtime, without a need to a profiling stage. As soon as the size of the queue reaches a specific threshold, t, we identify the partner processes at that level, and allocate a dedicated message queue to each of them. From this point on, the incoming QEs from the partner processes are added to their own dedicated queues, whereas the messages from all the other processes (non-partners) are added to a single, shared non-partner queue. This procedure is repeated in a multi-level fashion during the application execution time. The motivation behind this approach is to capture the dynamics of the applications at runtime

and allocate the partner queues based on the discovery of the new partners in each phase of the application.

As shown in Figure 4.3, the Dynamic approach consists of multiple levels. Each level includes a number of partner queues as well as a single non-partner queue. If the length of the non-partner queue reaches the threshold, t, some new partners are identified and the non-partner queue itself is divided into a number of new partner queues and a new non-partner queue.

When searching the queues for non-partner processes, the initial (non-partner) queue at the Base Level is searched first. Then, all the non-partner queues from Level 0 to Level $L-1$ are searched in sequence. The green arrows in Figure 4.3 show the order in which a queue element from a non-partner process is searched. The order in which the queues are searched for partner processes depends on its level of partnership (i.e., the level in which a process becomes a partner[1]). For example, if a process becomes a partner at Level 0, then the initial queue and its dedicated queue are searched, in that order. However, if the process becomes a partner at Level 1, it might still have some elements in the non-partner queue at Level 0. Therefore, the order in which the queues are searched for this partner process will be: 1) the initial queue, 2) the non-partner queue at Level 0, and 3) its own dedicated queue (red arrows in Figure 4.3).

Similar to the Static approach, we use a hash table to store the partner processes, and to distinguish them from the non-partner processes at search time. To search the partner queues, the hash table should also contain the queue number and the level of partnership for the corresponding partner process. The hash key is derived from Eq. 4.1 and is unique for each partner process. Each key has two hash values. The first value shows the queue number of the corresponding partner process, whereas the second hash value shows the level of partnership for that process. Any hash function can be used for this purpose. However, we do not use the perfect hash function that was used in the Static approach,

[1]From this point on, we refer to level of partnership for process p as level-of-partnership$_P$

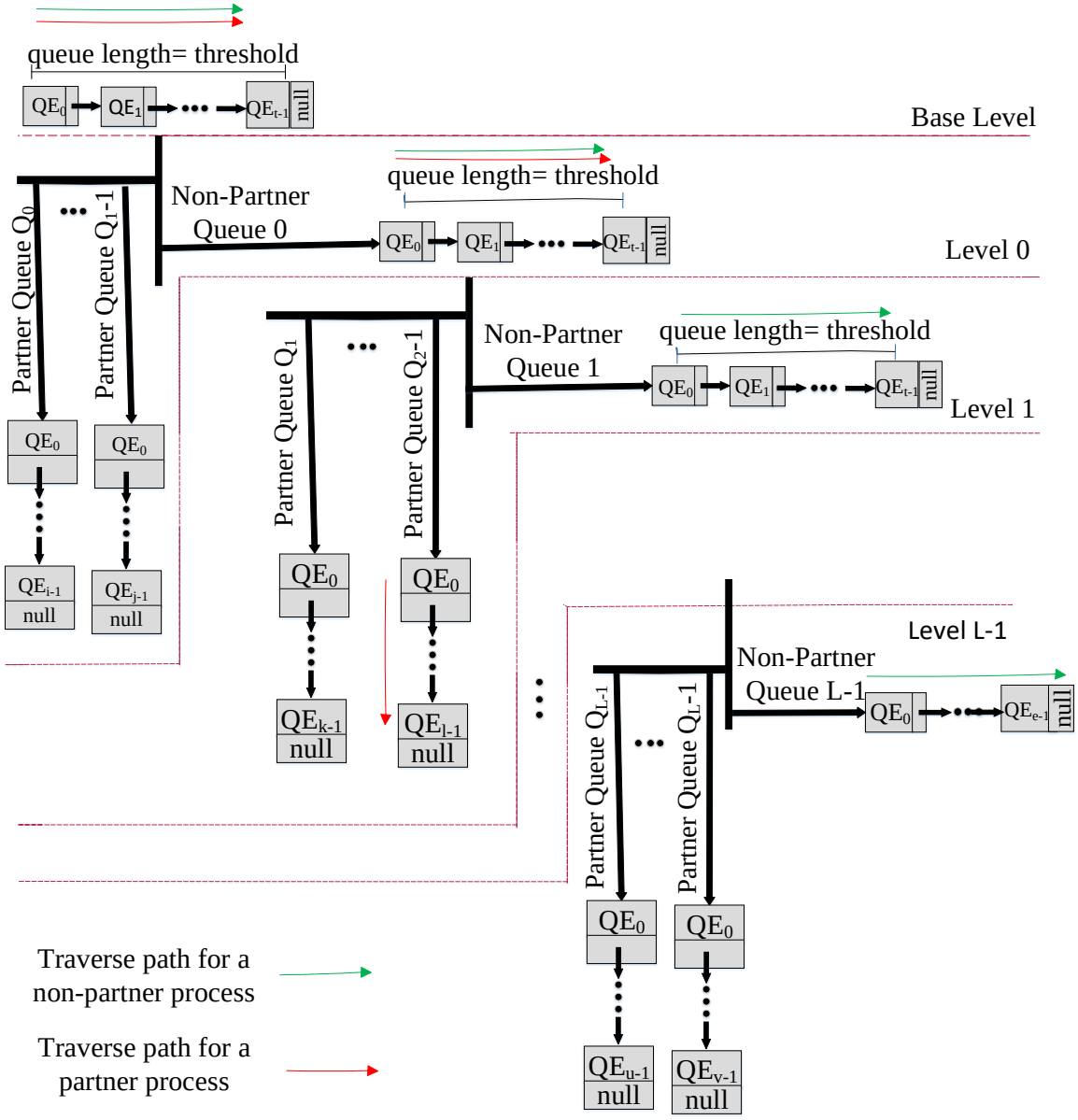


Figure 4.3: Dynamic partner/non-partner message queue design

as it is designed to search a Static list with no insertions or deletions. The Dynamic approach though requires some new partners to be inserted into the hash table at each level. Therefore, we use a round-robin hash table for our purpose. To maintain memory scalability in the Dynamic approach, an important advantage of our design is that we bound

the size of the hash table to a sublinear order of the total number of processes. So, new partners are extracted only if the bound on hash table size is not violated. We restrict the total number of partner processes to $c\sqrt{N}$, where N denotes the number of processes and c is an environment variable. The hash table, thus, has $\sqrt{N}$ number of entries, where each entry can have c elements, each for a partner process.

Algorithm 4.3 shows the message matching design in the Dynamic approach for the UMQ. The same scheme is used when searching the PRQ. When searching the UMQ, the initial queue is searched first in Line 1. If the element is found, we are done with the search. Otherwise, we check the level number. If the level number is less than zero, it means that there are no more queues to search and thus, the element is added to the initial PRQ (Line 37). If the level number is more than 0, it means that there exist some partner and non-partner queues, so the search is continued. In order to determine the queues that should be searched at this point, we should find out whether the searching element belongs to a partner process or not. To do so, we simply generate the search key corresponding to the searching element using Eq. 4.1 (Line 6). Then, we look into the hash table for the search key (Lines 7 to 9). The same procedure used in the Static approach is used here to determine if the search key exists in the hash table. If the search key was in the hash table, it means that the searching element belongs to a partner process (for example, process p). Therefore, all of the non-partner queues from Level 0 to the level-of-partnership$_{\text{p}}$ are searched first, followed by the partner queue dedicated to process p (Lines 10 to 15). Otherwise, the process is not a partner, so all the non-partner queues from Level 0 to $L-1$ are searched, respectively (Line 23 to 25). If the element is not found in any of these queues, it is added to the partner queue pq_{p} (if it is a partner), or to the non-partner queue $npq_{\text{L-1}}$ (if it is a non-partner process) as shown in Lines 19 and 29, respectively. Each time an element is added to the non-partner queue $L-1$, its queue length is compared to the threshold value, t. If the queue length is greater than the threshold, some new partners are then identified. We use the same procedure presented in Algorithm 4.1 for selecting the partners (Line 31).

Algorithm 4.3: Message matching design in the Dynamic approach (Search UMQ if not found add a new QE to PRQ)

Input : The hash table HT, Number of levels L, The partner queues (pq_{Q_0}...$pq_{Q_{L-1}}$) and non-partner queues (npq_0...npq_{L-1}), The searching element (Context-id, rank, tag), The UMQ length QL, the threshold T

Output: The element found or added to the queue QE

```
1  Search the initial queue;
2  if element found then
3  |  Return QE;
4  else
5  |  if L ≥ 0 then
6  |  |  Generate the search_key from rank and context_id;
7  |  |  hash_table_index = hash_function(search_key);
8  |  |  < hash_key, hash_value_1, hash_value_2 > ⇐ HT_Hash_table_index;
9  |  |  if hash_key == search_key then
10 |  |  |  pq_P = hash_value_1;
11 |  |  |  level-of-partnership_P = hash_value_2;
12 |  |  |  for i ∈ 0...level_of_partnership_P do
13 |  |  |  |  Search the non-partner queue i;
14 |  |  |  end
15 |  |  |  Search the partner queue pq_P;
16 |  |  |  if element found then
17 |  |  |  |  Return QE;
18 |  |  |  else
19 |  |  |  |  Generate QE and add it to partner queue pq_P;
20 |  |  |  |  Return QE;
21 |  |  |  end
22 |  |  else
23 |  |  |  for i ∈ 0...L − 1 do
24 |  |  |  |  Search the non partner queue i;
25 |  |  |  end
26 |  |  |  if Element found then
27 |  |  |  |  Return QE;
28 |  |  |  else
29 |  |  |  |  Generate QE and add it to npq_L-1;
30 |  |  |  |  if QL > T then
31 |  |  |  |  |  Select new partners and add them to the hash table using Algorithm 4.1;
32 |  |  |  |  end
33 |  |  |  |  Return QE;
34 |  |  |  end
35 |  |  end
36 |  else
37 |  |  Generate QE and add it to initial PRQ;
38 |  |  if QL > T then
39 |  |  |  Select partners and add them to the hash table using Algorithm 4.1;
40 |  |  end
41 |  |  Return QE;
42 |  end
43 end
```

The procedure of identifying the new partners is continued until we are limited by the size of the hash table.

It should be mentioned that there is a small, fixed cost associated with searching the hash table in the proposed partner/non-partner message matching architecture. To compensate for this cost, the queue length threshold parameter, t, should be selected wisely so as to improve the performance for long list traversals while delivering the same performance as the linked list data structure for short list queues. In the experimental results, we show the impact of different queue length thresholds. Note that we use the same mechanism discussed in Chapter 3 to support the wildcard communications.

4.3 Complexity Analysis

In this section, we present the time and memory complexities of the proposed partner/non-partner message queue and compare them against the linked list approach used in MVAPICH and the array-based approach used in Open MPI.

4.3.1 Runtime Complexity

Message matching consists of three main operations: insertion, deletion and search. Insertion has an $\mathcal{O}(1)$ complexity in all the three designs (linked list, Open MPI queue data structure and the proposed partner/non-partner design). For the linked list, this is achieved by storing the tail of the linked list and inserting the elements at the end of the list. The same is true for insertion into a specific linked list in the array-based and partner/non-partner designs. Finding the target linked list in these designs though involves certain computations, but such computations are actually incurred in a search operation that precedes each insertion. More specifically, insertion into the UMQ precedes by a search in the PRQ, and vice versa. The deletion operation will also have an $\mathcal{O}(1)$ complexity in all the three designs since it always happens after the search operation at the position where a

Table 4.1: List of parameters and their definitions

q	total number of elements in the queue
q_k	number of queue elements with context-id k
q_{kr}	number of queue elements with context-id k and rank r
K	number of active context-ids at a given process
r_k	number of ranks in context-id k
L	total number of levels
n_{lkr}	number of non-partner queue elements with context-id k and rank r at level l
p_{lkr}	number of partner queue elements with context-id k and rank r at level l
l_{kr}	level at which the process with rank r in context-id k becomes a partner
a_k	the number of any_source queue items associated with the communicator k
R	maximum size of the communicators
N	total number of processes
P	total number of partner processes

queue item is found.

Linked List Message Queue

In order to discuss the search complexity, we will use the parameters defined in Table 4.1. As discussed in Section 2.2.5, the search complexity for the linked list data structure can be given by Eq. 4.3 as we need to search through all the elements in the queue.

$$\mathcal{O}(q) \tag{4.3}$$

In order to compare the search complexity of the linked list data structure with Open MPI and the proposed partner/non-partner message queue design, we will present the search complexity based on the number of elements from each rank and context-id. Eq. 4.4 shows that the total number of elements in the queue is the sum of the number of elements from all context-ids.

$$q = \sum_{k=0}^{K-1} q_k \tag{4.4}$$

The number of elements in the queue from context-id k can be derived from Eq. 4.5.

This equation shows that the number of elements in the queue with context-id k is the sum of the number of elements from all its ranks.

$$q_k = \sum_{r=0}^{r_k-1} q_{kr} \tag{4.5}$$

By substituting Eq. 4.5 into Eq. 4.4, we have:

$$q = \sum_{k=0}^{K-1} \sum_{r=0}^{r_k-1} q_{kr} \tag{4.6}$$

By substituting Eq. 4.6 into Eq. 4.3, the search complexity based on the number of elements from each rank and context-id will be:

$$\mathcal{O}\left(\sum_{k=0}^{K-1} \sum_{r=0}^{r_k-1} q_{kr}\right) \tag{4.7}$$

In the case of MPI_ANY_SOURCE wildcard communication, such elements in the queue should be traversed as well. Therefore, the time complexity is given by Eq. 4.8.

$$\mathcal{O}\left(\sum_{k=0}^{K-1} \sum_{r=0}^{r_k-1} (q_{kr} + a_k)\right) \tag{4.8}$$

Open MPI Message Queue

In the array-based design, we first search the linked list of context-ids in $\mathcal{O}(K)$, then use the array to access the dedicated linked list for rank r in $\mathcal{O}(1)$ and scan through it in $\mathcal{O}(q_{kr})$. Thus, its total search complexity is $\mathcal{O}(K + q_{kr})$. For long message queues, the number of context-ids is negligible compared to the total number of queue elements and the search complexity can be given by Eq. 4.9.

$$\mathcal{O}(q_{kr}) \tag{4.9}$$

In the presence of MPI_ANY_SOURCE wildcard communication, the posted receive

queue keeps such messages in a separate linked list queue, PRQ_ANY queue, as their rank field cannot be used as an array index. Therefore, we need to traverse the PRQ_ANY queue for any incoming messages on top of the linked list associated with the rank from the message envelope. The PRQ traversal complexity in the presence of MPI_ANY_SOURCE is therefore given by Eq. 4.10.

$$\mathcal{O}\left(q_{\mathrm{kr}} + a_{\mathrm{k}}\right) \tag{4.10}$$

When an MPI_ANY_SOURCE receive call is posted, all the UMQ elements associated with all the ranks must be searched. The UMQ traversal complexity in the presence of MPI_ANY_SOURCE is given by Eq. 4.11.

$$\mathcal{O}\left(\sum_{r=0}^{r_k-1} q_{kr}\right) \tag{4.11}$$

Partner/Non-partner Message Queue

In the proposed partner/non-partner design, the search operation starts with generating a hash key for the target element based on its corresponding context-id and rank. To this end, we first map the context-id to an integer value between 0 and the number of communicators. Then, the perfect hash function in the Static approach or the round-robin hash function in the Dynamic approach is used to generate the hash key. The context-id map takes $\mathcal{O}(K)$ in our current design as we need to traverse an array of size K. Using the hash key, we query the hash table in $\mathcal{O}(1)$ to find out whether the target element belongs to a partner process or not. If the source process is a non-partner process, we will scan each non-partner queue in increasing order of their level number $\mathcal{O}\left(K + \sum_{l=0}^{L-1}\sum_{k=0}^{K-1}\sum_{r=0}^{r_k-1} n_{lkr}\right)$. However, if it is a partner, we first scan the non-partner queues at Levels 0 to l_{kr}. After that, we jump to the dedicated partner queue of r to proceed with search $\mathcal{O}\left(K + (\sum_{l=0}^{l_{kr}}\sum_{k=0}^{K-1}\sum_{r=0}^{r_k-1} n_{lkr}) + p_{lkr}\right)$. For long message queues, the number of communicators K is negligible compared to the

number of queue elements and the search complexity for non-partner and partner process can be given by Eq. 4.12 and 4.13, respectively. We should note that these equations apply to both the Static and Dynamic approaches. However, there is only one level in the Static approach and hence, the summations corresponding to multiple levels in Eq. 4.12 and Eq. 4.13 will drop.

$$\mathcal{O}\left(\sum_{l=0}^{L-1}\sum_{k=0}^{K-1}\sum_{r=0}^{r_k-1} n_{lkr}\right) \tag{4.12}$$

$$\mathcal{O}\left((\sum_{l=0}^{l_{kr}}\sum_{k=0}^{K-1}\sum_{r=0}^{r_k-1} n_{lkr}) + p_{lkr}\right) \tag{4.13}$$

Note that the total number of elements q in the partner/non-partner message queue design is the sum of the elements for all partner and non-partner processes. In other words:

$$q = \sum_{l=0}^{L-1}\sum_{k=0}^{K-1}\sum_{r=0}^{r_k-1} n_{lkr} + \sum_{l=0}^{L-1}\sum_{k=0}^{K-1}\sum_{r=0}^{r_k-1} p_{lkr} \tag{4.14}$$

This shows that the partner/non-partner message queue design can reduce the number of traversals from $\mathcal{O}(q)$ in the linked list data structure (Eq. 4.3) to Eq. 4.12 and Eq. 4.13 for non-partner and partner processes, respectively.

In the presence of MPI_ANY_SOURCE wildcard communication, the posted receive queue, PRQ_ANY, dedicated to such messages should be searched as well. Therefore, the PRQ traversal complexity for a non-partner and partner process in the presence of MPI_ANY_SOURCE is given by Eq. 4.15 and Eq. 4.16, respectively.

$$O\left(\sum_{l=0}^{l=L-1}\sum_{k=0}^{K-1}\sum_{r=0}^{r_k-1} n_{lkr} + \sum_{k=0}^{K-1} a_k\right) \tag{4.15}$$

$$O\left((\sum_{l=0}^{l_{kr}}\sum_{k=0}^{K-1}\sum_{r=0}^{r_k-1} n_{lkr}) + p_{lkr} + \sum_{k=0}^{K-1} a_k\right) \tag{4.16}$$

When an MPI_ANY_SOURCE receive call is posted, all the UMQ elements associated with all the ranks must be searched. The UMQ traversal complexity in the presence of

MPI_ANY_SOURCE is given by Eq. 4.17.

$$O\left(\sum_{l=0}^{L-1}\sum_{k=0}^{K-1}\sum_{r=0}^{r_k-1} n_{lkr} + \sum_{l=0}^{L-1}\sum_{k=0}^{K-1}\sum_{r=0}^{r_k-1} p_{lkr}\right) \tag{4.17}$$

4.3.2 Memory Overhead Complexity

The total amount of memory required for each message queue data structure consists of two parts: the memory required to store the message queue elements, and the additional memory required to maintain the data structure, which we refer to it as the memory overhead of the data structure. The required memory for message queue elements is the same for all three designs (linked list, Open MPI queue data structure, and the proposed partner/non-partner design). Therefore, in this section, we just discuss the memory overhead complexity of the three designs.

Linked List Message Queue

The linked list message queue design in MVAPICH does not impose any memory overhead as it only stores message queue elements in a linked list. Therefore, the memory overhead complexity can be given by Eq. 4.18.

$$\mathcal{O}(1) \tag{4.18}$$

Open MPI Message Queue

In the Open MPI queue design, the memory overhead consists of two parts: (1) the linked list of communicators/context-ids, and (2), one array of size R per communicator. Thus, its memory complexity is given by Eq. 4.19.

$$\mathcal{O}(K \times R) \tag{4.19}$$

Note that in case of an application having only a single communicator (MPI_COMM _WORLD) $K = 1$, and the total number of processes N is equal to number of ranks R in MPI_COMM_WORLD. In this case, the memory overhead complexity of Open MPI queue design is minimized and Eq. 4.19 is equal to $\mathcal{O}(N)$.

Partner/Non-partner Message Queue

The partner/non-partner design imposes some memory overhead to store the information about the partner processes in a hash table. The number of entries in the hash table is proportional to the number of extracted partners. Therefore, the memory complexity overhead of the partner/non-partner design will be $\mathcal{O}(P)$. Thus, our approach does not have the fixed and unscalable memory overhead of Open MPI queue data structure.

The number of partner processes depends on the application characteristics. In the worst case, all the processes will become a partner which results in the same memory consumption as the Open MPI queue design. However, we bound the number of partners (hash table entries) in the partner/non-partner design so as to guarantee a better memory scalability than the Open MPI queue design.

For choosing the cap for the number of queues, we considered the memory consumption in linked list and Open MPI queue data structure. As discussed in Section 4.3.2, the inked list data structure has scalable memory consumption and its memory overhead is $\mathcal{O}(1)$. However, it is not scalable in terms of speed of operation. On the other hand, Open MPI queue data structure is faster than linked list but its memory overhead is at least $\mathcal{O}(N)$, as discussed in this section. In partner/non-partner message queue design, we take an in-between approach and bound the number of queues for partner processes to $c \times \sqrt{N}$, where c denotes a constant factor used to evaluate the impact of increasing the memory cap on message matching performance. This will result in a memory complexity overhead of $\mathcal{O}(c \times \sqrt{N})$. Considering c as a constant parameter, the memory overhead complexity can be given by Eq. 4.20 for our partner/non-partner design. Bounding the number of partners

will provide a trade-off between memory and performance. Extracting more partners will benefit the execution time due to providing more dedicated queues. However, more partners will consume more memory in the partners hash table.

$$\mathcal{O}\left(\sqrt{N}\right) \tag{4.20}$$

4.4 Performance Results and Analysis

We evaluate the efficiency of our proposed approach by comparing it against the linked list data structure used in MVAPICH. The evaluation is done with three MPI applications: AMG2006, version 1.0 [114], LAMMPS, version 14May16 [97] and FDS, version 6.1.2 [89]. The evaluation was conducted on Cluster A. The specification of these applications as well as the cluster information are explained in Section 3.4.1.

4.4.1 Selecting the Partners

The first step towards the efficiency of the partner/non-partner message queue design is to choose the partners efficiently. Choosing a low communicating process as partner would result in allocating unnecessary queues which degrades the memory consumption. On the other hand, if a high communicating process is not selected as partner, it would weaken the efficiency of the partner/non-partner approach. In this section, we provide experimental results to show that the partners are selected appropriately in our design.

Figure 4.4 shows the selected partners using the average metrics in AMG2006 and LAMMPS applications, respectively. The processes are selected using the Dynamic approach with the queue length threshold equal to 100. The black data points are the processes that are selected as partners while the white data points are non-partner processes. Comparing Figure 4.1 with the partners selected in Figure 4.4, it is obvious that most of the high communicating processes are correctly selected as partners. Note that we have

observed almost the same results with other threshold values and metrics.

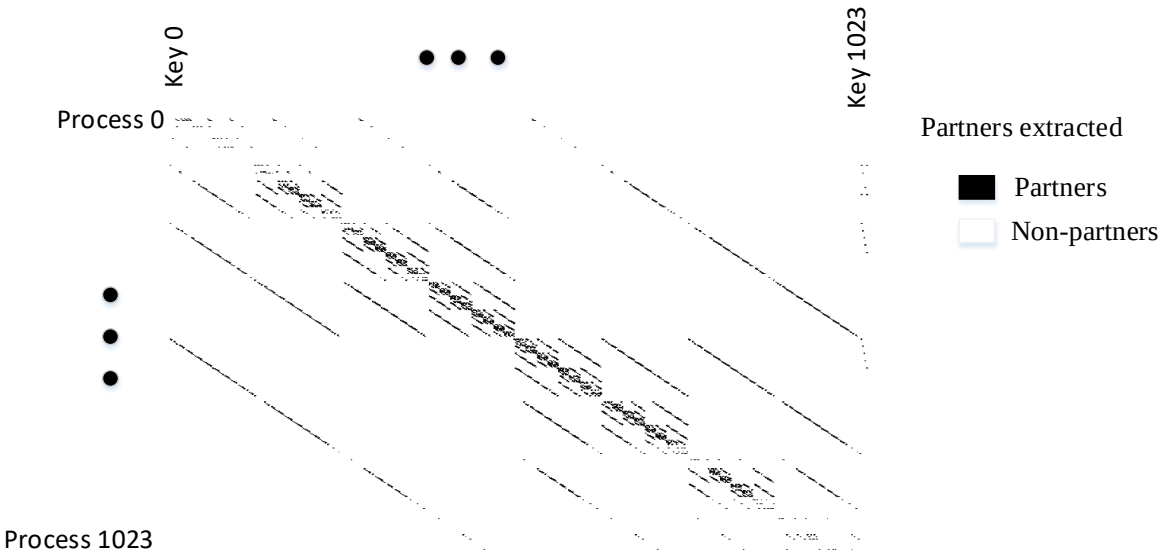
As discussed in Chapter 3, in the FDS application, process 0 is the only process that has a significant number of communications. Therefore, we avoid presenting a figure similar to Figure 4.4 for the FDS application as it would not convey much information due to containing mostly white data points, except for the first row.

4.4.2 Queue Search Time

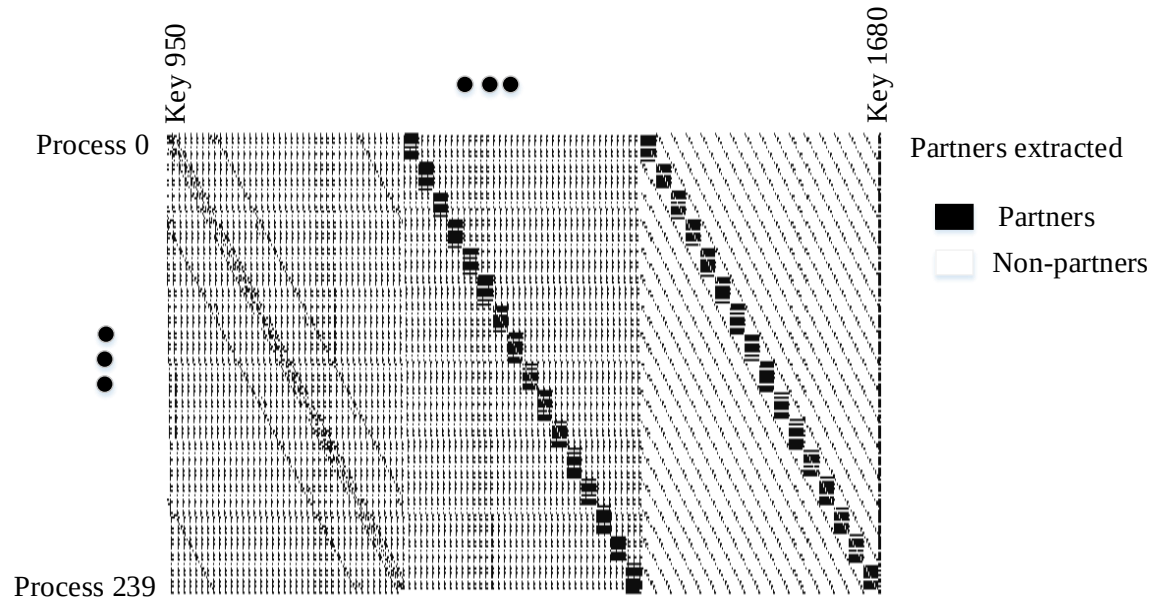
In this section, we compare the queue search time of the linked list data structure with the partner/non-partner message queue design for the FDS, AMG2006 and LAMMPS applications. Figure 4.5 shows the PRQ and UMQ search time speedup over the linked list for these applications when the average, median and outliers (with $\alpha = 0$) metrics are used for selecting the partners. The results for the Dynamic approach are shown with different threshold values, t, from 25 to 1600.

The first observation from Figure 4.5 is that in almost all cases the Static approach provides a better or similar performance compared to the Dynamic approach. This complies with the general intuition that the Static approach should always provide a better performance due to having a global view of application's message queue characteristics. However, as shown in the figure, the Dynamic approach can lead to better results in some cases due to a more aggressive partner extraction. This is because the Dynamic approach decides about partner extraction with respect to (multiple) short-term queue status. On the other hand, the Static approach considers the overall communication pattern based on which fewer processes might appear to be partners in some cases. Another observation is that all the studied metrics (average, median and outliers) could successfully select the partners and none of them has a significant advantage over the others.

Figure 4.5(a) and Figure 4.5(b) show the average PRQ and UMQ search time over all processes for AMG2006, respectively. Figure 4.5(a) shows that we can reach up to $2.32x$ speedup in PRQ search time for AMG2006. However, as can be seen in Figure 4.5(b), the

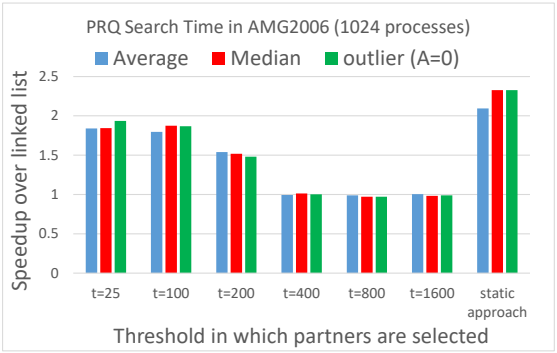


(a) AMG2006

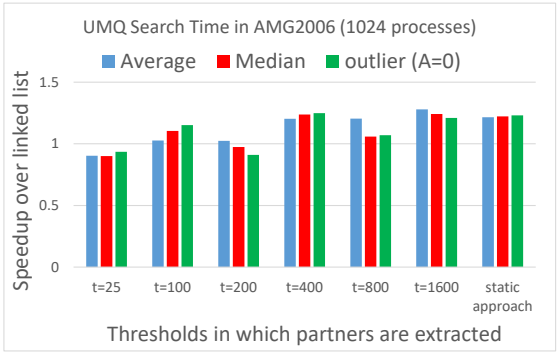


(b) LAMMPS

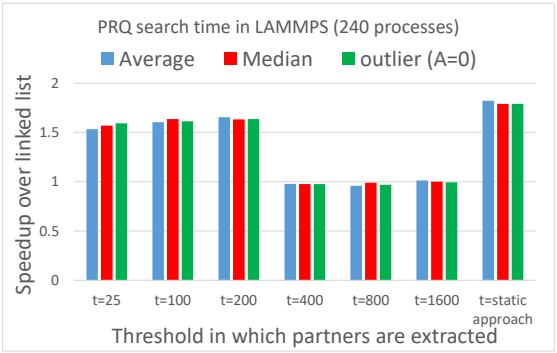
Figure 4.4: Selected partners using the average metric in AMG2006 and LAMMPS with threshold = 100 in the Dynamic approach on Cluster A



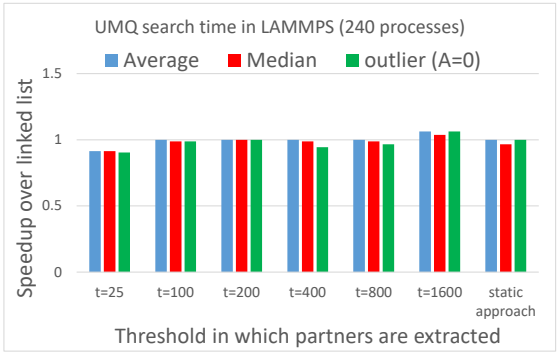
(a) PRQ search time speedup in AMG2006



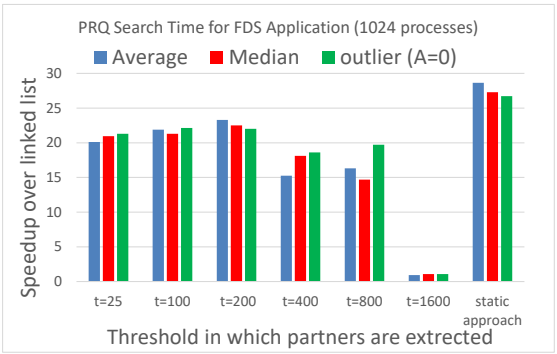
(b) UMQ search time speedup in AMG2006



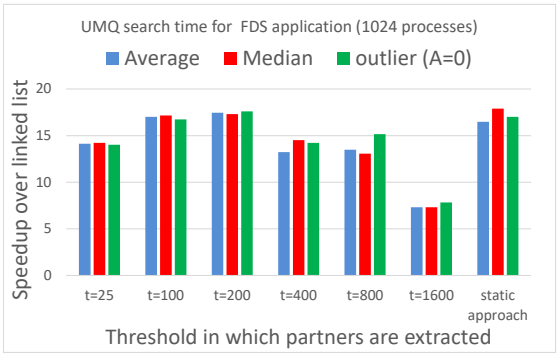
(c) PRQ search time speedup in LAMMPS



(d) UMQ search time speedup in LAMMPS



(e) PRQ search time speedup in FDS



(f) UMQ search time speedup in FDS

Figure 4.5: PRQ and UMQ search time speedup of partner/non-partner design over linked list in AMG2006, LAMMPS and FDS applications on Cluster A

UMQ speedup is not considerable, and with inappropriate t ($t = 25$) the speedup would be even less than 1. The reason for this lies in the short list traversal of the UMQ for many processes in AMG2006 which does not compensate for the overhead of the hash table lookup time when t is small. On the other hand, if the queue length, t, is considered very large, some processes do not build up such a lengthy queue, and so they would not extract any partners which results in queue search time degradation. For example, as can be seen in Figure 4.5(a) with $t \geq 400$, none of the processes selects any partner and the speedup is around 1. Figure 4.5(c) and 4.5(d) show almost the same results for LAMMPS. We should reiterate that LAMMPS generates longer queues with 240 processes than 1024 processes. Therefore, we show the results with 240 processes.

Figure 4.5(e) and Figure 4.5(f) show the average PRQ/UMQ search time for the FDS application. In this application, the majority of communications is done with process 0. Therefore, we show the PRQ and UMQ search time for process 0. It is evident that increasing t from 25 to 100 would result in higher improvement in the PRQ/UMQ search time. The reason is that larger t values would provide more profiling information for selecting the partners. However, if t is too large (e.g., $t = 1600$), only a few partners are selected which results in less performance improvement.

Figure 4.5 shows that $t = 100$ can provide reasonably good performance in almost all cases. We might get a better performance with $t > 100$, but some applications do not build up such a lengthy queue (Figure 4.5(c)). On the other hand, smaller t values might induce overhead for short list traversals (Figure 4.5(d)). Therefore, the value of t should simply represent the threshold at which a linked list traversal becomes too costly in terms of message progression. This can be experimentally measured for each application on a given target system.

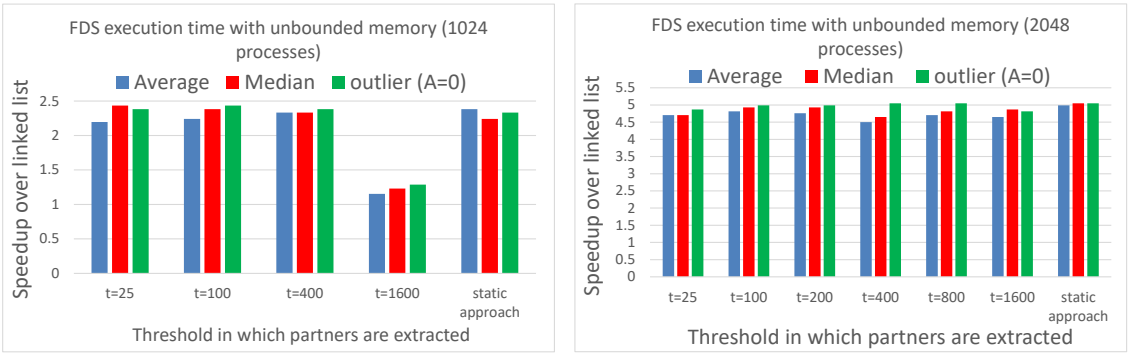
4.4.3 Application Execution Time

Figure 4.6 compares the FDS execution time using the partner/non-partner message queue against the linked list for 1024 and 2048 processes and under different metrics and queue length thresholds. As can be seen, the proposed design can improve the application runtime by up to 2.4x for 1024 processes and 5x for 2048 processes.

Figure 4.6 shows the execution time speedup when the size of the hash table is unbounded, so-called the *unbounded memory* case. This way, the application can add as many partners as it can find to the hash table. The problem with having too many partners is that it increases the memory consumption. In worst case, the number of partners for each process would be equal to the total number of processes in all communicators and the memory consumption would converge to the Open MPI data structure.

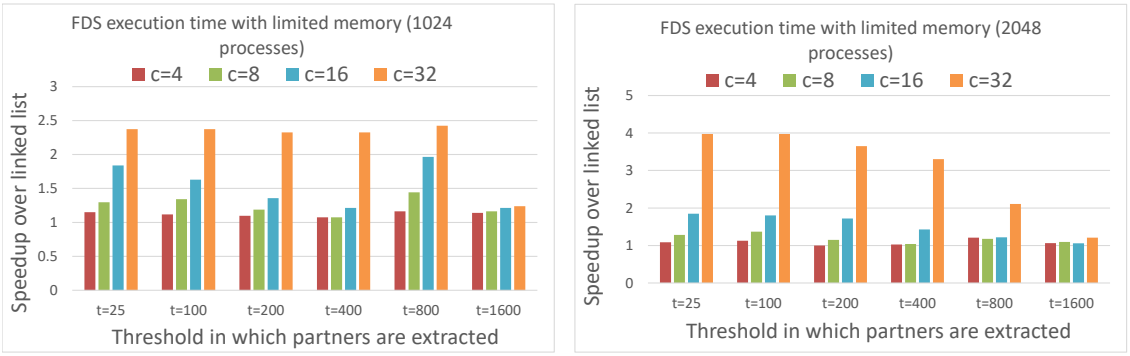
In order to achieve a sublinear memory complexity as discussed in Section 4.2.3 and Section 4.3.2, we restrict the number of elements in each entry of the hash table to c in the Dynamic approach. We call this the *bounded memory* case. Figure 4.7 shows the results for 1024 and 2048 processes, respectively. Because the different partner extraction metrics provide almost the same performance improvement in Figure 4.5 and Figure 4.6, we only show the results for the average metric in Figure 4.7. The results in this figure suggest that the speedup increases with the increasing values for c, except with a much lesser extent for the case of t equal to 1600. The reason is that as we use larger values for c, the size of the hash table increases and that would allow having more partners/queues.

We do not present the runtime results for AMG2006 and LAMMPS because their queue search time improvements do not translate to a significant improvement in their application runtime. The reason behind this lies in the number of times the UMQ and PRQ are searched in these applications. As discussed in Section 3.4.4, there is a sharp contrast between the number of queue searches for AMG2006 and LAMMPS and that of FDS, which could translate to application performance only for FDS.



(a) 1024 processes (b) 2048 processes

Figure 4.6: FDS application runtime speedup over linked list with unbounded memory on Cluster A



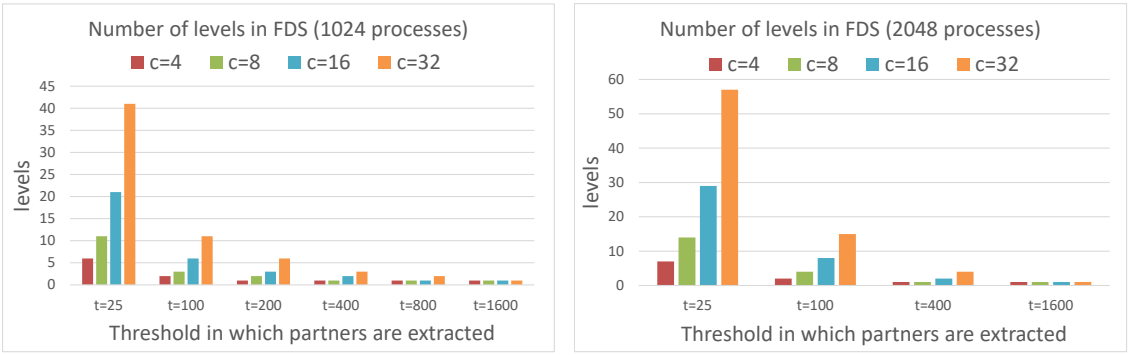
(a) 1024 processes (b) 2048 processes

Figure 4.7: FDS application runtime speedup over linked list with bounded memory and using average as the partner extraction metric on Cluster A

4.4.4 Number of Levels and Partners

As mentioned earlier, the Dynamic approach finds the partner processes in multiple levels, and that allows capturing the dynamics of the applications. Figure 4.8 presents the number of levels in the FDS application with 1024 and 2048 processes. The results show that as we decrease the queue length threshold, t, the queue length reaches the threshold more frequently, and consequently the number of levels is increased. Another observation is that reducing c fills out the hash table sooner which results in reducing the number of levels.

Figure 4.9 presents the total number of selected partners for all processes in the FDS



(a) 1024 processes (b) 2048 processes

Figure 4.8: Number of levels in FDS application on Cluster A

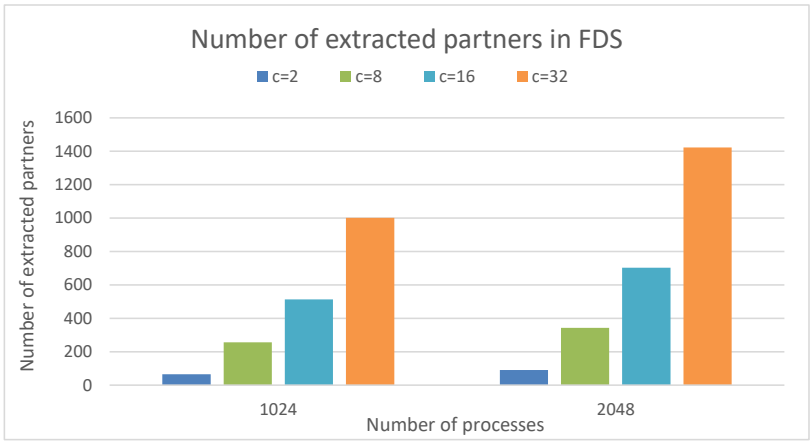
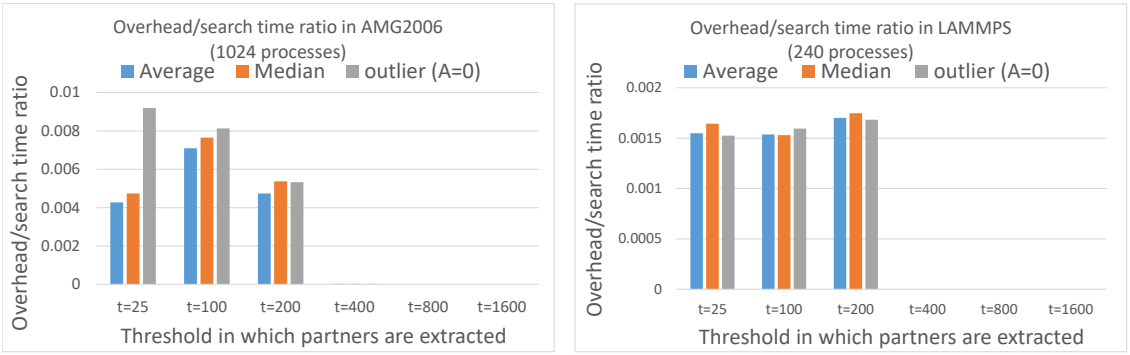


Figure 4.9: Number of extracted partners in FDS application for different number of processes and $t = 100$ on Cluster A

application for 1024 and 2048 processes when $t = 100$. The number of partners depends on the size of the hash table and the parameter c. As long as the hash table is not full, the partners are extracted independent of the threshold, t. We have observed similar results with other t values.

4.4.5 Overhead of Extracting Partners

The partner/non-partner message queue imposes some overhead when partners are extracted. In the Static approach, the partners are selected in the profiling stage, so there is



(a) AMG2006 (b) LAMMPS

Figure 4.10: Partner extraction overhead/search time ratio on Cluster A

no overhead in the future runs. However, in the Dynamic approach, partners are selected during the application runtime, and therefore the overhead should be quantified. Figure 4.10 shows the ratio of the average overhead for partner extraction over the average queue search time across all processes in AMG2006 and LAMMPS. It is clear that the overhead of extracting partners is negligible compared to the queue search time. For $t \geq 400$, the overhead is close to 0.

We should note that there is also some overhead involved in searching the hash table in our design. However, as searching the hash table is a part of searching the queue, the overhead is already included in the queue search results shown in Section 4.4.2.

4.5 Related Work

In Section 3.1, we discussed various research studies on message queue operations in MPI. In this section, we evaluate some of the state-of-the-art message queue mechanisms with more details. This way we can highlight the benefits of the partner/non-partner message queue data structure over them.

Recently, there have been various works on improving message queue operations in MPI [116, 46, 27, 71]. Most of these approaches try to improve the queue search time by reducing

the number of queue traversals. The problem with 4-dimensional data structure [116] is that it has a small fixed overhead for searching any queue item. If the queue length is large enough this overhead is negligible. However, in the case of short list traversals this data structure performs worse than the linked list data structure. In order to mitigate this problem, the authors used the communicator size as a metric to decide whether to use the 4-dimensional data structure. However, the communicator size is not always a good indicator of the average search length. For example, even with a communicator size of two, the two processes can have a lot of communications with each other which results in long list traversals.

In regard to bin-based approach [46], one of the issues is that it imposes some overhead for searching short queues. Another main problem is that we do not know how many bins should be used for a given application in order to have a good distribution of messages across all bins. Increasing the number of bins accelerates the search operation for long list traversals while it keeps the search time of short lists unchanged. Another impact of increasing the number of bins is that it increments the memory consumption. The optimal number of bins is totally application dependent and might be different for each process within an application. Our work differs from this work in that it avoids the unnecessary memory overhead by determining the suitable number of queues for each process based on application runtime communication pattern in the message queues. Moreover, our approach avoids sharing the bins between the processes who have high frequency of communication, and instead allocates a dedicated bin to each of such processes.

The authors in [27] try to address the overhead issue for short list traversals in the bin-based approach by proposing a message matching design that dynamically switches to one of the existing designs: a linked list, a bin-based design [46] and a rank-based design. This work performs better than the linked list data structure for long list traversals, and better than the bin-based/rank-based design for short list traversals. However, it suffers from the issues associated with these data structures. For example, it does not find the number of

bins/queues for a given application adaptively and dynamically. Switching to the rank-based design requires a user-configurable parameter, and also allocating a dedicated queue for each source process, resulting in low memory scalability. The proposed partner/non-partner message queue design differs from [27] in that it profiles the message queue communication traffic between each communicating peers and allocates a dedicated message queue only to the processes with high frequency of communication. Moreover, our approach determines the appropriate number of queues dynamically during the application runtime and avoids sharing the queues between high communicating processes.

4.6 Summary

Many parallel applications expose a sparse communication pattern in which each process has more communications with a certain group of other processes. We take advantage of this feature to propose a new message matching design that adapts based on the communication traffic of the applications. To do so, we measure the frequency of the messages sent from different processes to the queues and use this information to select a list of partners for each process. Then, we allocate a dedicate message queue for the partner processes. Other processes share a single non-partner queue.

We design the work using a Static and a Dynamic approach. The Static approach works based on the information from a profiling stage while the Dynamic approach uses the runtime information. We show that the Dynamic approach can be as efficient as the Static approach in terms of execution time. Our proposed approach can successfully decrease the queue search time for long list traversals while maintaining the performance for short list traversals. Moreover, it provides better scalability in terms of memory consumption. The evaluation results show that the queue search time and application runtime are improved by up to 28x and 5x for applications with extreme message matching requirements, respectively.

So far, we have proposed message matching mechanism that adapts based on application

characteristics. In Chapter 5, we will improve the message matching performance further by considering the type of communication (e.g. collective and point-to-point) in our design.

Chapter 5

A Unified, Dedicated Message Matching Engine for MPI Communications

In Chapter 3 and 4, we propose message matching mechanisms that consider the application behavior to improve the performance of message queue operations. In this chapter, we look at the message matching misery from a different perspective and we propose a new queue data structure that considers the type of communication to further improve the performance.

As discussed in Section 2, there are different types of communication supported in MPI implementations such as point-to-point and collective communications. In point-to-point operations and also collective operations that run on top of point-to-point communications, the messages must be matched between the sender and receiver. In previous chapters, we discussed that modern MPI implementations, such as MPICH, MVAPICH and Open MPI separate traffic at coarse granularity, either not at all, on a per MPI communicator level or by communicator and source rank. These solutions can be improved by intelligently separating traffic into logical fine-grained message streams dynamically during program execution. More specifically, message queue search time can be improved by distinguishing messages coming from point-to-point and collective communications and adapting message queue design accordingly. Collective operations are implemented using different algorithms. Each of these algorithms can have specific impact on the message queues. We take advantage of this feature to propose a new message matching mechanism that considers the type of

communication to enhance the message matching performance [50, 49]. In this chapter, we make the following contributions to improve the message matching performance:

- We propose a novel communication optimization that accelerates MPI traffic by dynamically profiling the impact of different types of communications on message matching performance and using this information to allocate dedicated message matching resources to collective and point-to-point communications. We use the partner/non-partner message queue design [52] (proposed in Chapter 4) for point-to-point communications alongside a proposed collective engine in a unified manner to enhance both collective and point-to-point message matching performance. Our approach determines the number of dedicated queues dynamically during the application runtime.
- We conduct several experiments to evaluate the impact of the proposed approaches on the performance gain of the collective and point-to-point elements from different perspectives. We show the impact of the unified collective and point-to-point message matching design on queue search time. Moreover, we evaluate the impact of collective message matching optimization on both collective and point-to-point elements. We demonstrate that our unified approach accelerates the collective and point-to-point queue search time by up to 80x and 71x, respectively, and that it achieves a 5.5x runtime speedup for a full application over MVAPICH.

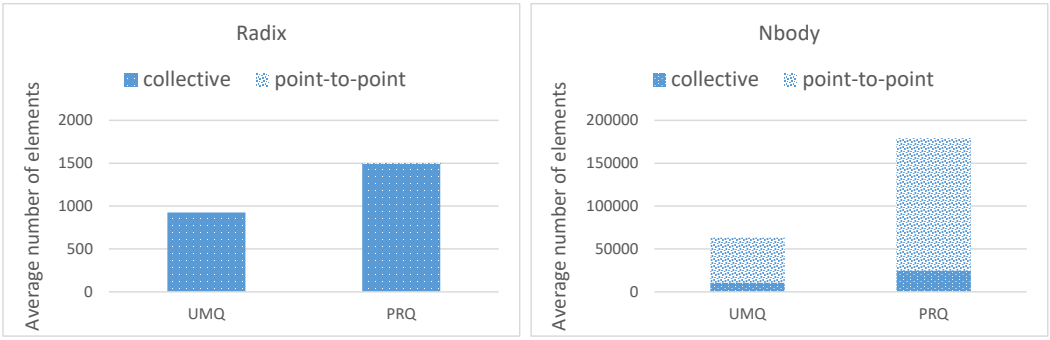
The rest of this chapter is organized as follows. Section 5.1 discusses the motivation behind this research and distinguishes this work from the other works in literature. Section 5.2 explains the proposed message matching approach. The experimental results are presented in Section 5.3. Finally, Section 5.4 concludes the chapter.

5.1 Motivation and Related Work

Improving the message matching performance for collective communication operations is only useful if they have considerable contribution in posting elements to the message queues.

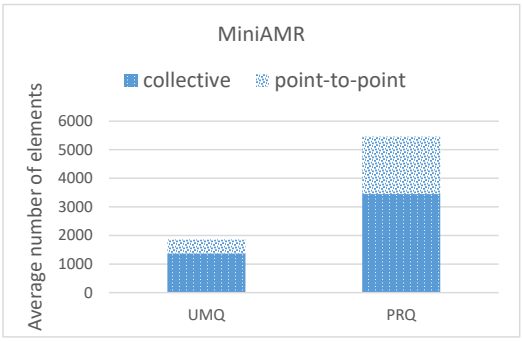
In order to understand if improving message matching performance for collective communications is useful, we profile several applications to understand their matching characteristics. In this experiment, we count the number of elements that enter the queues from point-to-point communication. We also count the number of elements that enter the queues from collective communications. For this, we provide a hint from MPI layer to the device layer to indicate whether the incoming message is from a point-to-point or collective communication. We use the experimental platform with Cluster A described in Section 3.4.1 for this test.

Figure 5.1 shows the application results for Radix [64], Nbody [103, 22], MiniAMR [9] and FDS [89] with 512 processes. The descriptions of these applications are explained in Section 5.3.1. As can be seen in Figure 5.1(a), almost all the elements that enter the queues in Radix are from collective communications. Figure 5.1(b) shows that the majority of the elements that enter the queues in Nbody are from point-to-point communications but that it still has a significant number of elements from collective communications (around 11k and 25k for UMQ and PRQ, respectively). As can be seen in Figure 5.1(c) and 5.1(d), both point-to-point and collective communications have some contributions to the queue length of MiniAMR and FDS. In general, Figure 5.1 shows that both collective and point-to-point communications can have considerable impact on the number of elements posted to the message queues. On the other hand, the list searches of $> 1k$ have significant impact on message latency [25]. This shows the importance of improving the message matching performance for collective and point-to-point communications and motivates us to propose a unified message queue design for collective and point-to-point operation. For collective communications, we propose a message queue design that dynamically profiles the impact of the collective communications on the queues and uses that information to adapt the message queue data structure for each and every collective communication. We then use the partner/non-partner message queue design discussed in Chapter 4 for elements coming from point-to-point communications.

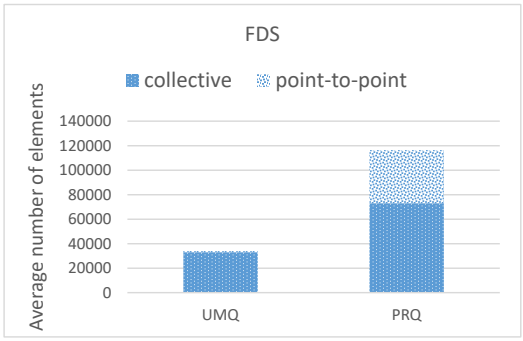


(a) Radix

(b) Nbody



(c) MiniAMR



(d) FDS

Figure 5.1: Average number of elements in the queues from collective and point-to-point communications across all processes in different applications (512 processes) in Cluster A

Note that the message queue mechanisms that are used in current well-known MPI implementations, such as MPICH, MVAPICH and Open MPI, or are proposed in the literature [116, 46, 71, 27, 48] do not consider the type of communication for message matching, and therefore they keep the messages from all types of communication in a single data structure.

5.2 The Proposed Unified Message Queue Design

Figure 5.2 shows the proposed unified message queue design. Whenever a new element wants to be added to the queue, we check if the element is coming from a point-to-point or a collective communication. If the element is coming from point-to-point communication, we use the partner/non-partner message queue data structure which we call it the *PNP*

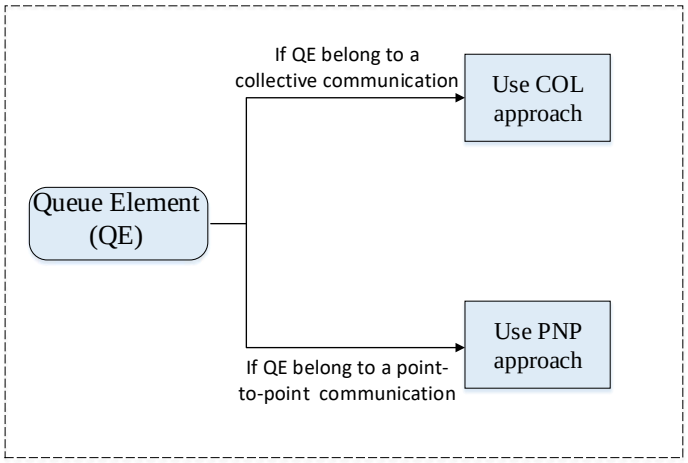


Figure 5.2: The proposed unified message matching mechanism

approach in this chapter. Otherwise, if the element is coming from a collective operation, we use the proposed message queue design for collective elements which is referred to as the *COL* approach. We refer to the unified design as COL+PNP. The PNP approach is discussed in detail in Chapter 4. We discuss the proposed COL message queue design for collective elements in Section 5.2.1.

5.2.1 The COL Message Queue Design for Collective Elements

There are many different algorithms (such as ring, binomial tree, fan-in/fan-out, etc.) proposed in literature or in MPI libraries for collective operations. For each collective operation, the choice of the algorithm depends on parameters such as message size and communicator size. Each collective communication algorithm has a specific impact on the behavior of message queues. We take advantage of this feature to design a message matching mechanism that adapts itself to the impact of collective communication algorithms on message queues. Figure 5.3 shows the proposed message queue mechanism for collective communications.

An overview of the design can be summarized as follows:

- There is a runtime profiling stage that determines the number of queues for each

collective communication operation with their specific parameters (message size and communicator size). At the profiling stage, all the collective operations share a single *profiling queue (pq)*.

- The profiling queue, pq, is only used for the first call of each collective operation with specific parameters. After that, each collective operation generates its own set of queues.

- The queues allocated to each collective operation could be defined in multiple levels. At each level, a hashing approach based on the number of queues is used for message matching.

- A new level is defined if two conditions are met: First, a collective operation is called with new parameters. Secondly, the required number of queues for this collective is more than the number of the queues that are already allocated for the collective.

As can be seen in this figure, the profiling queue is used for the first call of each collective operation with specific parameters. The information from profiling queue is used to determine the number of queues that are deemed sufficient to have the minimum queue search time for each collective operation. For example, in Figure 5.3, q_1 number of queues are allocated for MPI_Allreduce in the first level.

If the same collective is called with different parameters, we again profile its message queue behavior to calculate the required number of queues (q_2). If q_2 was larger than q_1, it means that the queues that are currently allocated in Level 1 are not sufficient for the new collective operation. Therefore, we define a set of q_2 queues in a new level. This procedure is continued as long as the collective operation is used with the new parameters or until we are limited by the memory consumption cap. The same procedure is used for other collective operations including both blocking and non-blocking collectives such as MPI_Gather, MPI_Iallgather, etc. Note that each collective operation uses specific tags

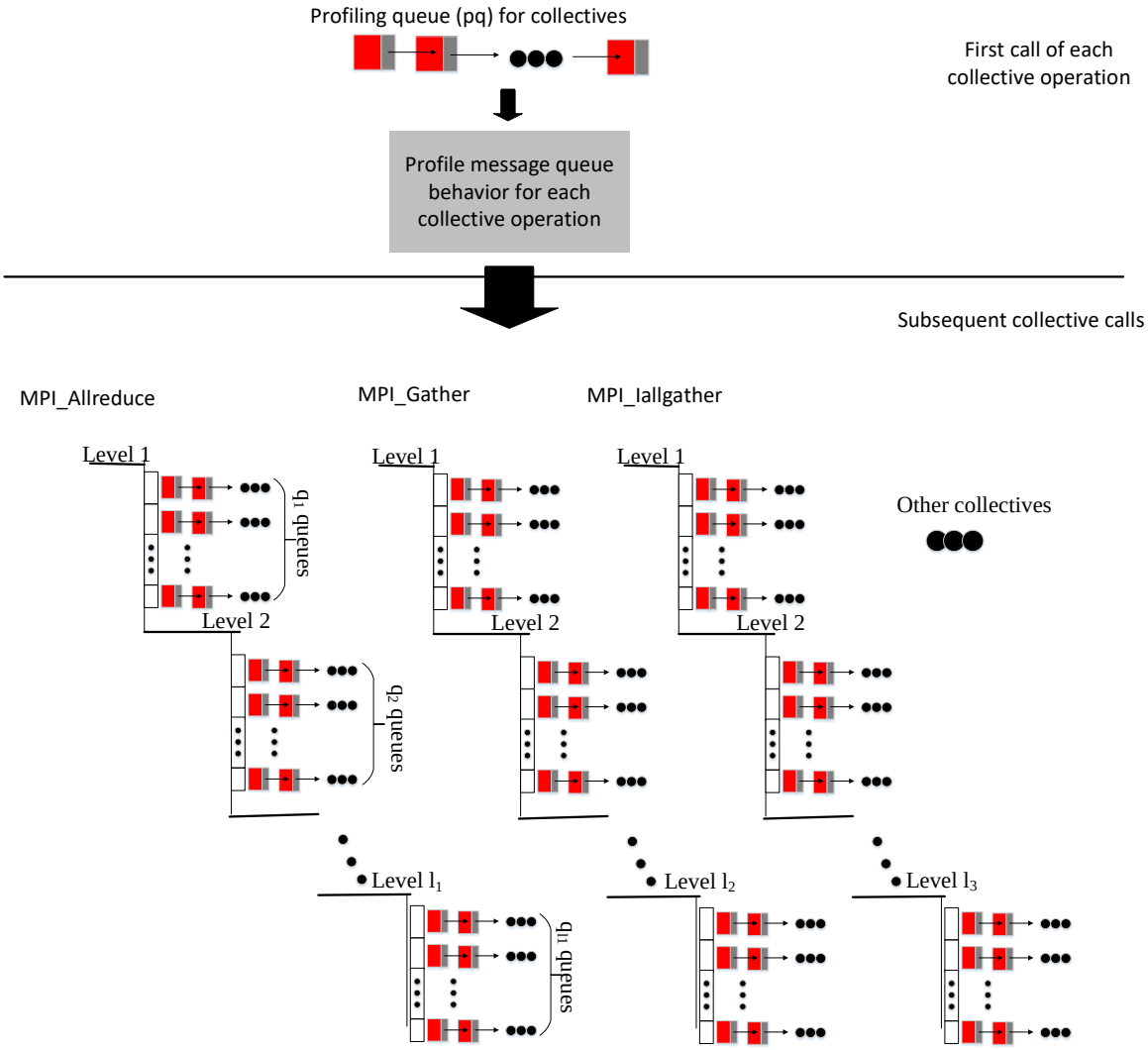


Figure 5.3: Proposed message matching mechanism for collective elements

for message matching. Therefore allocating dedicated queues for each collective operation automatically creates dedicated channels for individual tags.

For each collective communication operation, we always insert the new queue elements to the last level. For searching an element that is originated from collective communication, we always start from the profiling queue and then search the dedicated queues for the collective operation from the first level to the last level in order. This mechanism assures that message matching ordering semantics are preserved. We explain the queue allocation

and the search mechanism in more details in the following sections.

5.2.2 The Unified Queue Allocation Mechanism for Collective and Point-to-Point Elements

Algorithm 5.1 shows the detailed description of the queue allocation mechanism in the proposed design. Table 5.1 lists the parameters that are used as inputs and outputs of the algorithm and provides their definitions.

In the unified algorithm, we first check the type of communication. If it is a point-to-point communication, we add the element to the partner/non-partner message queue data structure in Line 2. Otherwise, when a collective communication is executed, we call the function *is_profiled(p)* (Line 4). This function determines if the collective operation with specific message size and communicator size has already been profiled or not. If it has not been profiled, we profile its message queue behavior and save the profiling information in P_p (Line 5). We use the average number of queue traversals as the profiling information since it is the critical factor that determines the cost of message matching [27]. If the collective operation has already been profiled, we call the function *is_q_allocated(p)* in Line 7. This function determines if queues have already been allocated for the collective operation with this specific message size and communicator size range. If queue is not allocated, we call the function *calcul_num_queues* (P_p) in Line 8. This function gets the profiling information gathered in the previous call of the collective operation and returns the required number of queues (nq).

In the best-case scenario, the average number of traversals to find an element is one. For this to happen, the number of queues should be equal to the average number of traversals. However, this may come at the expense of large memory allocation if the number of traversals is significant. Therefore, we limit the total number of queues allocated for all collective and point-to-point operations. For choosing the cap for the number of queues, we considered the memory consumption in MPICH and Open MPI. MPICH provides scalable

Table 5.1: List of parameters used for collective queue allocation and search mechanism

p	Collective operation parameters (the type of collective operation, its message size and communicator size)
pq	The profiling queue for collective operations
T	The total number of dedicated queues for all collectives
l_c	The number of levels for collective operation c
nq_c	Number of queues for collective c in the last level
n	Total number of processes
k	Memory consumption cap parameter
q_c	The set of queues in the last level for collective c
t	The type of communication
SE	The searching element
Q_{PNP}	The queue data structure for point-to-point elements
nq_{cl}	The number of queues that are allocated for collective operation c at level l
q_{cl}	The set of queues that are allocated for collective operation c at level l
QE	The matched element in the queue

memory consumption but it allocates only one queue for message matching, resulting in poor search performance for long list traversals. On the other hand, as discussed in Chapter 2, Open MPI is faster than MPICH for long match lists but it has unscalable memory consumption as it allocates n queues for each process. In our design, we take an in-between approach and bound the total number of queues that are allocated in the COL and PNP approaches to $k \times \sqrt{n}$. k is an environment variable to evaluate the impact of increasing the memory cap on message matching performance.

If the number of queues, nq, plus the total number of queues, T, that are already allocated, was less than $k \times \sqrt{n}$ (Line 9), it means that we are still allowed to allocate the new queues, and so we will check the second condition in Line 10.

The second condition compares nq with the number of queues that are currently allocated for collective operation c in the last level (nq_c). If nq was less than nq_c, there is no need to define a new level and allocate a new set of queues since nq_c number of queues is sufficient for this collective. However, if nq was greater than nq_c, the new set of queues should be allocated in a new level (Line 11). Finally, we update T, l_c and nq_c in Line 12 to

Algorithm 5.1: The Unified Queue Allocation Mechanism for Collective and Point-to-Point Elements

Input: The communication type (t), The collective operation parameters (p), The queue for profiling collective communications (pq), Total number of the allocated queues for collective operations (T), Number of levels for collective operation c (l_c), The number of queues that are allocated for collective operation c in the last level (nq_c), Total number of processes (n), The memory consumption cap parameter (k)

Output: The set of queues generated in the last level of collective operation c (q_c)

```
1  if t==point-to-point then
2      Add the element to Q_PNP;
3  else
4      if is_profiled(p)==0 then
5          P_p= profile(pq);
6      else
7          if is_q_allocated(p)==0 then
8              nq=calcul_num_queues(P_p);
9              if nq + T < k × √n then
10                 if nq > nq_c then
11                     Generate queues q_c[0 ... nq − 1];
12                     T = T + nq;
13                     l_c + +;
14                     nq_c = nq;
15                 end
16             end
17         end
18     end
19 end
```

14.

5.2.3 The Unified Search Mechanism for Collective and Point-to-Point Elements

Algorithm 5.2 shows the search mechanism in the proposed unified message queue design. A brief description of the inputs and outputs of the algorithm is provided in Table 5.1. The inputs of this algorithm are as follows: the type of communication (t), whether it is point-to-point or collective. If the communication was collective, the parameter c determines the

Algorithm 5.2: The Unified Queue Search Mechanism for Collective and Point-to-Point Elements

Input: The communication type (t), The collective operation (c), The searching element (SE), The queue data structure for point-to-point elements (Q_{PNP}), The queue for profiling collective communications (pq), The number of levels for collective operation c (l_c), The number of queues that are allocated for collective operation c at level l (nq_{cl}), the set of queues that are allocated for collective operation c at level l (q_{cl})

Output: The result of the search (QE)

```
1  if t==point-to-point then
2      QE=Search Q_PNP for SE;
3      Return QE;
4  else
5      Search pq;
6      for i = 1 to l_c
7       do
8          q_num= extract_queue_number(SE, nq_ci);
9          QE=Search q_ci[q_num] for SE;
10     end
11     Return QE;
12 end
```

type of collective operation. The parameters (c and t) are ported from the MPI layer to the device layer. Other inputs of the algorithm are the searching element (SE) which is the tuple rank, tag and communicator, the queue data structure for point-to-point elements (Q_{PNP}), the profiling queue for collective operations (pq), the number of levels for collective operation c (l_c), and the number of queues that are allocated for collective operation c at level l (nq_{cl}). The output of the algorithm is the search result (QE). If the element is not found, QE will be null.

At the time of searching, we first check the type of the communication in Line 1. If it is a point-to-point communication, the partner/non-partner queue data structure for point-to-point elements (Q_{PNP}) is searched (Line 2). If the message originated from a collective operation, we search the profiling queue pq since it might have some elements (Line 5). Then we search the queues allocated for this collective operation from the first level to the

last level in order (Lines 6 to 10). Each level consists of a specific number of queues (nq_{ci}), and the queue elements are enqueued using a hashing approach. For searching each level, first we call the function *extract_queue_ number* (SE, nq_{ci}) which takes the search element and nq_{ci} as input and returns the queue number for the search element. For this, it simply divides the rank number of the searching element by the number of queues nq_{ci} and returns the remainder as the output.

5.3 Experimental Results and Analysis

In this section, we first describe the experimental setup. We then evaluate the impact of the proposed COL+PNP approach on the performance of some blocking and non-blocking collective operations including MPI_ Gather, MPI_Allreduce and MPI_Iallgather in Section 5.3.2. In Section 5.3.3, we present and analyze the queue search time speedup of the proposed COL+PNP message matching approach on four real applications and compare them against the COL+LL approach. In COL+LL, the COL approach is used for collective elements and a single linked list queue is used for point-to-point elements. Section 5.3.4 presents the number of dedicated queues in both collective and point-to-point approaches. Finally, Section 5.3.5 and Section 5.3.6 discuss the application runtime speedup and the overhead of the proposed COL+PNP approach, respectively.

5.3.1 Experimental Platform

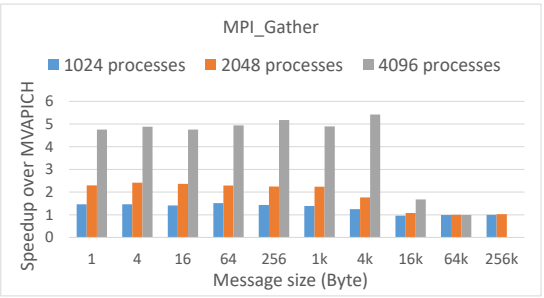
The evaluation was conducted on two clusters. The first cluster is Cluster A in which its specifications have been discussed in Section 3.4.1. The second cluster is the Graham cluster from Compute Canada. Graham has 924 nodes, each having 32 Intel E5-2683 V4 CPU cores, running at 2.1GHz. We use 1G memory per core in our experiments. The network interconnect is EDR Infiniband. We refer to the Graham cluster as Cluster B in this dissertation. The MPI implementation is MVAPICH2-2.2.

The applications that we use for the experiments are Radix [64], Nbody [103, 22], MiniAMR [9] and FDS [89]. Radix is an efficient and practical algorithm for sorting numerical keys which is used in different areas such as computer graphics, database systems, and sparse matrix multiplication. Nbody is a simulation of a dynamical system of particles, usually under the influence of physical forces, such as gravity. MiniAMR is a 3D stencil calculation with adaptive mesh refinement. We use MiniAMR's default mesh refinement options for the experiments. The FDS specification is presented in Section 3.4.1. All the application results are averaged across the entire application runtime. Note that we have used these applications since they have different message queue behavior in terms of the number of point-to-point and collective elements in the queue. For example, in Radix, almost all the elements in the queue are from collectives (Figure 5.1(a)). In Nbody, most of the elements in the queue are from point-to-point communications (Figure 5.1(b)). Moreover, these applications span short list traversals (Radix and MiniAMR) to long list traversals (FDS). This provides us with the opportunity to evaluate our design on applications with different message queue behavior.

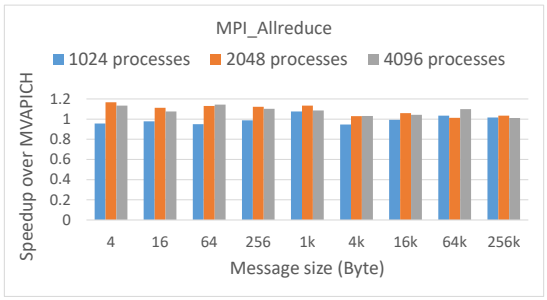
As discussed earlier, we bound the total memory overhead of the COL+PNP approach to $k \times \sqrt{n}$. We define $k = k_C + k_P$, where k_C represents the number of allocated queues in the COL approach, and k_P refers to the number of allocated queues in the PNP approach. This would allow us to evaluate the impact of memory consumption on the performance of both COL and PNP approaches, individually. Note that in the COL+LL approach, there is no memory overhead for point-to-point queue elements and $k = k_C$.

5.3.2 Microbenchmark Results

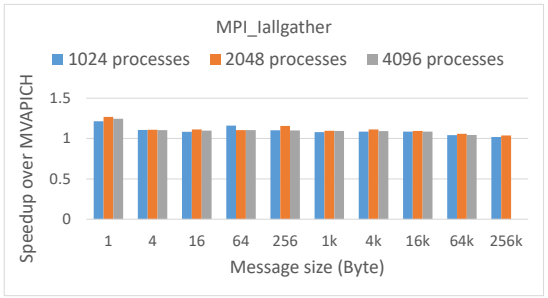
This section evaluates the impact of the proposed COL+PNP approach on the performance of some collective operations. Figure 5.4(a) shows that for MPI_Gather we can gain up to 1.5x, 2.4x and 5.4x latency reduction for 1024, 2048 and 4096 processes, respectively. In this collective operation, process 0 gathers data from all the other processes which result in long



(a) Latency improvement in MPI_Gather



(b) Latency improvement in MPI_Allreduce



(c) Latency improvement in MPI_Iallgather

Figure 5.4: Latency improvement in MPI_Gather, MPI_Allreduce and MPI_Iallgather, for k=1 on Cluster A

message queues for this process. Therefore, the proposed message matching mechanism generates as many queues as it can to reduce the queue search time for process 0. For example, the number of PRQs that are generated for 1024, 2048 and 4096 processes is 32 ($\sqrt{1024}$), 45 ($\sqrt{2048}$) and 64 ($\sqrt{4096}$), respectively. In other words, process 0 reaches the memory consumption cap for the number of queues for these message sizes. Other processes generate only a few queues (around 1 or 2) as their queue length is small. Figure 5.4(b) and 5.4(c) show that we can gain up to 1.16x and 1.26x latency reduction for MPI_Allreduce and MPI_Iallgather, respectively. The queues in these collective operations are not as long as MPI_Gather. Therefore, around 10 to 20 queues will be enough for them to gain this speedup.

One observation from Figure 5.4 is that the performance improvement decreases with increasing message size. The reason for this is that as we increase the message size, the

network's data transfer speed becomes the bottleneck rather than message matching performance.

5.3.3 Application Queue Search Time

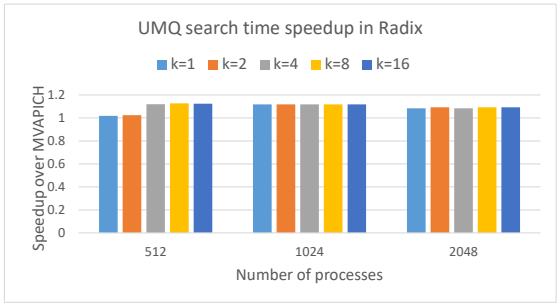
In this section, first we discuss the queue search time performance for collective elements in the COL+LL approach with different memory consumption cap parameter k. This allows us to evaluate the impact of different k values on the performance. In the COL+LL approach, the COL approach discussed in Section 5.2.1 is used for queue elements coming from collective communications and a single linked list queue is used for point-to-point communications. Then, we discuss the impact of the COL+PNP approach as well as the COL+LL approach on the performance of point-to-point elements, collective elements and all the elements in the queue. All the results are compared to the linked list data structure for Radix, Nbody, MiniAMR and FDS.

COL+LL impact on Collective Queue Search Time

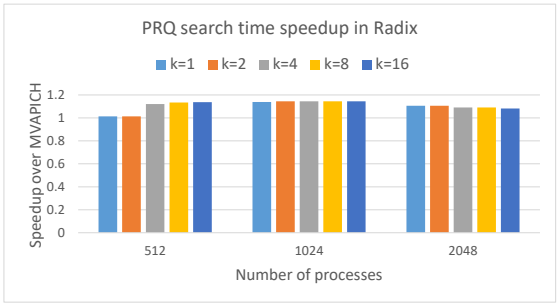
Figure 5.5 presents the queue search time speedup of the COL +LL approach on the queue search time of the collective elements. It also evaluates the impact of different memory cap parameter k on the queue search time speedup. The results in Figure 5.5 are conducted on Cluster A.

Figures 5.5(a) and 5.5(b) show the average UMQ and PRQ search time speedup across all processes for Radix, respectively. As can be seen in these figures, we can gain up to 1.12x and 1.14x search time speedup for UMQ and PRQ of Radix, respectively. Increasing k does not impact the queue search time significantly. That is because of the short list traversals of the queues (around 10 elements) for this application that make a small queue memory footprint sufficient to get a minor speedup. We observe almost the same behavior for Nbody in Figure 5.5(c) and 5.5(d).

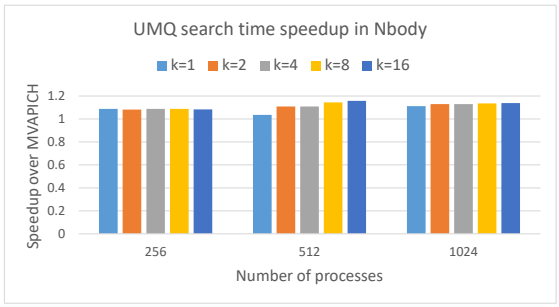
Figures 5.5(e) and 5.5(f) show the average queue search time speedup for collective



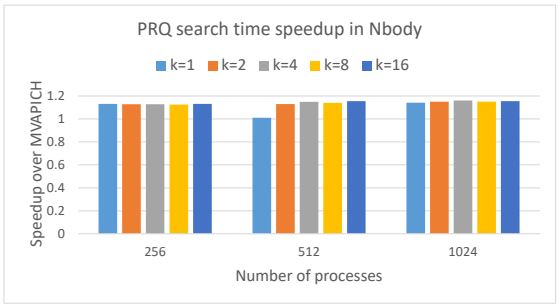
(a) UMQ in Radix



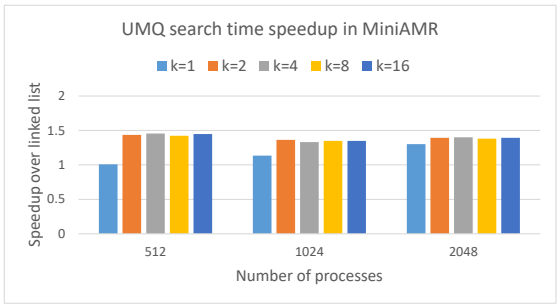
(b) PRQ in Radix



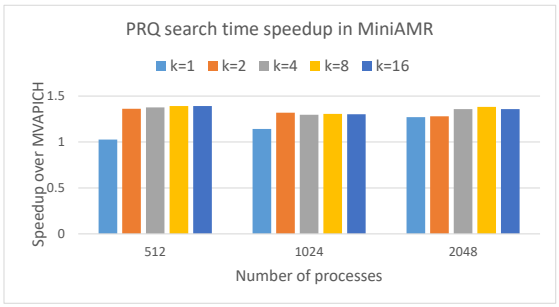
(c) UMQ in Nbody



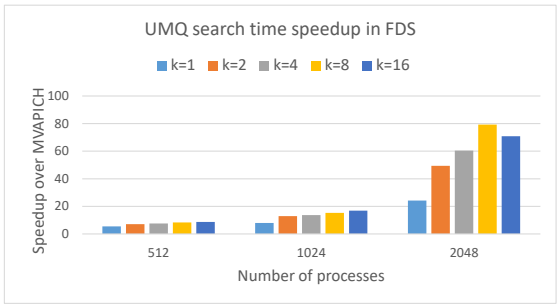
(d) PRQ in Nbody



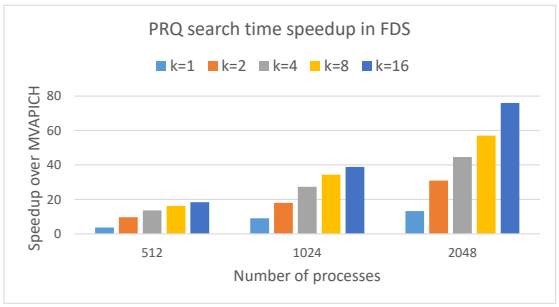
(e) UMQ in MiniAMR



(f) PRQ in MiniAMR



(g) UMQ in FDS



(h) PRQ in FDS

Figure 5.5: Average UMQ and PRQ search time speedup for collective elements with COL+LL approach and different k values in Radix, Nbody, MiniAMR and FDS in Cluster A

communications across all processes in MiniAMR. In this application, we can gain up to 1.45x and 1.4x search time speedup for UMQ and PRQ, respectively. As can be seen in this figure, by increasing k from 1 to 2 (or doubling the memory consumption), we can improve the queue search time speedup. However, increasing k further does not have considerable impact on queue search time. For example, with 512 processes, increasing k from 1 to 2 improves the search time speedup around 40% for both UMQ and PRQ. However, increasing it further does not improve the search time speedup considerably. This shows that for 512 processes, less than 44 ($2 \times \sqrt{512}$) queues is enough to gain the maximum search time speedup. We will discuss the number of generated queues for different number of processes in each application with more details in Section 5.3.4.

Figures 5.5(g) and 5.5(h) present the UMQ and PRQ search time speedup for collective communications in FDS. For this application, we show the search time speedup for process 0 since this process has the majority of communications. As can be seen in the figures, we can gain around 80x queue search time speedup for collective queue elements in this application.

As discussed in previous chapters, in FDS, each process sends a number of messages to process 0 through MPI_Gather(v). This hotspot behavior places significant stress on the MPI matching engine. Therefore, FDS results show the potential maximum performance that can be gained by the proposed message matching mechanism. Moreover, they provide the opportunity to indirectly compare the performance gain of our approach with other message matching proposals that use this application [46]. Finally, these results show that with an MPI implementation that support long message queue traversals, we can provide the opportunity to the programmer to design less complicated code while maintaining high performance.

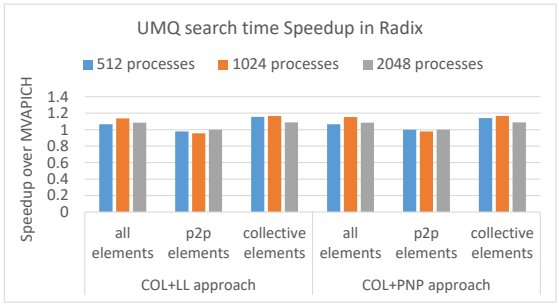
COL+PNP vs. COL+LL Impact on Collective, Point-to-point and Total Queue Search Time

Figure 5.6 shows the queue search time speedup of COL+PNP and COL+LL for Radix, Nbody, MiniAMR and FDS. We present the performance gain for all the elements in the queue, the point-to-point elements and also collective elements. The experiments in this section are conducted on Cluster B.

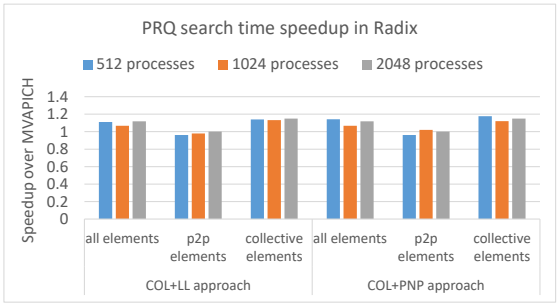
This figure shows the speedup when $k = k_C = 8$. The reason we choose $k_C = 8$ is that it provides a scalable memory consumption and at the same time it has the maximum performance gain in almost all cases as shown in Figure 5.5. Also, we choose $k_P = 8$ so as not to exceed k of 16 which is the maximum memory cap for 512 processes. We include the results for collective elements here to compare them with the point-to-point and total queue search time speedup on the same cluster. The threshold for the partner/non-partner design is $\theta = 100$ since it provides the maximum performance and scalable memory consumption as discussed in Chapter 4.

Figure 5.6(a) and 5.6(b) show the average UMQ and PRQ search time speedup for Radix, respectively. As can be seen in these figures, in both COL+LL and COL+PNP, the speedup for point-to-point elements is close to 1. The reason for this is that, in Radix, almost all the elements in the queue are coming from collective communications and there is no point-to-point element in the queue (Figure 5.1(a)). These figures also show that in total we can gain up to 1.15x and 1.14x search time speedup for all the elements in the queue. One observation from Figures 5.6(a) and 5.6(b) is that the performance of COL+LL and COL+PNP approaches is almost similar. In other words, the use of LL or PNP approaches for point-to-point elements does not affect the performance since there are just a few, if any, point-ro-point elements in the queue.

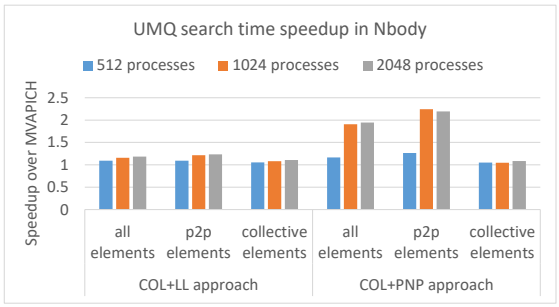
Figure 5.6(c) and 5.6(d) show the average UMQ and PRQ search time speedup for Nbody. In this application, there are a significant number of elements in the queue from



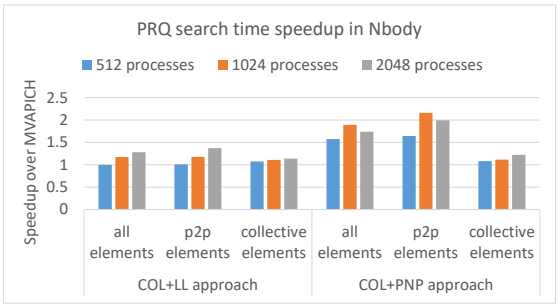
(a) UMQ in Radix



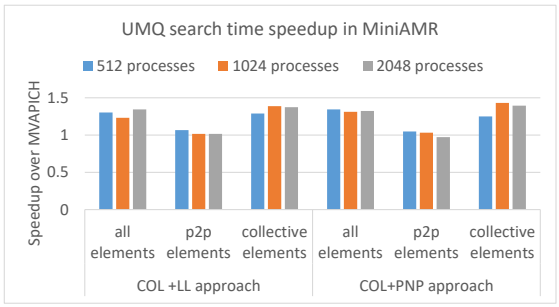
(b) PRQ in Radix



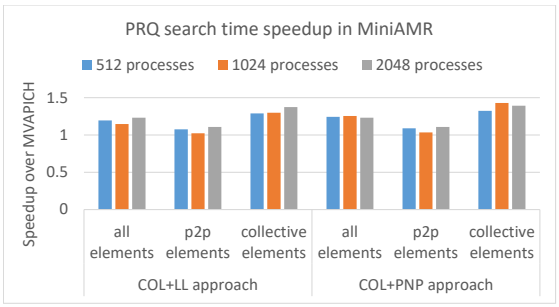
(c) UMQ in Nbody



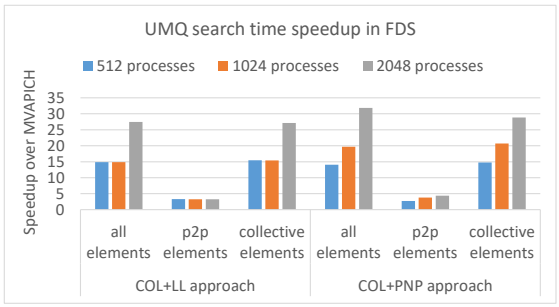
(d) PRQ in Nbody



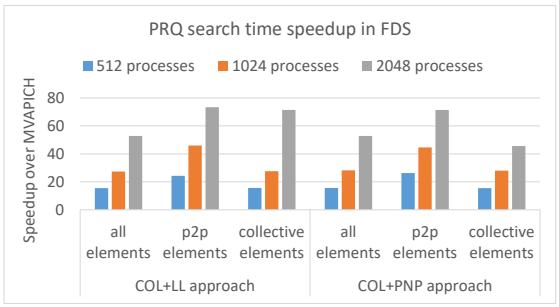
(e) UMQ in MiniAMR



(f) PRQ in MiniAMR



(g) UMQ in FDS



(h) PRQ in FDS

Figure 5.6: Average UMQ and PRQ search time speedup for collective, point-to-point and total elements with COL+PNP and COL+LL approaches and $k = 16$ in Radix, Nbody, MiniAMR and FDS in Cluster B

point-to-point communication (Figure 5.1(b)). When we separate the queue for collective elements in the COL+LL approach, the queue length for point-to-point elements is reduced and its queue search time improves by up to 1.23x and 1.37x in UMQ and PRQ, respectively. The total search time speedup for all elements in UMQ and PRQ is 1.18x and 1.27x, respectively. Using the partner/non-partner message queue design for point-to-point communication in COL+PNP approach further improves the queue search time. As can be seen in these figures, in the COL+PNP approach, we can gain up to 2.24x and 2.15x speedup in UMQ and PRQ search time of point-to-point elements, respectively. Comparing the COL+PNP results with the COL+LL results, we can observe that some part of the point-to-point speedup is because of separating point-to-point elements from collective elements. However, the majority of performance gain comes from the partner/non-partner message queue design.

Figure 5.6(e) and 5.6(f) show the queue search time speedup in MiniAMR. In this application, there are just a few number of point-to-point elements in the queue (Figure 5.1(c)). Therefore, the speedup for point-to-point elements is around 1 in both COL+LL and COL+PNP approaches. Consequently, the speedup of the COL+LL approach and COL+PNP approach is almost the same for collective elements as well as all the elements.

Finally, the COL+PNP results in Figure 5.6(g) and 5.6(h) show that we can gain up to 4.4x and 73x search time speedup for UMQ and PRQ point-to-point elements in FDS, respectively. Moreover, the COL+LL results show that by taking out the collective elements from the queue in FDS, the search time of point-to-point elements in UMQ and PRQ improves by up to 3.34x and 71x, respectively. The total UMQ and PRQ search time speedup in the COL+LL approach is 27x and 52x, respectively. As discussed in Section 5.3.3, this large performance gain is possible due to the long list traversals in this application. Comparing the COL+PNP results with the COL+LL results in Figure 5.6(g), we can observe that around 3.2x of UMQ speedup is because of separating the the point-to-point

elements from collectives and the rest of the performance gain is coming from partner/non-partner message queue design. On the other hand, comparing the COL+PNP results with the COL+LL results in Figure 5.6(h) shows that most of the performance gain for point-to-point PRQ messages is because of separating the collective and point-to-point elements. This shows that in PRQ, a significant number of collective elements should be traversed to search for a point-to-point message.

Note that the performance gain for all the elements in the queue has a direct relationship with the performance gain for point-to-point and collective elements. However, whether it is more due to point-to-point or collective performance depends on the distribution of these elements in each application. In Radix, almost all the elements in the queue are from collective communication (Figure 5.1(a)). Therefore, the queue search time speedup for all elements is roughly similar to the search time speedup for collective elements (Figure 5.6(a) and 5.6(b)). On the other hand, in Nbody, the number of point-to-point elements in the queue is more (Figure 5.1(b)). Therefore, the search time speedup for all elements is more dependent on search time speedup of point-to-point elements (Figure 5.6(c) and 5.6(d)). In MiniAMR and FDS, both point-to-point and collective elements contribute to the message queues. However, the contribution of collective elements is more (Figure 5.1(c) and 5.1(d)). As a result, in these applications, the performance gain for all elements is due to both point-to-point and collective message matching improvement, but it is mainly because of the collective speedup (Figure 5.6(e) to 5.6(h)).

5.3.4 Number of Dedicated Queues for the Applications in COL+PNP Approach

As discussed earlier, the memory consumption in the collective and point-to-point approach is directly related to the number of allocated queues in these data structures. Therefore, in this section, we present the number of dedicated queues for the applications studied in this chapter. The same parameters discussed in Section 5.3.3 are used for the experiments.

Figure 5.7 shows the number of allocated queues in Radix, Nbody, MiniAMR and FDS for collective and point-to-point communications with the COL+ PNP approach. In the following, we discuss the number of dedicated queues for collective and point-to-point communications, respectively.

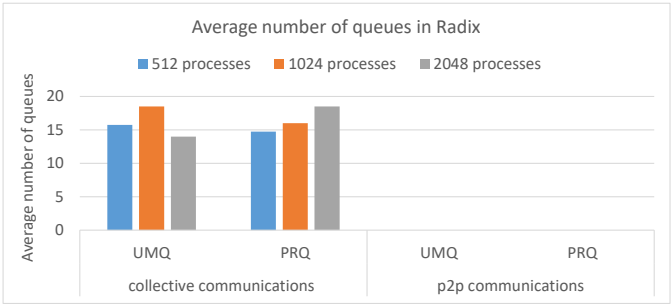
Number of Dedicated Queues for Collective Communications

Figure 5.7(a) shows the average number of queues across all processes in the Radix application. This application uses the collective operations MPI_Iallgather, MPI_Allreduce, MPI_Reduce and MPI_ Reduce_scatter. Among these collectives, MPI_Iallgather has the most contributions in generating long list traversals for this application, and around 65% of the queues for collective elements in Figure 5.7(a) are for this operation.

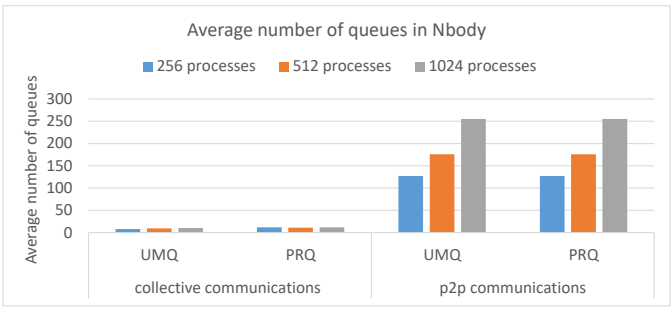
Figure 5.7(b) presents the average number of dedicated queues across all processes for Nbody. This application has the collective operations MPI_Allgather, MPI_Allreduce, MPI_Bcast and MPI_Reduce. Among these collectives, MPI_Allgather has the most contribution in generating long list traversals and thus, most of the dedicated queues for collective elements in Figure 5.7(b) belong to this collective. Other collectives either do not generate long message queues or they are called just a few times in the application.

Figure 5.7(c) shows the average number of generated UMQs and PRQs across all processes for MiniAMR with different number of processes. This application uses the collective operations MPI_Allreduce, MPI_Bcast, MPI_Allgather and MPI_Reduce. For this application, MPI_Allreduce has the most contribution in generating long list traversals. Therefore, most of the dedicated collective queues in Figure 5.7(c) belong to this operation.

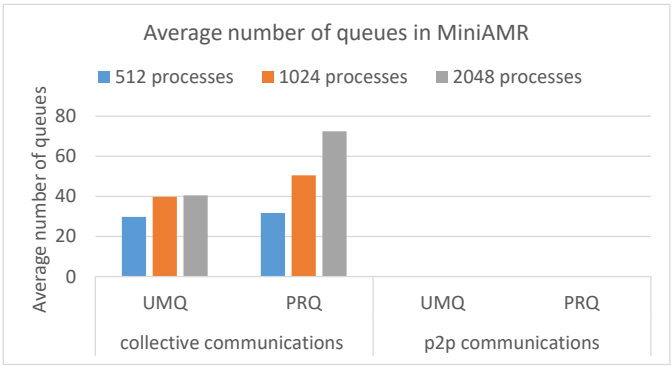
In Figures 5.7(d), we present the number of UMQs and PRQs that are generated in the FDS application. Here again, we show the results for rank 0 since this process has the majority of communications (as discussed in Section 5.3.3). This figure show that process 0 of FDS generates as many PRQs as it can for collective communications. For example, when the number of processes is 1024, 256 collective queues are generated which is the memory cap



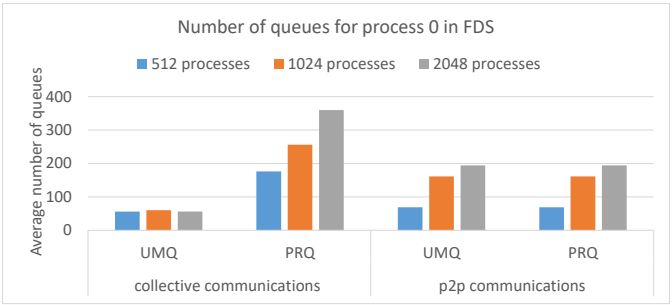
(a) Radix



(b) Nbody



(c) MiniAMR



(d) FDS

Figure 5.7: Number of dedicated UMQs and PRQs for collective operations with COL+PNP approach and $k = 16$ in Radix, Nbody, MiniAMR and FDS in Cluster B

for the number of queues ($8 \times \sqrt{1024} = 256$). Note that FDS uses the collective operations MPI_Gather, MPI_ Gatherv, MPI_Allgatherv, MPI_ Allreduce and the majority of the queues that are generated for process 0 belong to MPI_Gatherv.

Comparing Figure 5.7 with Figure 5.1 shows that the number of allocated queues for collective communications is in concert with the number of collective elements in the queues for each application. For example, Nbody does not have significant number of collective elements in both UMQ and PRQ (Figure 5.1(b)), so the number of collective UMQ and PRQ allocated for this application is around 10 (Figure 5.7(b)). On the other hand, there are significant number of collective elements in PRQ of MiniAMR (Figure 5.1(c)) and the number of collective PRQ allocated for this application reaches 72 (Figure 5.7(c)). In general, Figure 5.7 shows that the applications with longer collective traversal such as MiniAMR and FDS have a larger number of collective queues compared to Radix and Nbody with short collective traversal.

Number of Dedicated Queues for Point-to-Point communications

Figure 5.7 shows that the number of generated queues for point-to-point communications in the COL+PNP approach directly relates to the number of point-to-point queue elements. For example, there are a few number of point-to-point elements in Radix and MiniAMR (Figure 5.1). Therefore, the queue length of the original linked list queue does not reach the threshold, θ, and the number of generated queues is 0. On the other hand, the number of point-to-point elements in Nbody and FDS is more significant. This causes the queue length to reach the threshold, θ, more frequently which results in more queues.

Comparing Figure 5.7 with Figure 5.6, we can observe that the number of the allocated queues in the proposed COL+PNP message matching design is in concert with the queue search time speedup for both collective and point-to-point communications. One observation from Figure 5.7 is that in many cases, the number of allocated queues increases with increasing number of processes. However, it never exceeds the memory consumption cap

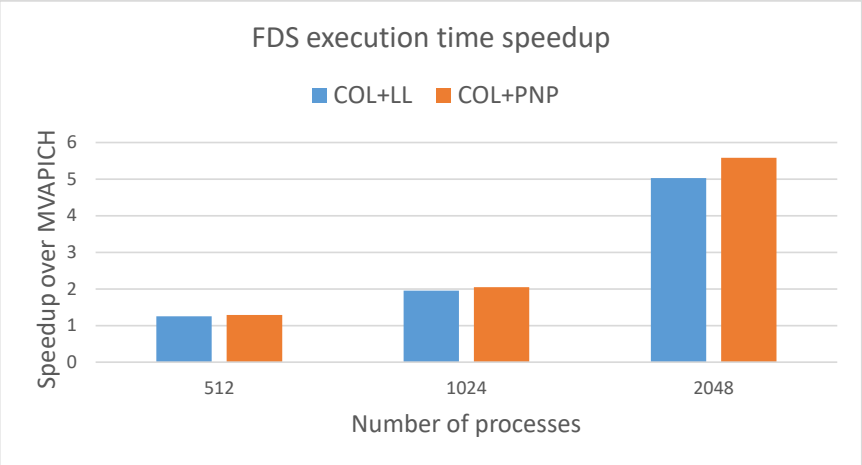


Figure 5.8: FDS runtime speedup over MVAPICH in Cluster B

$k \times \sqrt{n}$. The number of dedicated queues is limited to a few queues for applications with short list traversals, up to the max $k \times \sqrt{n}$ for applications with long list traversals. This shows the scalability of the proposed approach in terms of memory consumption.

5.3.5 Application Execution Time with COL+PNP and CPL+LL approaches

Figure 5.8 shows the FDS runtime in the proposed COL+PNP approach and compares it against the COL+LL and MVAPICH2 message queue designs. The results show that we can gain up to 5x runtime speedup with COL+LL. This improvement comes from two factors: 1) message matching speedup for point-to-point elements since they are separated from collective elements; and 2) message matching speedup for collective elements. In Section 5.3.3, we discussed the impact of each of these factors on total search time improvement. The orange bar shows that we can gain up to 5.5x runtime speedup with COL+PNP approach. We can observe that using partner/non-partner message queue design for point-to-point elements can further improve the performance gain.

The results in this figure follows the results in Figure 5.6(g) and 5.6(h) and also Figure 5.7(d). As more queues are generated (Figure 5.7(d)), the UMQ and PRQ search time speedup improves (Figure 5.6(g) and 5.6(h)) and consequently, the FDS execution time

Table 5.2: Overhead/search time ratio in COL+PNP

Applications	Number of processes	Overhead/search time ratio
Radix	512	0.0359
	1024	0.03027
	2048	0.0175
Nbody	256	0.1563
	512	0.0589
	1024	0.0343
MiniAMR	512	0.0192
	1024	0.0366
	2048	0.0138
FDS	512	0.0027
	1024	0.0021
	2048	0.0020

speedup increases (Figure 5.8). Note that we do not show the results for Radix, Nbody and MiniAMR since their queue search time speedup does not translate to considerable improvement in their application execution time (their runtime speedup is around 1).

5.3.6 Runtime Overhead of the Message Queue Design

The COL approach imposes some runtime overhead for calculating the required number of queues for each collective and allocating the queues. On the other hand, the PNP approach has some overhead for extracting the partners. Table 5.2 presents the ratio of the average runtime overhead of COL+PNP approach across all processes over the average queue search time across all processes in Radix, Nbody, MiniAMR and FDS for different number of processes. The results show that for all applications the overhead of the proposed design is negligible compared to their queue search time.

5.4 Summary

In this chapter, we propose unified COL+PNP message matching mechanism that considers the type of communication to improve the queue search time performance. For this, we separate the queue elements based on their type of communication (point-to-point or

collective). For collective operations, we dynamically profile the impact of each collective call on message queue traversals and use this information to adapt the message queue data structure (the COL approach). For the point-to-point queue elements, we use the partner/non-partner message queue data structure discussed in Chapter 4 (the PNP approach). The proposed approach can improve the message matching performance while maintaining a scalable number of queues (memory consumption).

Our experimental evaluation shows that by allocating 194 queues for point-to-point elements and 416 queues for collective elements, we can gain $5.5x$ runtime speedup for 2048 processes in applications with long list traversals. For applications with medium list traversals such as MiniAMR, it allocates the maximum of 74 queues for 2048 processes to reduce the queue search time of collective communications by 40%. We also compare the COL+PNP performance gain with COL+LL. In the COL+LL approach, the COL approach is used for collective communications and a single linked list queue is used for point-to-point communications. The evaluation results show that the COL+PNP approach provides similar or better performance compared to COL+LL. However, the performance gap between these two approaches depends on the number of point-to-point and collective elements in each application.

In chapter 6, we will look at the message matching issue from a different perspective and investigate mechanisms to take advantage of hardware features to improve the matching performance.

Chapter 6

Message Queue Matching Improvement on Modern Architectures

So far, we have proposed mechanisms to improve the message matching performance considering the communication characteristics of the applications. In this chapter, we investigate some techniques to take advantage of the properties of the new many-core processors with the aim of improving the performance of the message queue operations. Note that the approaches in this chapter are orthogonal with proposed message queue designs in previous chapters.

Many-core processors and coprocessors such as Intel Xeon Phi [65, 66] are well-known because of their energy efficiency and massive parallelism. However, the MPI libraries are designed for traditional heavy-weight cores with large amount of serial compute power. Therefore, current message matching architectures or the ones proposed in literature are not efficient on many-core systems and their performance is significantly worse than on traditional systems. For example, recent studies have shown that system message rates that were previously bottlenecked by networking overheads are instead limited by compute core performance on many-core systems [25]. This shows that optimizing message matching on many-core systems is crucially important to obtain scalability on future machines.

In this chapter, we consider two techniques to take advantage of vectorization capabilities

on modern CPU architectures to enhance message matching performance [40, 41] [1]. Note that some of the techniques in this chapter can be applied to both heavyweight cores such as Intel Xeon processors as well as Xeon Phi lightweight cores.

The rest of this chapter is organized as follows. Section 6.1 discusses the motivation behind this research and distinguishes it from the other works in literature. Section 6.2 discusses techniques to improve MPI message matching. Section 6.3 presents the complexity analysis. The experimental results are presented in Section 6.4. Finally, Section 6.5 concludes the chapter.

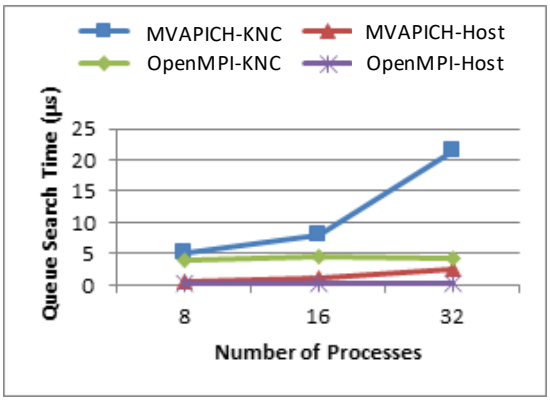
6.1 Motivation and Related Work

In this section, we conduct an experiment to compare the message matching performance on many-core systems with traditional heavy-weight cores. We consider message queue data structures that are used in current MPI libraries: Open MPI data structure and linked list data structures used in MVAPICH.

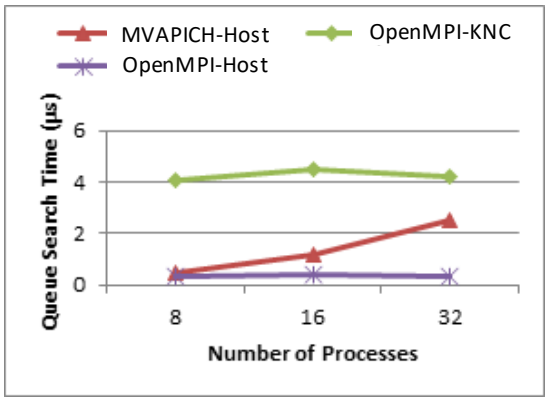
To this aim, we use a microbenchmark that fills in the UMQ with a specific number of elements. In this microbenchmark, each process (P_1 to P_{N-1}) sends M data to P_0 to fill in its UMQ. Then P_1 sends a new data to P_0 with a specific tag. We measure the queue search time for the very last element in the queue that was sent by P_1. We use the MPI_Barrier function to synchronize the ordering of messages sent by different processes and to make sure that the data sent by P_1 is the last element in the queue. Each test is averaged over 5 iterations. The experiments are conducted on Cluster C. The detailed description of Cluster C and the experimental setup is presented in Section 6.4.1.

Figure 6.1(a) shows the queue search time in different data structures (Open MPI and MVAPICH2) and hardware (Intel Xeon as host, and Xeon Phi KNC) when $M = 10$. Since the difference in queue search time in OpenMPI-Host, MVAPICH-Host and OpenMPI-KNC

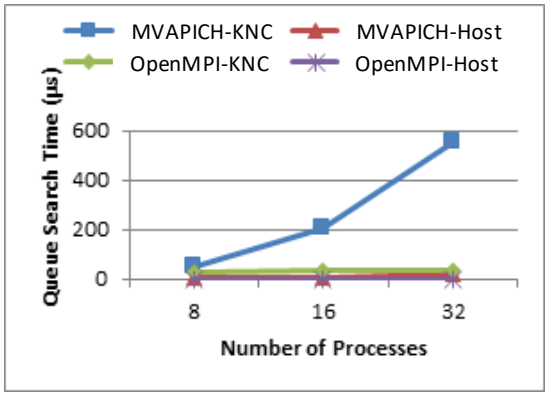
[1] These papers are done collaboratively with Sandia National Laboratories and the University of New Mexico. The hot caching technique and some of the results in this paper are developed by Dr. Dosanjh so they are not claimed and presented in this chapter.



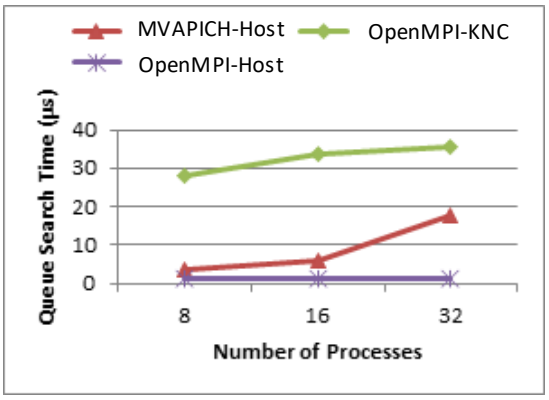
(a) M=10



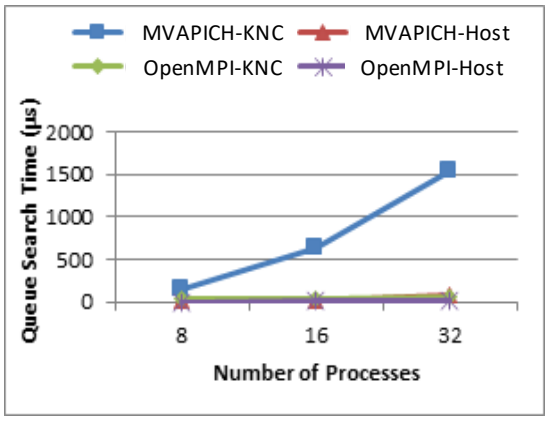
(b) M=10 (enlarged of (a))



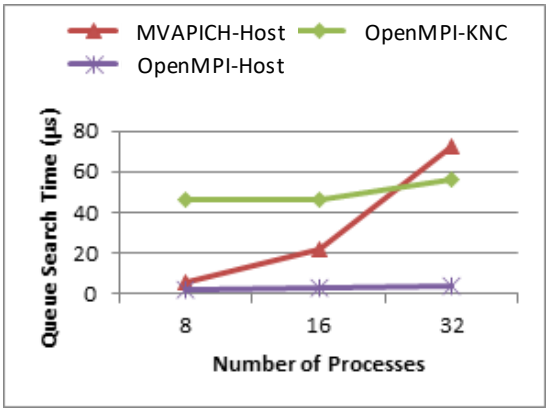
(c) M=50



(d) M=50 (enlarged of (c))



(e) M=100



(f) M=100 (enlarged of (e))

Figure 6.1: Queue search time in different data structures (MVAPICH2 and OPEN MPI) and hardwares (Intel Xeon as host, and Xeon Phi KNC) on cluster C

is not clear in Figure 6.1(a), they are enlarged in Figure 6.1(b). As can be seen in these figures, in both MVAPICH and Open MPI data structures, the queue search time in the KNC is worse than the host. Another observation from these figures is that Open MPI data structure performs better than linked list data structure and its queue search time does not change a lot with increasing number of processes. The reason for this lies in message queue data structure in Open MPI. As discussed in Section 2.2.5, in Open MPI, each source process has its own queue for the tags; however, MVAPICH2 provides a linear queue for the entire source processes.

We repeat our tests with $M = 50$ and $M = 100$. Figure 6.1(c) to Figure 6.1(f) present almost the same trend as in Figure 6.1(a) and 6.1(b). However, the queue search time in these figures is longer since each process is sending more data to process 0 and consequently the queue length is longer. Based on these figures, we can confirm that the queue search time in Intel Xeon Phi KNC is worse than the host in both MVAPICH and Open MPI data structures. In order to improve the queue search time in KNC, in the following we will experiment with two techniques that are designed to utilize the vectorization capabilities available in Xeon Phi. It should be noted that none of the message queue mechanisms that are used in current well-known MPI implementations, such as MPICH, MVAPICH and Open MPI, or are proposed in the literature [116, 46, 71, 27, 48] take advantage of vectorization capabilities to improve message matching performance.

6.2 Techniques to Improve MPI Message Matching

In this section, we present two techniques for improving message matching. Section 6.2.1 presents a *linked list of arrays* matching architecture that attempts to better interact with the memory subsystem by combining multiple matching elements into a single linked list element. Section 6.2.2 expands on this idea by rearranging data into AVX vectors.

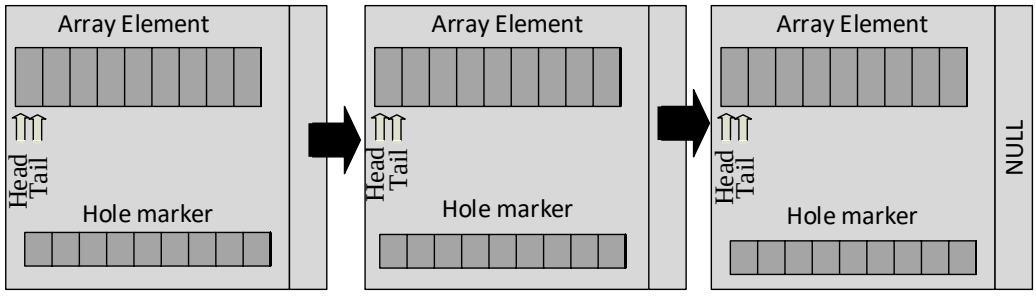


Figure 6.2: Linked list of arrays queue data structure

6.2.1 Linked List of Arrays Queue Data Structure

Figure 6.2 shows an overview of the linked list of array queue data structure used in this chapter. In this data structure, each linked list element contains multiple queue elements in an array. By using an array for representing the queue, the time for accessing each element in the queue is reduced and consequently the queue search time is improved. One issue with linked list of array design is that whenever a searching element is found, it should be removed from the queue. This makes an empty element or hole in the array. Therefore, we need a mechanism to keep track of these holes.

In the linked list of arrays design, we deal with this issue by using two pointers called *head* and *tail* pointers for each array as can be seen in Figure 6.2. The head points to the head of the array and the tail points to the bottom of the array. The elements are always added to the bottom of the array where the tail points to. This way, we can make sure that the sequence of the incoming messages is preserved, as required in MPI standard.

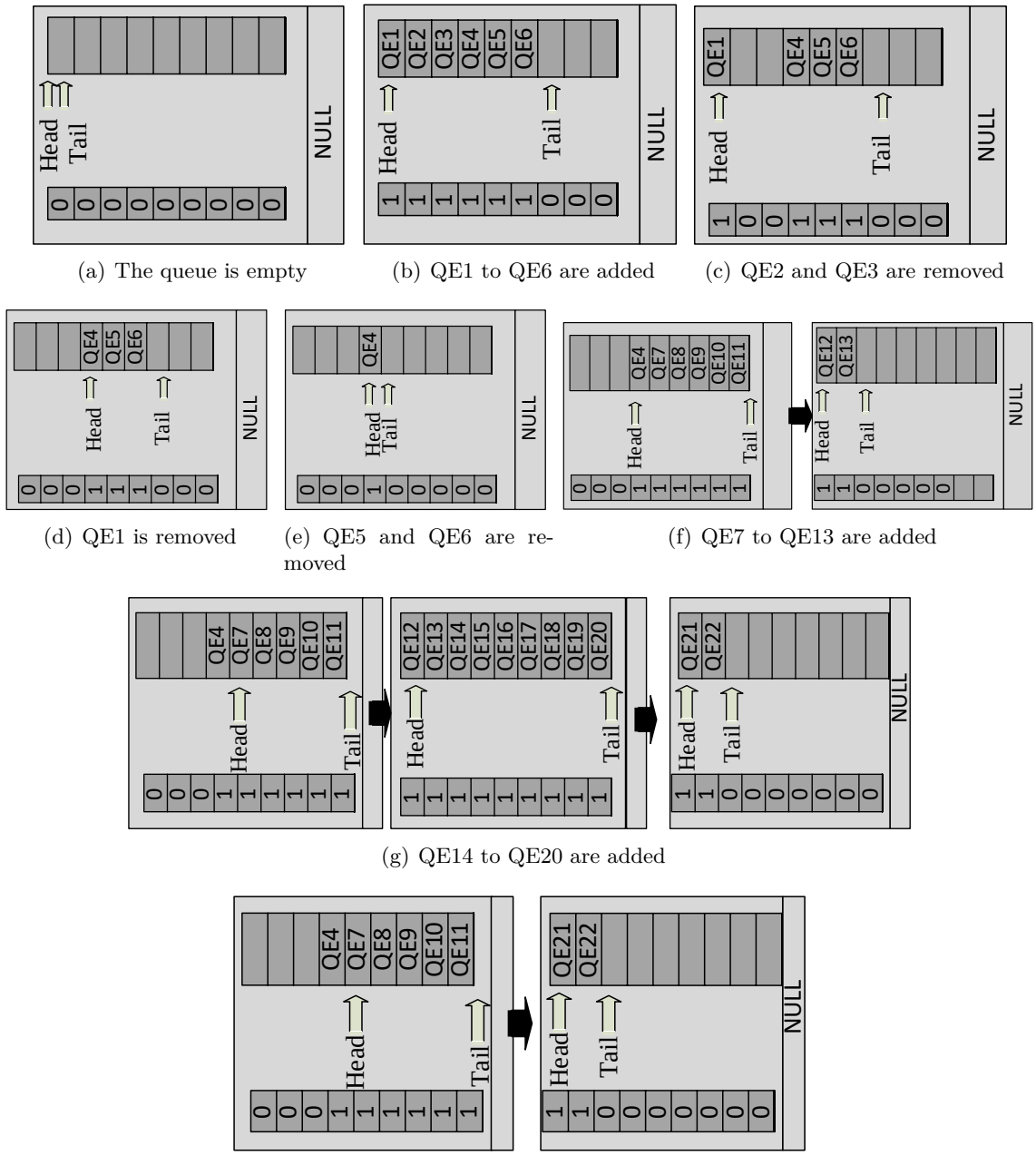
Figure 6.2 shows an array called *hole_marker* in the data structure. This is a boolean array to keep track of the empty elements. More specifically, it indicates whether the elements in the array are empty or not. At the beginning, the initial value of the array hole_marker is zero since the array is empty. As soon as one element is added to the array, the corresponding element in hole_marker becomes 1. This way, we can recognize the empty elements in the array. Whenever an element is removed from the array, if the

element is at the middle of the array, the head and tail pointers do not change; otherwise, if the removed element is at the end or head of the array, then the head or tail pointer is updated accordingly. The search starts from the head pointer to the tail pointer.

We refer to the array, the head and tail pointers, hole_marker and the linked list pointer as a *queue block* in the rest of this chapter. For example, Figure 6.2 shows three queue blocks in the data structure. Whenever the tail pointer of the last queue block reaches the end of the array, a new queue block is allocated for the incoming messages. Moreover, whenever one queue block becomes empty, it is removed from the linked list and the pointers of the linked list are updated accordingly.

Figure 6.3 shows how to manage the holes and incoming messages. In this simple example, assume the size of the array is 9. At first (Figure 6.3(a)), the queue is empty, the head and tail point to the first location in array and the elements in the hole_marker array are all zero. Moreover, the pointer to the next element of the linked list is null. In Figure 6.3(b), six elements are added to the queue. Therefore, the tail pointer and the corresponding bits in hole_marker are updated accordingly. In Figure 6.3(c), queue elements 2 and 3 are removed from the array and the corresponding bits in hole_marker are changed to zero. The head and tail pointer do not change at this stage. However, as soon as QE1 is removed from the queue in Figure 6.3(d), the head pointer is updated. When QE5 and QE6 are removed from the bottom of the queue, the tail pointer is updated (6.3(e)). When the queue block is full, we allocate a new queue block and add it to the linked list. In Figure 6.3(f), QE7 to QE13 are added to the queue so a new queue block in generated. In Figure 6.3(g), QE14 to QE22 are added to the queue. Therefore, the number of queue blocks is increased to 3. Figure 6.3(h) shows the case when all the QEs in an array are removed and corresponding linked list element is removed from the linked list. In order to manage the MPI wildcard communication, we use the same approach used in linked list data structure explained in Section 2.2.5.

The advantage of the linked list of arrays data structure is twofold. First, it has less



(a) The queue is empty

(b) QE1 to QE6 are added

(c) QE2 and QE3 are removed

(d) QE1 is removed

(e) QE5 and QE6 are removed

(f) QE7 to QE13 are added

(g) QE14 to QE20 are added

(h) QE12 to QE20 are removed

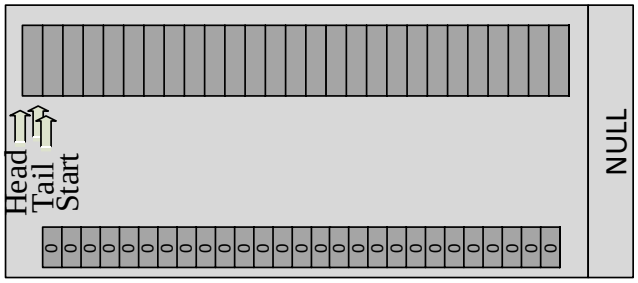
Figure 6.3: Managing the holes and incoming messages in the linked list of arrays queue data structure

memory reference compared to the linked list approach. In the linked list data structure, to access the next queue element we incur two memory references, one for reading the address of the next queue element and one for reading the queue element itself. However, in an array, the queue elements are located in contiguous memory locations so we only have one memory reference. Moreover, using linked list of arrays increases spatial locality which results in reducing the access time for searching the queue.

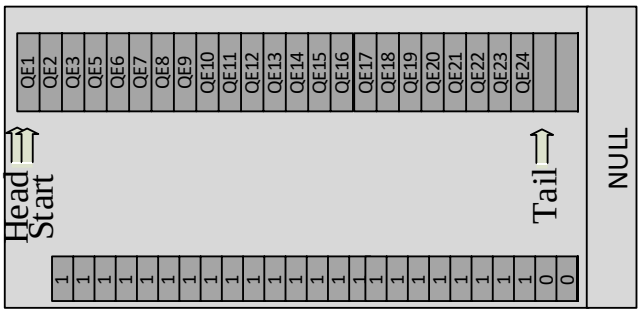
6.2.2 Linked List of Vectors Queue Data Structure

The performance of the linked list of arrays can be further improved by taking advantage of vector instructions (e.g., Intel AVX intrinsics). We refer to this approach as *linked list of vectors* queue design. In this approach, the queue elements are reorganized to be grouped per field into a vector block. The size of the vector block is architecture-dependent. For example, Intel Many Integrated Core (IMIC) instructions supported by Knights Corner Xeon Phi is 512-bit aligned. Therefore, assuming that each vector element is 32 bits integer, the size of vector block is 16. We refer to size of the vector block as *vec_size* in the rest of this chapter.

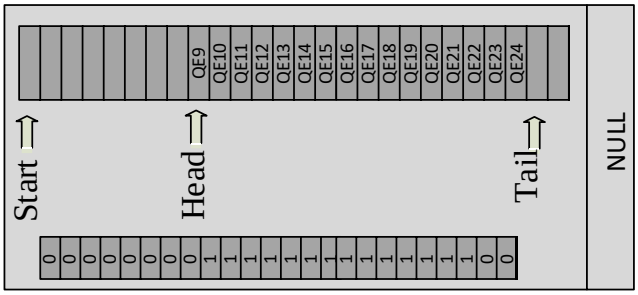
When using vector instructions, the search can start from any point in the array that is multiple of vec_size. That is because vector instructions read vec_size elements of the vector as a block, simultaneously. Figure 6.4 shows the way the holes and incoming messages are managed in linked list of vectors design when vec_size is 16. This design is very similar to linked list of arrays queue data structure. However, in this design a new pointer *start pointer* is defined to keep track of the starting points in searching the array. Figure 6.4(a) shows an empty queue. In Figure 6.4(b), 24 elements are added to the queue and the tail pointer is updated accordingly. In Figure 6.4(c), the first eight elements are removed from the queue and the head pointer is updated. However, the start pointer remains the same since there are still some elements in the first vector block. In Figure 6.4(d), QE9 to QE18 are removed from the queue which means that the first vector block is empty now and the



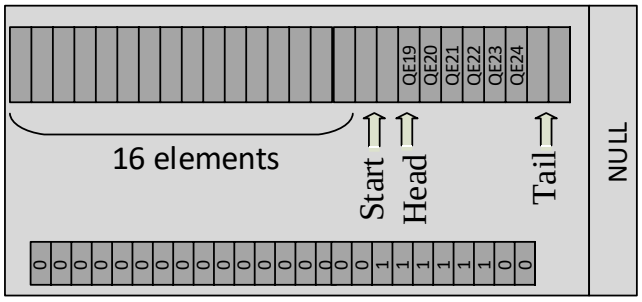
(a) The queue is empty



(b) 24 elements are added to the queue



(c) QE1 to QE8 are removed from the queue



(d) QE9 to QE18 are removed from the queue. start pointer is updated

Figure 6.4: Managing the holes and incoming messages in the linked list of vectors queue data structure with vec_size 16

start pointer can point to the start of second block which is element 16 of the vector.

It should be noted that the linked list of vectors design first searches for the tag in the first vector block. If any match occurs, it searches for the rank and then context_id in the same block. Otherwise, it searches the next vector block. This trend is continued until the whole vector is searched or the searching element is found.

There are a couple of caveats to this approach. First, similar to the linked list of arrays approach, it preforms best on a dense match-list with very few holes. Secondly, vector instructions are architecture-dependent so they should be tuned for different architectures.

6.3 Complexity Analysis

In this section, we present the runtime complexity, the number of memory references and memory complexity of the linked list of arrays/vectors data structures and compare it with the linked list and Open MPI data structures.

6.3.1 Runtime Complexity

Section 4.3.1 discussed the runtime complexity of the linked list and Open MPI data structures. In this section, we discuss the runtime complexity in the linked list of arrays/vectors data structure. For this, we use the same parameters defined in Table 4.1. In the linked list of arrays/vectors data structure, the insertion complexity is $\mathcal{O}(1)$ since the items are always added to the end of the arrays/vectors in the last linked list element where the tail points to. The deletion complexity is also $\mathcal{O}(1)$ as it always happens after the search operation at the position where the queue item is found.

As discussed earlier, in the linked list of arrays/vectors data structure there might be some holes in any of the arrays/vectors. The problem with holes is that they must be traversed as well even though they do not have any queue element. In the best case, there is no hole in the queue and the traversal complexity is $\mathcal{O}\left(\sum_{k=0}^{K-1}\sum_{r=0}^{r_k-1} q_{kr}\right)$. As shown

in Table 4.1, q_{kr} is the number of queue elements with context-id k and rank r. In the worst-case scenario where all the arrays/vectors in the data structure have only two queue elements at the head and tail of the array and $array\_size - 2$ holes, the search complexity becomes

$$\mathcal{O}\left(\sum_{k=0}^{K-1}\sum_{r=0}^{r_k-1} q_{kr} \times array\_size\right) \tag{6.1}$$

In this equation, *array_size* is the size of each array/vector in the linked list of arrays/vectors data structures. In the presence of MPI_ANY_SOURCE wildcard communication, we need to traverse MPI_ANY_SOURCE elements in the queues as well. Thus, the complexity becomes

$$\mathcal{O}\left(\sum_{k=0}^{K-1}\sum_{r=0}^{r_k-1} (q_{kr} + a_k) \times array\_size\right) \tag{6.2}$$

In this equation, a_k is the number of MPI_ANY_SOURCE queue items associated with the communicator k. By considering the array_size as a constant value, the traversal complexity becomes $\mathcal{O}\left(\sum_{k=0}^{K-1}\sum_{r=0}^{r_k-1} q_{kr}\right)$, which is equal to the complexity of the linked list data structure in Eq. 4.7.

We should note that despite the fact that search complexity in the linked list of arrays/vectors data structure is the same as linked list, its search operation is much faster. The reason for this is that this data structure provides the opportunity to utilize the cache more efficiently and reduce the number of memory references compared to the linked list data structure.

6.3.2 Memory Overhead Complexity

Section 4.3.2 discussed the memory overhead complexity of the linked list and Open MPI data structures. Here, we present the memory overhead complexity of the linked list of arrays/vectors data structure. In this data structure, the memory overhead is equal to the

number of holes *num_holes*. In the worse-case scenario, there are two queue elements in each array/vector and $array\_size - 2$ holes. Therefore, the memory overhead complexity will be

$$\mathcal{O}(array\_size) \tag{6.3}$$

By considering the *array_size* as a constant value, the memory overhead complexity in the linked list of arrays/vectors data structure becomes $O(1)$.

6.4 Performance Results and Analysis

In this section, we first describe the experimental platform. Then, selecting the size of array in each linked list element is discussed in Section 6.4.2. Finally, we provide microbenchmark and application results to show the efficiency of the linked list of arrays/vectors compared to MVAPICH queue data structure in Section 6.4.3 and Section 6.4.4, respectively.

6.4.1 Experimental Platform

We use two clusters for the experiments in this section. The first cluster has two nodes. Each node consists of a dual-socket 2.0 GHz 8-core Intel Xeon E5-2650 for a total of 16 cores, a 64 GB of memory, and running CentOS 7.1. Each node features an Intel Xeon Phi KNC coprocessor (5110P) with 60 cores clocked at 1.053 GHz, and running MPSS 3.4.5 and OFED-3.5-2-MIC. The nodes are interconnected through Mellanox ConnectX-3 FDR HCAs and switch. We refer to this cluster as Cluster C in the rest of this dissertation. The second cluster is Cluster B where its specifications have been discussed in Section 5.3.1. The MPI implementation is MVAPICH2-2.0. For fair comparative analysis, both linked list of arrays and linked list of vectors data structures are implemented in MVAPICH2.

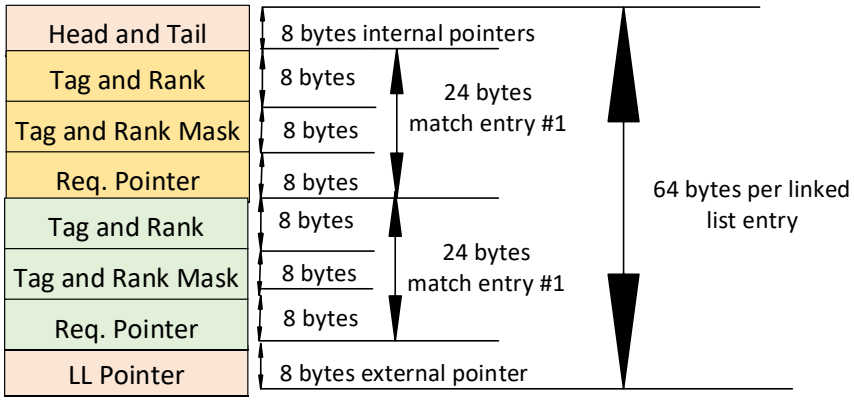
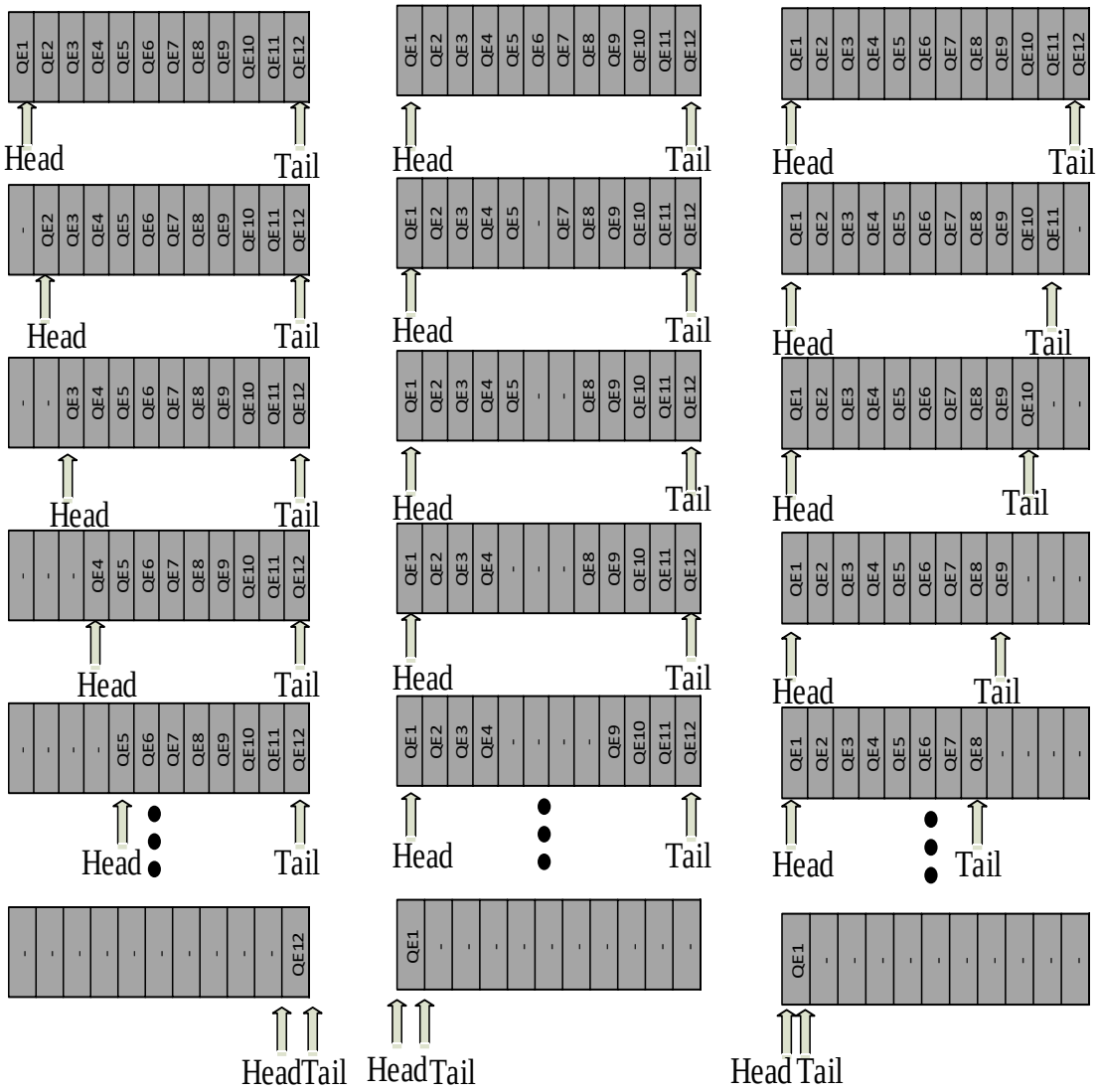


Figure 6.5: Packing data structures into 64 byte cache lines

6.4.2 The Size of Array in each Linked List Element

As discussed in Section 6.3, the size of array in each linked list element, *array_size*, can have considerable impact on runtime and memory overhead complexity. The advantage of increasing *array_size* is that it reduces the number of memory references and improves spatial locality. However, the disadvantage of increasing *array_size* is that it increases the number of traversals and memory overhead if there are significant number of holes in the array.

For the purposes of this research, the linked list of arrays technique is tuned to optimize the efficiency of each memory lookup. To accomplish this, the first logical spatial locality increase that we have explored aims to fill a cache-line of size 64 bytes (assuming x86 instruction set architecture) as shown in Figure 6.5. Each queue element for PRQ contains 24 bytes of information, 4 bytes for the tag, 2 bytes each for the rank and context id, 8 bytes of bit masks for matching, and an 8 byte pointer to the request. For readability, Figure 6.5 combines rank, tag and context id into a single 4 byte field. This results in each PRQ linked list element containing two entries ($array\_size = 2$). UMQ does not require masks, so it only requires 16 bytes per entry. Therefore, each UMQ linked list element can contain three entries ($array\_size = 3$) to fill in the cache-line [40]. Note that we use the *array_size* of 2 and 3 just for application results. For the microbenchmark results,



(a) Forward search (b) Middle search (c) Backward search

Figure 6.6: Different match order in microbenchmark tests

we use larger array sizes for each linked list element to better evaluate the impact of the linked list of array/vector technique on microbenchmark performance. The *array_size* of microbenchmark evaluations is discussed in Section 6.4.3.

6.4.3 Microbenchmark Results

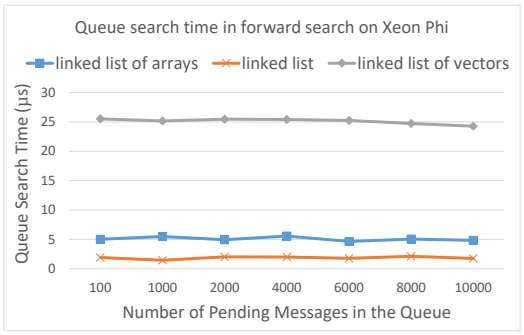
The microbenchmark program consists of two processes: a sender and a receiver. The goal is to build a certain length of the queue before the receiver starts searching through them. Therefore, we fill in the queue with a specific number of elements and see how the length of the queue affects the queue search time. We dequeue the elements once from the top of the queue (forward search), once from the middle of the queue (middle search) and also from the bottom of the queue (reverse search). Figure 6.6 shows how forward search, middle search and reverse search dequeue the elements from the array/vector in linked list of arrays/vectors data structure.

The Microbenchmark tests are conducted on Cluster C. We run the microbenchmark experiments twice. Once the experiments are executed in the native mode on the Intel Xeon Phi KNC coprocessors and once on Intel Xeon processors.

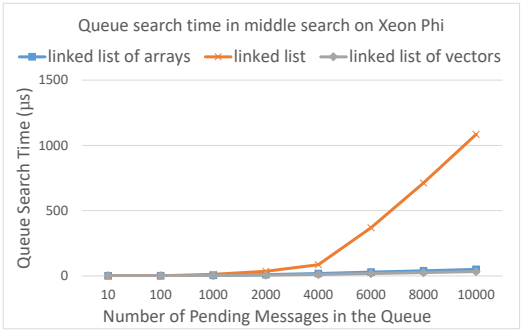
Microbenchmark Results with Intel Xeon Phi KNC Coprocessor

Figure 6.7 shows the average UMQ search time in linked list, linked list of arrays and linked list of vectors data structures on native mode on KNC. For the microbenchmark results on KNC coprocessors, the array length in each linked list element is equal to the size of the L1 cache size. This results in large array size in each linked list element which provides us with the opportunity to better evaluate the impact of the linked list of array technique on forward search, middle search and backward search microbenchmarks. The L1 cache size in KNC is 32 KB. Therefore, the size of the array for UMQ and PRQ is set to 2048 ($32KB/16$) and 1360 ($\lceil 32KB/24 \rceil$), respectively. In the absence of wildcard receive operations, the results for PRQ are the same as UMQ so they are not reported here.

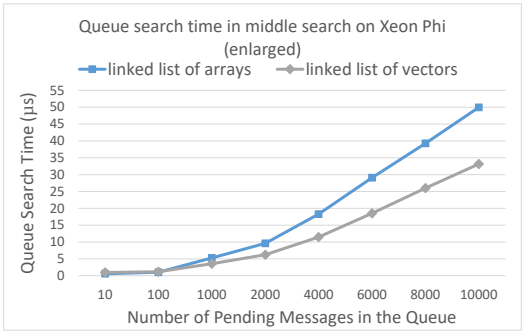
Figure 6.7(a) shows the queue search time in forward search. In forward search, the item of interest is always at the top of the queue in the linked list and linked list of arrays data structures so it can be found in one traversal. However, in linked list of vectors design



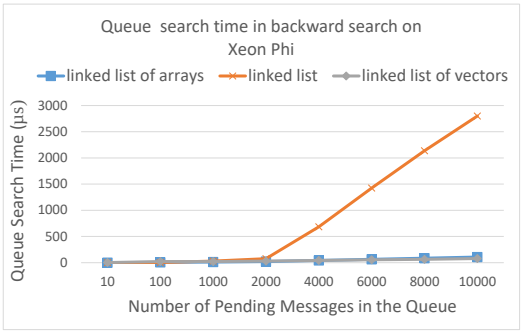
(a) Forward search



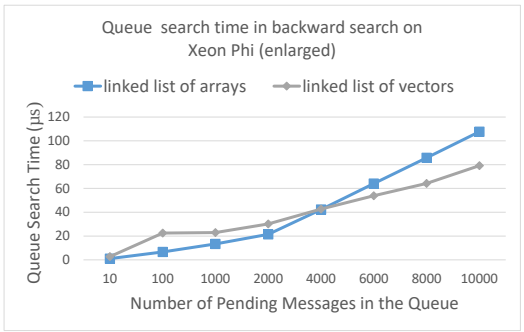
(b) Middle search



(c) Middle search (enlarged of (b))



(d) Backward search



(e) Backward search (enlarged of (d))

Figure 6.7: Queue Search Time in forward search, middle search and backward search with Xeon Phi KNC coprocessor on Cluster C

the search starts from the start pointer which only gets updated when the first vec_size elements are removed. This makes the linked list of vectors design to have more traversals. The linked list of vectors design has also some overhead for loading the vector elements to intrinsic variables. Due to these reasons, the linked list of vectors design incurs longer queue search time in forward search as can be seen in Figure 6.7(a). The queue search time in the other two approaches is less than $6\mu s$.

In regard to the middle search, the number of traversals in linked list of arrays/vectors is more than linked list data structure. This is because when an element is found at the middle of the linked list of arrays/vectors data structure, the head and tail pointers are not updated as can be seen in Figure 6.6(b). This creates holes in the array/vector. On the other hand, in the linked list data structure as soon as one element is found, it is removed from the queue. Despite the fact that linked list of arrays/vectors data structure has more traversals than the linked list data structure as can be seen in Figure 6.7(b), they perform better than linked list design. As discussed earlier, the reason for this is that accessing the queue elements in an array is much faster than linked list since it has less memory references. Moreover, when reading the first element from an array, the rest of the array elements are loaded into the cache (spatial locality). This decreases the queue search time, significantly. Figure 6.7(c) compares the queue search time in linked list of arrays with linked list of vectors data structure. As can be seen in this figure, using vector instructions can improve the queue search time specially for long list traversals.

Figure 6.7(d) shows the queue search time in reverse search in linked list and linked list of arrays/vectors. As can be seen in this figure, our approaches improve the queue search time significantly. Comparing linked list of arrays with linked list of vectors, we should note that linked list of vectors data structure compares every vec_size elements as a block, simultaneously. Therefore, even if there is only one element in a block, the whole block is searched. This makes the linked list of vectors design to incur more traversals.

Figure 6.7(e) compares the queue search time in linked list of arrays with linked list of

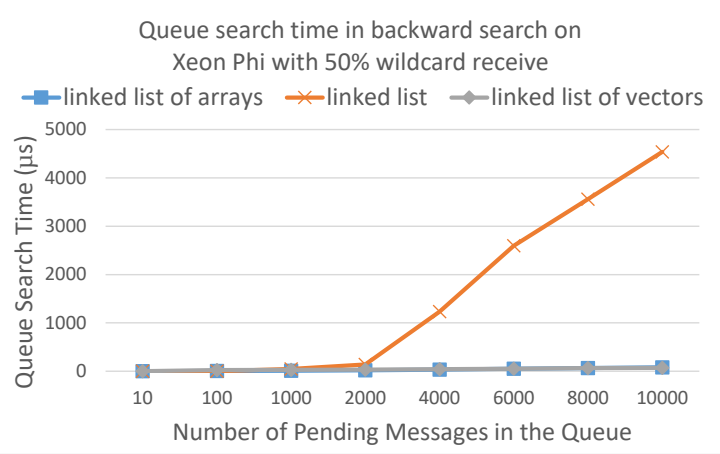


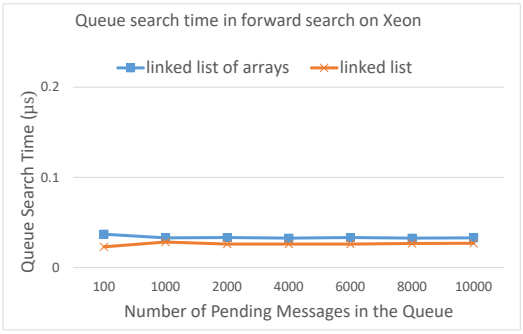
Figure 6.8: Queue search time with 50% wildcard receive in backward search with Xeon Phi KNC coprocessors on Cluster C

vectors. As can be seen in this figure, for short list traversals, linked list of arrays performs better than linked list of vectors due to having fewer number of traversals. However, for long list traversals, the use of vector instructions compensates the number of traversals and linked list of vectors performs better than linked list of arrays data structure.

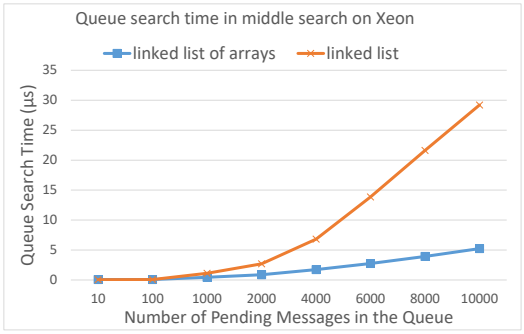
In order to show the applicability of the linked list of arrays/vectors in the presence of wildcard operation, we run the backward search microbenchmark with 50% wildcard receive operation (MPI_ANY_TAG). Figure 6.8 shows the results on Xeon Phi KNC. It is obvious from the figure that linked list of arrays/vectors perform much better than the linked list data structure even in the presence of wildcard communication.

Microbenchmark Results with Intel Xeon Processor

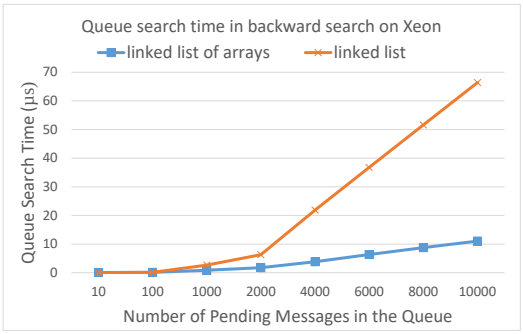
Figure 6.9 shows the queue search time in linked list and linked list of arrays data structures on Intel Xeon processor. We use Cluster B for the experiments in this figure. A detailed description of this cluster is provided in Section 5.3.1. Note that the Xeon processor used in this study does not support vector instructions. Therefore, the linked list of vector results are not reported on this processor. Similar to previous section, the results are shown for



(a) Forward search



(b) Middle search



(c) Backward search

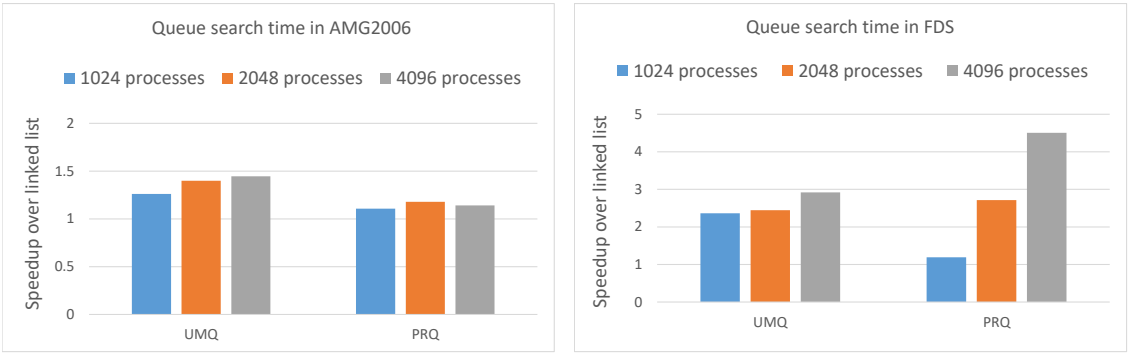
Figure 6.9: Queue Search Time in forward search, middle search and backward search with Xeon processor on Cluster B

large array size on UMQ and for different searching orders depicted in Figure 6.6 (forward search, middle search, and reverse search, respectively). Figure 6.9 shows that the linked list of arrays data structure can provide significant performance improvement for long list traversals on Xeon processor while preserving the performance for short lists.

6.4.4 Application Results

This section presents AMG2006 and FDS application results to show the efficiency of the linked list of arrays data structure compared to linked list. For the experiments, we use Cluster B. Note that we use array_size discussed in Section 6.4.2 for the experiments in this section.

Figure 6.10 shows the UMQ and PRQ search time speedup over linked list in AMG2006



(a) Queue search time speedup in AMG2006 (b) Queue search time speedup in FDS

Figure 6.10: PRQ and UMQ search time speedup of linked list of arrays approach over linked list in AMG2006 and FDS on Cluster B

and FDS applications, respectively. For AMG2006 , we show the average queue search time speedup over all processes. However, for FDS, average queue search time for process 0 is shown. The reason for this (as discussed in Section 3.4.3) is that in this application the majority of the communications is done with process 0 and other processes do not generate long message queues.

Figure 6.10(a) shows the average queue search time in AMG2006 application when the number of processes increases from 1024 to 4096. As can be seen from the figure, we can gain up to 1.44x and 1.17x speedup over linked list in UMQ and PRQ search time, respectively. Figure 6.10(b) shows the PRQ and UMQ search time speedup in the FDS application. In this application, the queue length increases with increasing number of processes. Therefore, higher improvement is gained as we increase the number of processes. With 4096 processes, it achieves up to 2.9x and 4.5x for UMQ and PRQ, respectively.

Figure 6.11 shows the FDS application runtime speedup. The results in this figure is in concert with the results shown in Figure 6.10(b). We can observe that the improvements in queue search time directly translates to application performance. With 4096 processes, we can gain around 2.92x performance improvement. We do not present the runtime results for AMG2006 since its queue search time improvements do not translate to a significant

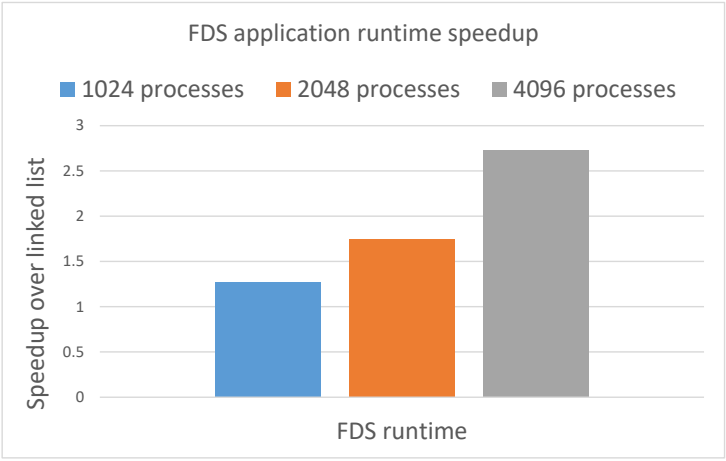


Figure 6.11: FDS application runtime on Cluster B

improvement in the application runtime. As discussed in Section 3.4.4, there is a sharp contrast between the number of queue searches for AMG2006 and that of FDS, which could translate to application performance only for FDS.

6.5 Summary

MPI message matching is designed in the first place for traditional systems with heavy-weight cores so it does not perform well on many-core systems with light-weight cores. In this chapter, we take advantage of two techniques that leverage the hardware features available on the new parallel computing systems to improve the matching performance on many-core systems while maintaining or improving the performance on traditional Xeon processors as well. In the first technique, the message queue design is restructured to linked list of arrays. By using an array for representing the queue, the time for accessing each element in the queue is reduced. We further improve the performance by taking advantage of vector operations such as Intel's AVX intrinsics on KNC for searching the queue.

The evaluation results on three different microbenchmarks show that the proposed approaches can improve the queue search time on Xeon Phi many-core coprocessor by up to 35.3x. We also show the queue search time and application runtime improvement by

up to 4.5x and 2.92x on Xeon processors for applications with extreme message matching requirements, respectively.

So far, we improve the performance of MPI communication by considering message queue operations. In Chapter 7, we will consider neighborhood collective operations to enhance MPI communication performance.

Chapter 7

MPI Neighborhood Collective Optimization

In previous chapters, we have proposed mechanisms to improve MPI communication through efficient message matching operations. In this chapter, we consider this issue from a different perspective and propose an efficient collaborative communication mechanism to improve the performance of neighborhood collective operations.

As discussed in Chapter 2, neighborhood collectives are added to the MPI standard to address the issues associated with conventional collective communications. In neighborhood collectives, each process only communicates with certain processes considered its neighbors. The neighbors are specified using the communication pattern derived from the topology graph of the processes. MPI libraries can leverage the topology information associated with neighborhood collectives to derive an optimized communication pattern. However, the well-known MPI libraries such as MPICH, MVAPICH, and Open MPI perform neighborhood collectives using a naïve approach, where every process uses nonblocking send/receive operations to send/receive data to/from each of its incoming/outgoing neighbors. In other words, no specific pattern is used to govern the communications in order to gain better performance.

In this chapter, we propose an optimized collaborative communication pattern and schedule to improve the performance of neighborhood collectives [51]. With respect to

the process topology, we target the distributed graph topology which provides the highest flexibility for describing various topologies. The distributed graph topology is used to extract useful information for communication optimization. More specifically, this information is leveraged to discover and exploit *common neighborhoods* in the topology with the aim of optimizing the performance of neighborhood collectives through *message combining*. Message combining provides the opportunity to reduce the number of communications which is especially beneficial for small message sizes, which is our main objective in this chapter.

The communication schedule provides a detailed description and ordering of the send, receive, and memory copying operations to implement a given neighborhood collective operation. The experimental results show that the proposed algorithm improves the performance of nonblocking neighborhood allgather operation up to 8x times.

The contributions of this chapter are as follows:

- We design a collaborative communication pattern for neighborhood collectives. In this communication pattern, groups of k processes collaborate with each other to perform message combining and reduce the number of message transfers. For this, we propose a *distributed maximum matching* algorithm for weighted k-uniform hypergraphs.

- We propose a topology-agnostic communication pattern as well as an enhanced topology-aware communication pattern. In the topology-aware design, the physical topology of the system is considered in deriving the communication pattern.

The rest of the chapter is organized as follows. Section 7.1 discusses the related work. Section 7.2 explains the preliminary concepts related to our design. Section 7.3 and 7.4 discuss the proposed communication pattern and communication schedule designs, respectively. Selecting the parameters of the design is discussed in Section 7.5. Section 7.6 presents the experimental results. Finally, Section 7.7 concludes the chapter.

7.1 Related Work

Challenges in designing sparse collective operations in MPI is discussed by Hoefler and Traff [61]. They propose three operations that formed the basis for neighborhood collectives in MPI. Ovcharenko et al. [93] introduce a general-purpose package on top of MPI to improve the performance of sparse communications in some applications. Hoefler et al. [59] evaluate the applications that can benefit from distributed topology interfaces. They also discuss the optimization challenges that must be addressed by MPI implementers. Kandalla et al. [68] design non-blocking neighborhood collective operations by using a network-offload-based approach, and redesign the BFS algorithm. Kumar et al. [75] optimize applications with neighborhood collective operations by leveraging the multisend interface of the IBM Deep Computing Framework (DCMF) [74]. Unlike our design, this approach does not attempt to derive an optimized communication pattern. Each process still sends a message to each of its outgoing neighbors directly. Our work is orthogonal to this approach.

A number of optimization principles for neighborhood collectives are discussed by Hoefler and Schneider [60]. They leverage graph coloring to design a communication schedule that avoids creation of hotspots at the end nodes. Moreover, they provide an algorithm that balances communications by offloading them from high-outdegree processes to those having lower outdegrees. As compared to our work that can be applied to both balanced and unbalanced topologies, balancing the communication tree is only beneficial for unbalanced topologies. This is important because many real applications have balanced neighborhood topologies.

Traff, et al. propose a specification [109] and message combining algorithms [108] for *isomorphic, sparse collectives* in which all processes have the same neighborhood structure. Their work is limited to isomorphic neighborhoods, which are not yet available in MPI. In contrast, our approach is designed over distributed graph topology interface that defines neighborhood topologies in the most generic and flexible way.

Lübbe [86] presents a microbenchmark and an experimental methodology to assess the performance of neighborhood collectives. This work formulates the performance expectations of neighborhood collectives as *self-consistent performance guidelines* and show that MPI libraries are sensitive to the specification of topological neighbors.

Mirsadeghi et al. [91] propose a message-combining mechanism for distributed graph topologies. Although this work achieves good performance improvement over the default approach, it has some limitations. The first limitation is that it uses a basic message combining strategy that considers only pairs of processes at each round. Moreover, it does not consider the physical topology and/or mapping of processes. In this dissertation, we address these limitations by designing a communication pattern in which groups of k processes collaborate with each other to perform message combining. For this, we propose a distributed maximum matching algorithm for weighted k-uniform hypergraphs. Moreover, we further optimize the communication pattern by considering the physical topology of the system in our design.

7.2 Preliminaries

The core idea of our proposed design is that groups of k processes collaborate with each other to reduce the number of message transfers among the processes in neighborhood collectives. This can particularly improve performance for small messages, which is our main objective in this chapter.

7.2.1 Common Neighbors and Friend Groups

In order to reduce the number of communications in neighborhood collectives, we classify the processes into groups of size k. The processes in each group collaborate with each other to perform message combining. The *common neighbors* of the processes p_1, p_2, ..., p_k are the set of processes that are an outgoing neighbor of all the processes p_1, p_2, ..., p_k. The

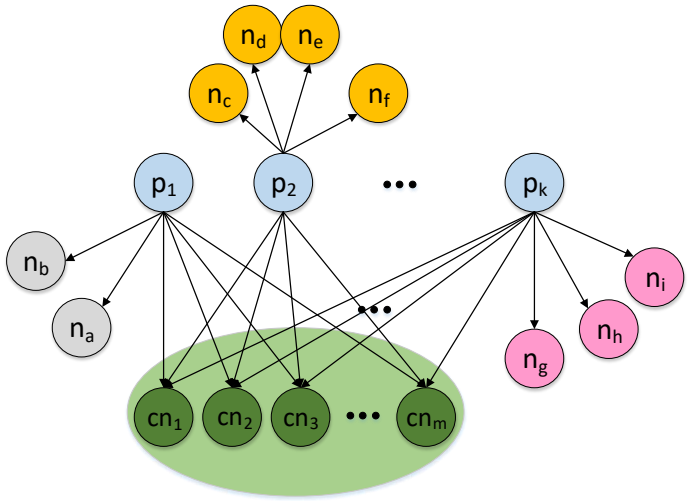


Figure 7.1: An example of process topology graph: processes cn_1, cn_2, ..., cn_m are common neighbors of processes P_1, P_2, ..., P_k

processes p_1, p_2, ..., p_k are said to be a *friend group* if they have a certain number of common neighbors. The minimum number of common neighbors that a group of k processes should have to be considered as a friend group is defined as θ. Figure 7.1 shows an example of a process topology graph. In this figure, processes cn_1, cn_2, ..., cn_m are common neighbors of processes p_1, p_2, ..., p_k. Assuming $m > \theta$, then processes p_1, p_2, ..., p_k will be a friend group. We should note that each friend process might also have other outgoing neighbors that are not common with other members of the group. For example, such outgoing neighbors of the processes p_1, p_2 and p_k are shown in gray, orange and pink, respectively. The common neighbors of processes p_1, p_2, ..., p_k are shown in green.

7.2.2 Number of communications

In the collaborative mechanism, each friend process p_i is responsible for transferring a message to m/k of the common neighbors. To this end, the processes p_1, p_2, ..., p_k first have to communicate with each other to exchange their messages. Then, each process p_i makes a combined message for the subset of common outgoing neighbors that it is responsible

for. This way, each process p_i has a maximum of m/k communications with the common neighbors plus $k-1$ communications for exchanging messages with its friends. Therefore, the total number of communications for each process p_i and a specific k can be given by Eq. 7.1.

$$N = (m/k) + k - 1 \tag{7.1}$$

If m is not divisible by k, one more common neighbor is assigned to $(m \mod k)$ number of friends, as in Eq. 7.2.

$$N_k = \begin{cases} \lceil m/k \rceil + k - 1, & \text{if } i \leq (m \mod k) \\ \lfloor m/k \rfloor + k - 1, & \text{if } i > (m \mod k) \end{cases} \tag{7.2}$$

In order to get any benefit from message combining, the parameters k and θ should be selected appropriately. We discuss selecting the parameters in detail in Section 7.5.

7.2.3 Main Steps of the Design

Our proposed design consists of two main steps. In the first step, the communication pattern is built based on the given topology graph. For this, we find the common neighbors between a group of k processes and use a collaborative mechanism between these processes to build the communication pattern. In the second step, we use the derived communication pattern to build an efficient communication schedule for neighborhood collectives. The first step of the design depends only on the topology graph. Therefore, it is built only once for each given process topology graph and can be used multiple times for neighborhood collective calls. Moreover, the results from the first step can be used for different types of neighborhood collectives. For example, it can be used to make a communication schedule for both neighborhood allgather and neighborhood alltoall collectives.

In the rest of this chapter, first we discuss building the communication pattern using

the given topology graph in Section 7.3 (Step 1). Then, in Section 7.4 we provide an algorithm to make the communication schedule based on the derived pattern and the desired neighborhood collective operation (Step 2).

7.3 Communication Pattern Design

In this step, we classify the processes into groups of k friends. Each group performs a collaborative mechanism to build the communication pattern. For this, we mutually group the processes based on their common neighbors.

7.3.1 The Collaborative Mechanism

Algorithm 7.1 shows a distributed collaborative mechanism between a group of k friends. Each process in the topology graph runs this algorithm to make its own part of communication pattern. For each process such as p, the pattern designates the following: (1) the set of other processes with which p should build a friend group at each communication stage, (2) the set of outgoing neighbors to which p should send a message at each stage, and whether the message will be a combined message or not, and (3) the set of incoming neighbors from which p expects to receive a message at each stage, and again whether that message will be a combined message or not. Algorithm 7.1 shows how we build such a pattern. Table 7.1 shows the parameters that are used in Algorithm 7.1 and provides their definition. The inputs of this algorithm are the set of outgoing and incoming neighbors specified by O and I, respectively, the friendship threshold θ and the parameter k. The output of the algorithm is the communication pattern τ.

Line 1 saves process rank in p. Then, we generate a friendship matrix T in Line 2. Matrix T provides the following information: (1) groups of $k-1$ processes that have an exact same set of outgoing neighbors such as X in common with p, as well as with each other, and (2) the set of common outgoing neighbors X for each group of $k-1$ processes.

Table 7.1: The list of parameters

p	The process rank
O	The set of outgoing neighbors of p
I	The set of incoming neighbors of p
O_a	The set of active outgoing neighbors of p
I_a	The set of active incoming neighbors of p
θ	The friendship threshold
k	The friendship group size
τ	The output communication pattern
T	The friendship matrix
A	The auxiliary array to make matrix T
d	The number of rows in matrix A
s	The number of message combining steps
C	The neighbor information
F	A vector of selected friends at each iteration
CN	A vector of common neighbors at each iteration
CN_{off}	The common neighbors that are assigned to a friend of p
CN_{on}	The common neighbors that are assigned to p

Note that the size of X is greater that θ ($|X| > \theta$). Figure 7.2(b) shows an example of matrix T. Each column of this matrix corresponds to an outgoing neighbor of p. Each row of the matrix corresponds to a group of k friends. The (i, j)th element in the matrix is 1 if the friend group corresponding to row i has an outgoing neighbor corresponding to column j in common. Otherwise, this element is 0. Note that we have at least θ common neighbors at each row. The algorithm to generate matrix T is discussed in detail in Section 7.3.2. Line 3 initializes O_a and I_a, which denote the list of active outgoing and incoming neighbors of p, respectively. Active neighbors correspond to the neighbors that have not been dealt with yet. We also initialize s to 0; s corresponds to the number of steps in which message combining is performed in the design. We also initialize the *neighbor information* C and common neighbors CN to null in Line 4. The neighbor information has the information required for each outgoing neighbor and is a part of its communication pattern. For example, it shows if the message is sent directly to an outgoing neighbor or it is assigned to one of the friend processes. It also shows the step in which message transfer to each outgoing neighbor is handled. The neighbor information is sent to each outgoing neighbor of p to let them

know about the messages they should expect to receive from p in the resulting pattern.

Lines 5 to 26 present the main loop of the algorithm, where three main tasks are preformed: (1) the processes are classified into groups of k friend processes, (2) the common neighbors are divided between the friend processes, and (3) the topology information and the communication pattern is updated. In Line 6, p finds a group of friend processes with which it has the most number of common neighbors. The number of friend processes in each group is k and they are specified by $F = (f_1, f_2, ..., f_k)$. This line is explained in more detail in Section 7.3.3. The friends in vector F are sorted based on their process rank. If F is found, we use T to extract the set of common neighbors between the friend processes in F and save it in CN (Line 8). In Line 9, we find the index of p in vector F and save it as m. In Lines 10 to 17, the common neighbors are distributed among the friend processes in F. This distribution is done based on process ranks. For this, we divide the common neighbors into k parts (Line 10). Each part is assigned to one of the processes in F (Lines 11 to 17). For example, the first part is assigned to the process with smallest rank, the second part is assigned to the process with the second smallest rank and so on. In Line 18, we update the communication pattern τ using the vector F at step s. In Line 19, we use τ to extract the neighbor information C for each outgoing neighbor of p.

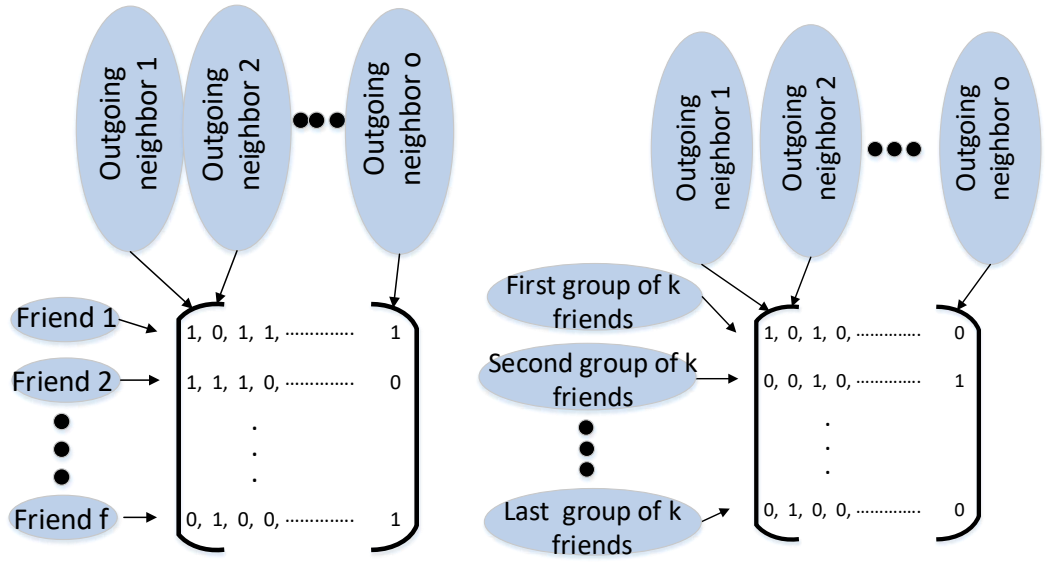
After distributing the common neighbors among the friend processes, we notify the active outgoing neighbors as to whether they are assigned to p or not (Line 21). Active outgoing neighbors receive this information and update their active incoming neighbor matrix I_a (Line 22). In Line 23, we remove CN from the set of active outgoing neighbors O_a. In Line 24, the friendship matrix T is updated based on the active outgoing and incoming neighbors (O_a and I_a). If the number of active outgoing neighbors in a row becomes less than θ, the corresponding row is removed from T. At the end of the loop (Line 25), the step number is incremented.

Once a process gets out of the loop in Lines 5 to 26 due to $T = \emptyset$, it should still issue receive operations corresponding to its active incoming neighbors. This is needed because

Algorithm 7.1: Distributed message combining mechanism for a group of k processes

Input : Set of outgoing neighbors O, set of incoming neighbors I, friendship threshold θ, the size of each group k
Output: The communication pattern τ

```
1  p = Self-rank;
2  T = Build-friendship-matrix(O,I,θ, k);
3  O_a = O, I_a = I, s = 0;
4  C= {∅}; CN= {∅};
5  while T ≠ ∅ do
6      F= Choose-friendgroup-for-messagecombining(T, k);
7      if F found then
8          CN=Find-common-neighbors(T, F);
9          m=The index of p in the sorted vector F;
10         Divide elements in CN into k subsets of equal sizes;
11         for i = 1 to k do
12             if i == m then
13                 Keep the ith part of CN;
14             else
15                 Assign the ith part of CN to F[i];
16             end
17         end
18         Update-communication-pattern(τ, F, CN_on, CN_off, s);
19         Extract-neighbor-info(C, τ);
20     end
21     Notify each neighbor in O_a if it is assigned to p or not;
22     Receive notifications from neighbors in I_a and update it;
23     O_a = O_a − CN;
24     Update-friendship-matrix(T, O_a, I_a);
25     s = s + 1;
26 end
27 while |I_a| > 0 do
28     Receive notifications from neighbors in I_a and update it;
29 end
30 Send neighbor information C to outgoing neighbors O;
31 τ=Update communication pattern based on neighbor information C received from
     Incoming neighbors I;
```



(a) Matrix A (b) Matrix T

Figure 7.2: Matrices used in the distributed message combining mechanism

those neighbors will be sending notifications to this process until they get out of the loop too. This is done in Lines 27 to 29. In Line 30, each process sends the neighbor information to its outgoing neighbors. The outgoing neighbors add this information to the second part of their communication pattern (Line 31). The communication pattern will be used as the input of the scheduling algorithm which will be discussed in Section 7.4.

7.3.2 Building the Friendship Matrix

Algorithm 7.2 shows how the friendship matrix, T, is created. As discussed in Section 7.3.1, this matrix has the information about all friend groups of p and their common neighbors, and it is used to make the communication pattern. The inputs of the algorithm are the outgoing and incoming neighbors, O and I, the threshold θ and the number of friends in each group k. The output of the algorithm is the friendship matrix T. In this algorithm, the Auxiliary matrix A is used for making matrix T. Matrix A has the list of friend processes of p and the set of common neighbors between process p and each one of its friends. Figure

7.2(a) shows an example of matrix A. In matrix A, each row l corresponds to one friend process of p and each column j corresponds to one outgoing neighbor of p. If p and the friend process corresponding to row l both have the outgoing neighbor corresponding to column j in common, then $A[l][j]$ is set to 1. Otherwise, it is set to 0.

First, we save process rank in p in Line 1. In Line 2, we initialize matrix A to null. We also initialize the number of rows, d, of matrix A to zero. In Line 3, we use the procedure explained in [91] to extract the common neighborhood matrix M. Each row of M is associated with one of the outgoing neighbors of p, and it lists all incoming neighbors of the corresponding outgoing neighbor. We use the term f for each incoming neighbor of outgoing neighbors.

In Lines 4 to 13, we build matrix A by processing the elements in M. Lines 5 to 7 correspond to topology-agnostic approach, whereas Lines 8 to 12 are associated with topology-aware approach. In the topology-agnostic approach, for all the rows and columns in M (Line 4), we call the function UpdateA (Line 6). This function (Line 20) finds the row corresponding to f in A (Line 26) and set the element associated with this row, and the column j to 1 (Line 27). Note that if there is not such a row, it adds a new row to A (Line 22) associated with f and initialize the row to zero (Line 23). We use the same procedure for topology-aware approach in Lines 8 to 12. The only difference is that we add a row corresponding to process f only when f is on the same node (Line 9). Otherwise, we do not consider it as a friend.

After matrix A is created, we use it to make matrix T. Figure 7.3 shows an example of generating a row of matrix T from matrix A. In this example, we assume $k = 4$ and the processes f_1, f_2 and f_4 make a group with process p. For generating this row, we select the rows associated with the friends f_1, f_2 and f_4 in matrix A. We call the corresponding vectors v_1, v_2 and v_3, respectively. Then we perform a bitwise AND (&) operation among these vectors to get a resulting vector r, such that $r[i] = j_1[i]\&j_2[i]\&j_3[i]$. If $r[i] = 1$, then it means that f_1, f_2, f_4 and p all have the ith outgoing neighbor of p in common. The

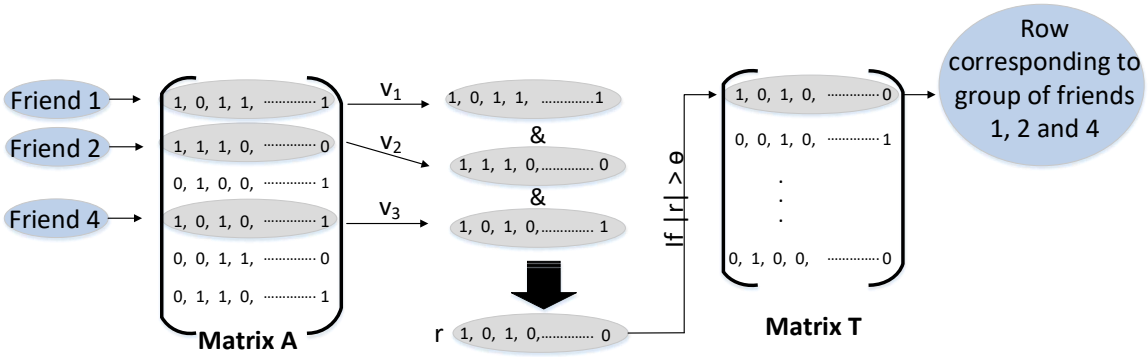


Figure 7.3: An example of generating a row of matrix T from matrix A

number of elements equal to 1 in vector r shows the number of common neighbors between the processes f_1, f_2, f_4 and p. If this number is greater than θ, we add row r to matrix T as can be seen in Figure 7.3. We perform this procedure for all possible combinations of choosing 3 rows in vector A. Lines 14 to 19 of Algorithm 7.2 shows this procedure for a group of k processes.

7.3.3 Mutual Grouping of the Processes

In Line 6 of Algorithm 7.1, a group of k friends is selected for message combining. Among all groups of k processes in matrix T, p should select one group with which it has the highest number of common neighbors. The advantage of selecting a group with higher number of common neighbors is that it provides more reduction in the number of communications.

In selecting a group of friends, it is important to ensure that the selection is consistent among all the members of the group. For instance, with $k = 3$, if p chooses (f_1, f_2) as its desired group, then we must make sure that f_1 and f_2 will also choose (p, f_2) and (p, f_1) as their friend groups, respectively. We model this problem as a distributed maximum weighted matching problem in hypergraphs. The corresponding hypergraph is the friendship hypergraph $H(P, E)$, where P is the set of all processes and $\{p_1, p_2, ..., p_k\} \in E$ if and only if $p_1, p_2, ..., p_k$ have at least θ neighbors in common. H is k-uniform because all friend groups

Algorithm 7.2: Building friendship matrix

Input : Set of outgoing neighbors O, set of incoming neighbors I, friendship threshold θ, the size of each group k
Output: The friendship matrix T

```
1  p= Self-rank;
2  A = null, d=0;;
3  M= Generate-common-neighborhood-matrix(O, I);
4  for all f = M[i][j] do
5      if topology-agnostic then
6          Update-A(A, f, j);
7      end
8      if topology-aware then
9          if p and f are on the same node then
10             Update-A(A, f, j);
11         end
12     end
13 end
14 for all combinations of k − 1 rows such as v_1, v_2, ..., v_{k−1} in A do
15     r= v_1 & v_2 &...& v_{k−1};
16     if |r| > θ then
17         Add row r to matrix T;
18     end
19 end
20 function Update-A(A, f, j)
21 if no row in A corresponding to f then
22     Add a new row to A associated with f with index d;
23     A[d] = 0;
24     d++;
25 end
26 l= the row corresponding to f in A;
27 A[l][j] = 1;
28 end function
```

are of the same size k. The weight of each hyperedge denotes the number of common neighbors of the group of friend processes that are represented by that hyperedge. Note that the friendship hypergraph H is a distributed graph that we extract from the process topology graph. The friendship matrix T discussed in Section 7.3.1 and 7.3.2 represents the part of H that is local to each process such as p. In other words, it has information about all the hyperedges of H that p is a member of.

Many papers consider maximum matching problem in hypergraphs [67, 36, 81, 94, 115, 72]. However, they either do not provide a distributed algorithm [67, 36, 81, 94, 115] or they use assumptions [72] that make their solution not applicable to our case in this chapter. For example, in [72], the authors assume that the distributed system has a process for each hyperedge. In mutual grouping problem, this means that each process should be responsible for one hyperedge of friendship hypergraph. This is problematic since the number of hyperedges of the friendship hypergraph can be more than the number of processes. Therefore, these approaches are not applicable to our work.

The research in [112, 62, 84, 85] propose algorithms for the distributed weighted matching in ordinary graphs. Based on an evaluation study in [63], Hoepman's algorithm [62] performs better than others. In this section, we propose an algorithm that extends Hoepman's algorithm to support hypergraphs. In other words, we propose a distributed maximum weighted matching algorithm for k-uniform hypergraphs.

Algorithm 7.3 provides a detailed description of the proposed algorithm. The input of this algorithm is the friendship matrix T and the parameter k. The output of the algorithm is a group of friends for message combining. This algorithm works based on the information in two main sets: B and R. These sets are updated in each iteration of the algorithm according to the messages received from other processes. These messages can be either a request or drop message. A request from process x to process y means that process x wants to make a friend group with process y. A drop message means that process x will not

make a friend group with process y. The set B maintains the hyperedges or friend groups[1] that can potentially be the chosen hyperedges in the maximum weighted matching. It also contains the weight of each hyperedge which is actually the number of common neighbors for each friend group. At initialization, this set contains all possible hyperedges in matrix T and their associated weights in Line 2. The set R contains all processes from which a request has been received, and their selected hyperedges. For example, the element $\{u, V\}$ in set R of process p shows that process u requests process p to make a friend group or hyperedge with the processes listed in vector V. This set is initialized to null in Line 3 of the algorithm. Among all hyperedges in vector B, we choose a hyperedge with maximum weight and save the processes of this hyperedge in vector G (Line 4). Then, in Lines 5 to 7, we send a request including vector G to all processes in G. This way, each process informs the processes with whom it wants to be friend and lets them know about its selected group of friends.

Lines 8 to 41 is the main loop of the algorithm. In each iteration of this loop, two main tasks are performed iteratively: (1) Process p receives a drop or request message from a process and updates R and B accordingly, and (2) Based on the received request and the information in R and B, process p sends a drop or request message to other processes. This trend continues until a group of friend processes is mutually selected by all processes in a group or until there is no friend group left in the set B. In Line 9, process p receives message m from process u. If m is a request message, then we add process u and its selected group of friends V to set R (Lines 10 to 12). Otherwise, if it is a drop message, we remove from B all groups of friends who have process u as one of their members (Lines 14 to 18). If p chooses u as one of its friends (Line 19) but it receives a drop message from u (Line 13), it has to choose a new group of friends from B. In Lines 20 to 21, we check if any friend group is left in B. If no friend group is left, we return null and exit the loop (Line 21). Otherwise, we choose a new group of friends (Line 23) and send them a request message

[1] In the rest of the chapter, we use the terms hyperedge and friend group interchangeably.

Algorithm 7.3: Distributed and mutual grouping of processes

Input : Neighborhood matrix T, the number of friend processes in each group k
Output: A group of friends for message combining

```
p= Self-rank;
B= extract-hyperedges-and their-weight(T);
R= ∅;
G = find-hyperedge-with-maximum-weight(B);
for i = 0 to k do
    send a request including G to process G[i];
end
while TRUE do
    Receive m from u;
    if m=request that contains friend group V then
        R = R ∪ {u, V};
    end
    if m=drop message then
        for all Z ∈ B do
            if u ∈ Z then
                B = B − {Z};
            end
        end
        if {u ∈ G} then
            if B = NULL then
                Return null;
            end
            G = find-the-hyperedge-with-maximum- weight(B);
            for i = 0 to k do
                send a request including G to process G[i];
            end
        end
    end
    c = 0;
    for i = 0 to k do
        if {{G[i], G} ∈ R} then
            c + +;
        end
    end
    if c == k − 1 then
        for all w ∈ B − {G} do
            Send drop message to w;
        end
        Return G;
    end
end
```

(Lines 24 to 27). In Lines 30 to 34, we check the set R to see if the processes that are chosen by process p have also selected p with the same group of friends G. In other words, we count the number of processes in G who meet this condition. If the number of such processes was $k-1$ (Line 35), it means that the friend group selection is mutual between all the processes in G. Therefore, we send a drop message to all the remaining processes in B (Lines 36 to 38) and return G as our chosen friend group for message combining (Line 39).

7.3.4 Complexities

The complexity of Algorithm 7.2 can be given by $O(\Delta^2 + F^k)$, where Δ denotes the maximum number of neighbors per process (degree of the process topology graph) and F is the maximum number of friends per process. In topology-aware approach, the worst-case complexity is $O(\rho^k)$, where ρ is the number of processes per node. On the other hand, the worst-case complexity of topology-agnostic approach is $O(n^k)$, where n is the number of processes.

The complexity of Algorithm 7.3 can be given by $O(r\alpha)$, where r is the number of iterations of the algorithm and α is the number of friend groups. In worst case, the number of iterations, r, is equal to $k\alpha$. For a complete friendship graph, we have $\alpha = \binom{\rho}{k}$ and $\alpha = \binom{n}{k}$ for topology-aware and topology-agnostic approaches, respectively. So, the worst-case complexity of Algorithm 7.3 is given by $O(\rho^{2k})$ and $O(n^{2k})$ for topology-aware and topology-agnostic approaches, respectively.

The complexity of Algorithm 7.1 is $O(F^k + tr\alpha)$, where t denotes the number of iterations of the algorithm. In worst case, the number of iterations, t, is equal to the number of friend groups α. Assuming a fully connected friendship graph, the worst-case complexity of Algorithm 7.1 is given by $O(\rho^{3k})$ and $O(n^{3k})$ for topology-aware and topology-agnostic approaches, respectively.

Note that these are worst-case complexities which assume complete process topology

and friendship graphs. In practical use cases of neighborhood collectives, these graphs are expected to have a lower degree than that of a complete graph.

7.4 The Proposed Communication Schedule Design

In Section 7.3, we discussed how to use the underlying process topology graph to design an optimized communication pattern. This section uses the derived communication pattern to build a communication schedule for the neighborhood allgather operation. In contrast to communication pattern that is built only once for each topology graph, communication schedule should be built each time a neighborhood collective is called. Algorithm 7.4 provides a detailed description of the proposed communication schedule design. The inputs of this algorithm are the communication pattern τ which is derived from Algorithm 7.1 and all the inputs required for a neighborhood collective operation such as send/receive buffers, send/receive message sizes, send/receive data types and MPI communicator. The output of the algorithm is the communication schedule ω for neighborhood allgather.

The communication pattern τ consists of multiple steps. Each step corresponds to one iteration in the main loop of Algorithm 7.1 and has the following information: (1) The group of friend processes for process p at step s, if any, and (2) the common neighbors associated with the friend processes in the group. This information is used in the main loop of Algorithm 7.4 (Lines 2 to 24) to make the communication schedule.

This loop performs two tasks iteratively: (1) The friend processes communicate with each other to exchange their messages (Lines 3 to 7), and (2) each friend process generates the combined message and sends it to the common neighbors associated with it (Lines 8 to 19). After the main loop is done, there might be some ongoing neighbors that do not belong to any group. So, we need to communicate with them directly in a naive way (Lines 25 to 34). More specifically, at each step s of the communication pattern τ, process p checks if it has a friend group for message combining (Line 3). If yes, it communicates with them

Algorithm 7.4: Communication Schedule Design

Input : Communication pattern τ, send/recv buffers, send/recv sizes, send/recv data types, MPI communicator
Output: Communication schedule ω

```
1  p= Self-rank;
2  for each step s in τ do
3      if have a goup friend at step s for message combining then
4          for each friend f in G associated with step s do
5              ω ← send its own message to f;
6              ω ← receive process f's message from him;
7          end
8          ω ← call MPI wait;
9          ω ← generate the combined message;
10         for each outgoing neighbor n_o do
11             if n_o is assigned to p at step s then
12                 ω ← send the combined message to n_o;
13             end
14         end
15     end
16     for each incoming neighbor n_i tagged with s do
17         ω ← receive the combined message from n_i;
18     end
19     ω ← call MPI wait;
20     for each incoming neighbor n_i tagged with s do
21         ω ← move combined received message to final buffers;
22     end
23     ω ← call MPI wait;
24 end
25 for each remaining outgoing neighbor n_o do
26     ω ← send direct message to n_o;
27 end
28 for each remaining incoming neighbor n_i do
29     ω ← receive the message from n_i;
30 end
31 ω ← call MPI wait;
32 for each remaining incoming neighbor n_i do
33     ω ← move the received message to final buffers;
34 end
```

(Lines 4 to 7). In other words, it sends its own message to its friend processes and receives the messages from its friends. Then, we add a hint to ω in Line 8 to make sure that all the processes are done with communicating with their friends. Next, process p generates the combined message which consists of its own message plus the messages of all of its friends (Line 9). In Lines 10 to 14, process p sends the combined message to the common neighbors that are assigned to it at step s.

7.5 Design Parameters Selection

In this section, we discuss selecting parameters k and θ to get the best performance from message combining mechanism. The drawback of increasing the parameter k is that it increases the overhead of generating friend groups (Section 7.3.2). This overhead is more considerable in topology-agnostic approach where the friends can be chosen from all processes in the cluster. Another disadvantage of increasing k is that it potentially decreases the number of common neighbors. On the other hand, increasing k provides the opportunity to perform message combining between larger group of friend processes. If the friend groups have a considerable number of common neighbors, this results in reducing the number of communications. So, there is a trade-off for selecting the parameters. In the following, we discuss selecting the parameters in topology-agnostic and topology-aware approaches. Note that in this section, we assume that the number of common neighbors is dividable by the number of processes in the friend group.

7.5.1 Topology-agnostic Approach

For deriving the parameters in topology-agnostic approach, we assume that a group of n processes have m common neighbors. In this graph, we can make friend groups of size k in which $k \leq n$. The question is for what value of m it is worth to make a friend group with maximum number of processes ($k = n$). As discussed earlier, in topology-agnostic approach,

increasing k increases the overhead of creating friend groups significantly. Therefore, we should not increase k unless increasing k reduces the number of communications ($N_k < N_{k-1}$). Otherwise, the overhead of increasing k is not compensated. By substituting the number of communications N_k derived from Eq. 7.1 in above-mentioned inequality, we have:

$$(m/k) + k - 1 < (m/(k-1)) + (k-1) - 1 \tag{7.3}$$

Therefore:

$$m > k \times (k-1) \tag{7.4}$$

Eq. 7.4 shows that for a friend group of size k, the number of common neighbors m should be greater than $k \times (k-1)$ to benefit from topology-agnostic approach. Therefore, we choose $\theta = k \times (k-1)$ in all the experiments for topology-agnostic design.

7.5.2 Topology-aware Approach

For selecting parameters in the topology-aware approach, we do not use the same procedure discussed in 7.5.1 for the topology-agnostic design. This is because of three main reasons:

- As discussed in Section 7.3.2, in topology-aware design, the friends are selected among intra-node processes. This provides the opportunity to remove some inter-node communications between processes and their outgoing neighbors and replace them with intra-node communications between friends, effectively providing a better performance than the topology-agnostic approach.
- The topology-aware design has less options for selecting friends since it chooses friends only among processes on the same node. This may make Eq. 7.4 a very hard condition for this approach, leading to no friend selection.

- In topology-agnostic design, increasing k increases the overhead of generating friend groups significantly. Therefore, we use Eq. 7.4 to make sure that we do not increase k unless we have a sufficient number of common neighbors. In topology-aware design, the friends are limited to intra-node processes. This alleviates the overhead of generating friend groups with increasing K.

Therefore, the topology-aware design does not have a significant overhead for a large k, and its minor overhead is compensated to some extent by replacing some inter-node communications with intra-node communications. Therefore, in the topology-aware approach, we choose to reduce the number of communications N_k to less than the default approach, m. In other words, we have:

$$(m/k) + k - 1 < m \tag{7.5}$$

Therefore,

$$m > k \tag{7.6}$$

Eq. 7.6 shows that the number of common neighbors should be greater than the number of friend processes k. Therefore, we choose $\theta = k$ in all the experiments for topology-aware design. Note that Eq. 7.4 for topology-agnostic approach also meets the condition in Eq. 7.6.

7.6 Experimental Results and Analysis

This section evaluates the performance of our proposed design on different neighborhood topologies. The evaluation is done on a microbenchmark that measures the latency of neighborhood allgather using collaborative communication mechanism and compares it with the MVAPICH default design. The MPI implementation is MVAPICH2-2.2. We use the

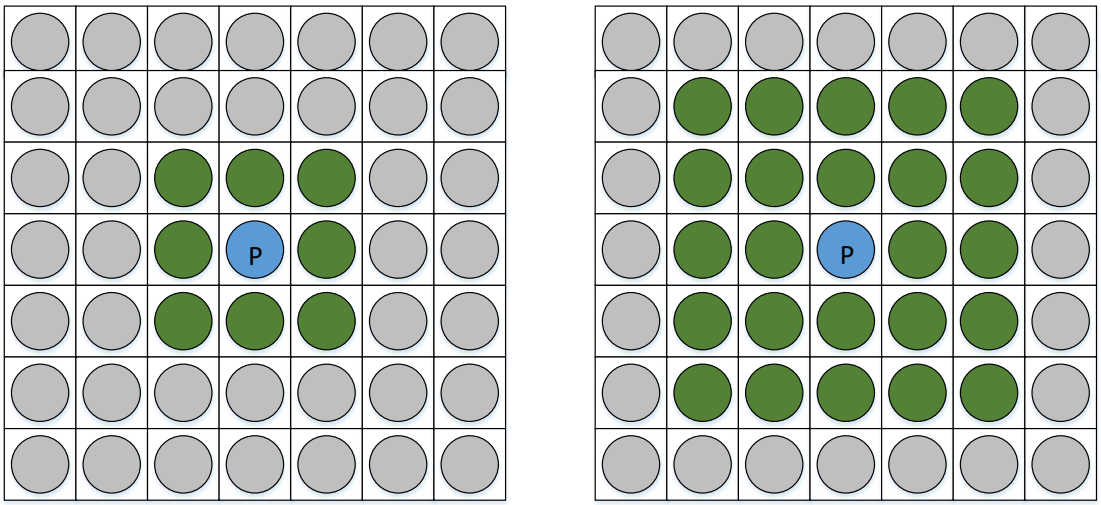
same microbenchmark used in [91]. This mircobenchmark builds the process topology graph, calls the function MPI_Ineighbor_allgather 1000 times, and then reports its average latency.

For the experiments in this chapter, we use two clusters. The first cluster is Niagra at the SciNet HPC Consortium of Compute Canada. the Niagra cluster consists of 1,500 nodes, for a total of 60,000 cores. Each node has 40 Intel Skylake cores operating at 2.4GHz, and a 202 GB of memory. It uses Mellanox EDR InfiniBand for the interconnect. We refer to Niagara as Cluster D in this dissertation. The second cluster is Cluster B in which its specifications have been presented in Section 5.3.1.

7.6.1 Moore Neighborhoods

The first topology graph that we use to evaluate the efficiency of our design is Moore neighborhood [106]. Two parameters define Moore neighborhood: dimension (d) and radius (r). The parameter d shows the number of grid dimensions for organizing the nodes or MPI processes. The parameter r shows the absolute value of the maximum distance at which other nodes are considered a neighbor of a given node. For a Moore neighborhood process topology graph with specific d and r, the number of neighbors of each node is $(2r+1)^d - 1$. Note that Moore neighborhoods are symmetric which means that each node has the same incoming and outgoing neighbor. Figure 7.4 shows a sample Moore neighborhood with d=2 and r=1, 2.

The advantage of Moore neighborhoods is that they provide regular type of (balanced) topologies. Moreover, they can be considered as a generalization of stencil patterns such as 2D 9-point or 3D 27-point. Also, experimenting with different values of d and r provides us the opportunity to model a wider variety of neighborhood shape and sizes. Moore neighborhoods have also been used by Mirsadeghi et al. [91], Träff et al. [109], [108] and also Lübbe [86]. For the Moore neighborhood experiments, we use Cluster D.



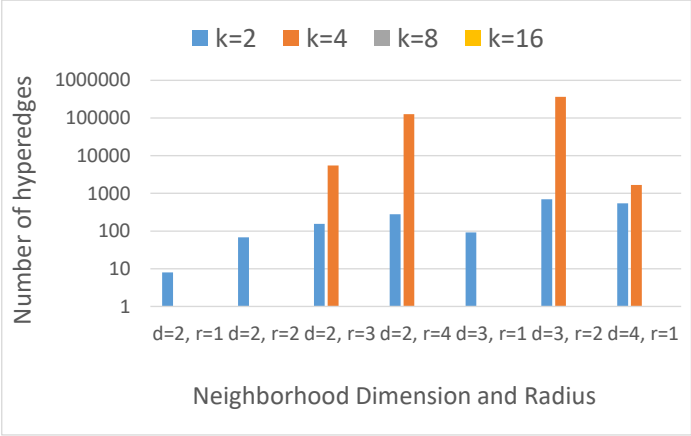
(a) d=2, r=1 (b) d=2, r=2

Figure 7.4: A sample Moore neighborhood with d = 2 and r = 1, 2. The neighbors are shown in green for node P

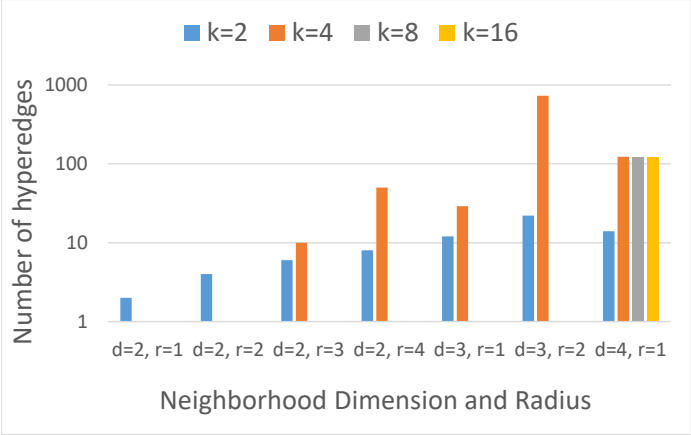
Number of Hyperedges

In this section, we discuss the number of hyperedges that are generated for each process with different values of d and r in Moore neighborhood. Figure 7.5 shows the results for different values of k in topology-agnostic and topology-aware designs. In these figures, the number of hyperedges are averaged across all processes. Note that the hypergraph is built only based on the topology graph and it is independent of the neighborhood operation and the size of the messages. One observation from these figures is that the number of hyperedges increases with the density of the topology graph. For example, for d=2, the number of hyperedges increases with increasing the neighborhood radius. Moreover, for the same radius ($r = 1$), the number of hyperedges increases with increasing the dimensions. The reason for this is that as we increase the density of the topology graph, the number of common neighbors between the processes increases. This makes the processes to have more friend groups which increases the number of hyperedges.

Another observation from Figure 7.5 is that increasing k up to a certain point increases



(a) Topology-agnostic



(b) Topology-aware

Figure 7.5: Number of hyperedges for Moore neighborhood for 4K processes on Cluster D. Missing bars represent a zero value

the number of hyperedges. However, increasing it further reduces the number of hyperedges. For example, for $d = 2$, $r = 3$ increasing k from 2 to 4 increases the number of hyperedges. However, increasing it further to $k = 8$ and $k = 16$ results in no hyperedges. This is because as we increase k, we can have more combinations of the friend processes in each friend group which results in more hyperedges. More specifically, increasing k increases the number of iterations in Line 14 of Algorithm 7.2 which results in more rows in matrix T. On the other hand, the probability of having sufficient number of common neighbors between processes in

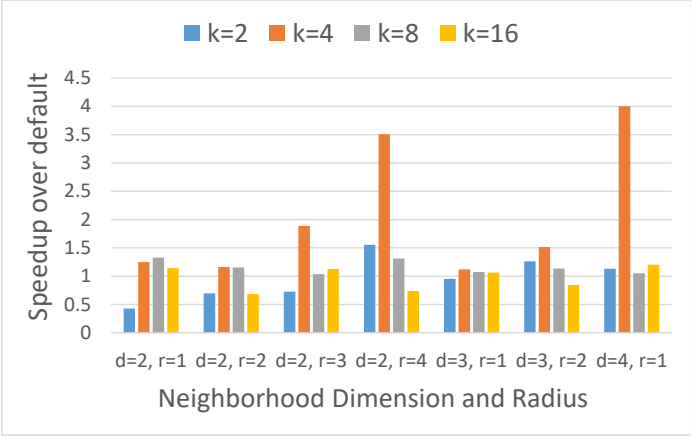
a friend group of size k decreases as we increase k. In particular, $|r|$ reduces as we increase k in Line 15 of Algorithm 7.2. Therefore, it would be less probable to meet the condition in Line 16 and consequently, the number of rows in matrix T reduces which results in fewer hyperedges.

Comparing Figure 7.5(a) with 7.5(b), we can observe that in many cases the number of hyperedges in the topology-agnostic design is considerably more than the topology-aware design. This is because in the topology-agnostic design, the processes can choose their friends from all the processes in the cluster while in topology-aware design, the friends are limited to intra-node processes. Although, there are a few cases such as $k = 8$ and $k = 16$ for d=4 that the topology-aware design has more hyperedges. This is because the parameter θ is smaller in the topology-aware design as has been discussed in Section 7.5. In these cases, each process has a few common neighbors with the processes on the same node. The number of these common neighbors is more than θ in the topology-aware design but they are not sufficient to make friend groups in topology-agnostic design which has larger θ.

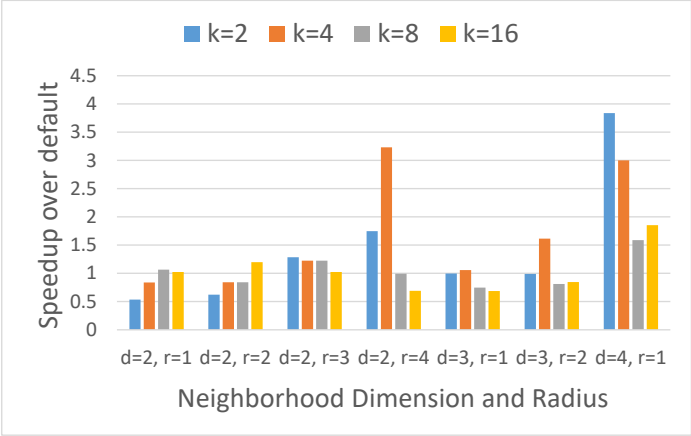
The advantage of having a large number of hyperedges is that we have more options to select the friend groups with maximum number of common neighbors. This would result in more improvement in performance of neighborhood collective operations. On the other hand, the disadvantage of having a large number of hyperedges is that it increases the required memory for saving the friend groups in matrix T. Moreover, it increases the overhead of message combining mechanism (Algorithm 7.1). We discuss the performance improvement of neighborhood allgather operation and the overhead of message combining mechanism in Moore neighborhood in the next sections.

Performance Improvement

Figure 7.6 shows the performance improvement of MPI_Ineighbor_allgather with 4 bytes message size compared to the default MVAPICH for Moore Neighborhood in topology-agnostic and topology-aware designs, respectively. These figures show that we can obtain



(a) Topology-agnostic



(b) Topology-aware

Figure 7.6: MPI_Ineighbor_allgather speedup for Moore neighborhood for 4K processes-4 byte message size on Cluster D

up to around 4x performance improvement in both topology-aware and topology-agnostic designs. The speedup increases with the density of the topology graph. That is because of the large number of neighbors in denser graphs which provide more opportunity for performance improvement. In the cases that the processes have a small number of common neighbors (such as $d = 2$, $r = 1$), we observe a slight performance degradation. The reason for this is that the message combining approach requires some time for the communication between friend processes. This is not compensated in topology graphs that have

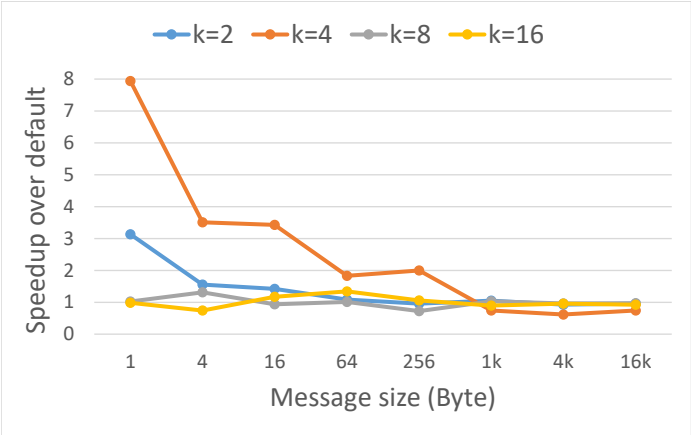
a small number of common neighbors. Note that the results in this figure is in concert with the results in Figure 7.5. As the number of hyperedges increases, more performance improvement is obtained. This is because with a large number of hyperedges, we have more options to select friend groups with maximum number of common neighbors which leads to more speedup. Comparing Figure 7.6(a) and 7.6(b), we can observe that in many cases, the performance improvement of the topology-aware approach is close to the performance improvement in the topology-agnostic approach and in some cases (such as $k = 2$ when d=4, r=1), it is even better. This shows the efficiency of the topology-aware approach that provides comparable performance improvement despite the fact that it has significantly less number of hyperedges compared to the topology agnostic approach (Figure 7.5). This performance gain is because of the fact that the friends are chosen among intra-node processes which reduces the communication time between the processes in a friend group.

Figure 7.7 shows the speedup for $d = 2$, $r = 4$ with different message sizes. This figure shows that the speedup reduces with increasing the size of the message and it reaches around 1 for message sizes greater than 1KB. This is because the message combining approach does not benefit the communication pattern in regard to its bandwidth. Therefore, it does not improve the communication performance of large messages, which are sensitive mainly to bandwidth characteristics of communication patterns. Note that for a message of size s, the size of the combined message is $k \times s$.

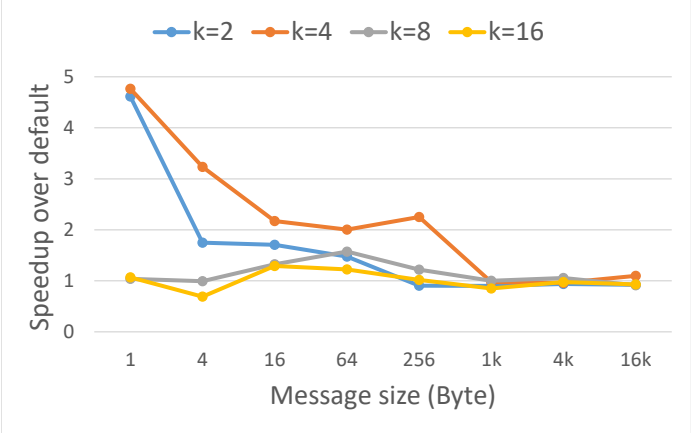
Another observation from Figure 7.7 is that the performance improvement for $k = 8$ and $k = 16$ is around 1. As can be seen in Figure 7.5, when $k = 8$ and $k = 16$, the number of hyperedges for $d = 2$ and $r = 4$ is 0. So there is no friend groups for message combining in these cases which leads to no speedup.

Overhead Analysis

Figure 7.8 shows the overhead of making the communication pattern in Algorithm 7.1. This overhead is encountered only once for a given topology and it is independent of the



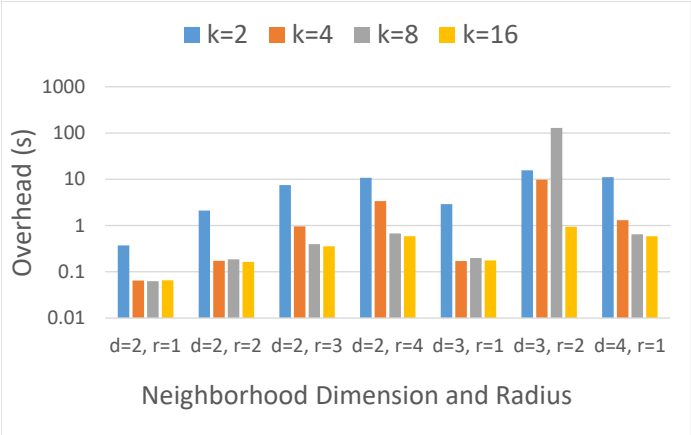
(a) Topology-agnostic



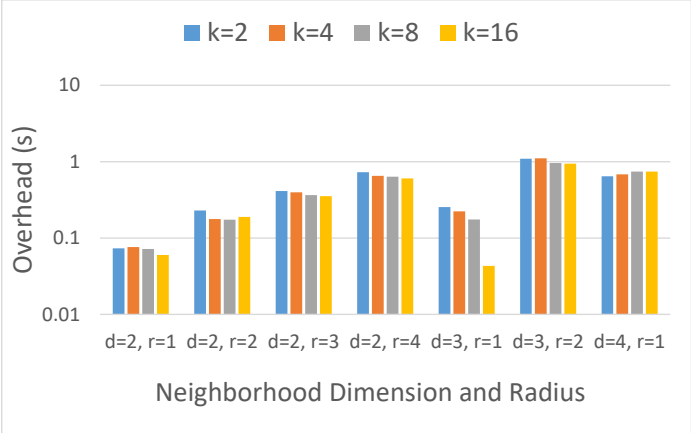
(b) Topology-aware

Figure 7.7: MPI_Ineighbor_allgather speedup for Moore neighborhood with d=2, r=4 for different message sizes and 4K processes on Cluster D

neighborhood operation and the message size. Comparing these figures with Figure 7.5, we can observe that the overhead of making the communication pattern has a direct relationship with the number of hyperedges. This makes sense since it takes more time to find the best friend group among a larger number of hyperedges. Figure 7.8(b) shows that in almost all cases, the overhead of making the communication pattern in topology-aware approach is less than 1s. This shows the efficiency of the topology-aware approach compared to the topology-agnostic approach (Figure 7.8(a)).



(a) Topology-agnostic



(b) Topology-aware

Figure 7.8: The overhead of Algorithm 1 for Moore neighborhood with 4K processes on Cluster D

7.6.2 Random Sparse Graph

The second topology graph that we use to evaluate the efficiency of our design is Erdös-Renyi random sparse graph $G(V, E)$ [42]. In this graph, V corresponds to the set of MPI processes, and an edge $(i, j) \in E$ shows an outgoing neighbor from process i to process j. In random sparse graph, the outgoing neighbors for each process are created randomly. The number of outgoing neighbors depends on the density factor δ ($0 < \delta < 1$). A higher δ means a denser graph. The same graph has also been used in other neighborhood collective

works such as Hoefler and Schneider [60] and Mirsadeghi et al. [91]. We show the results with different values of δ to evaluate the collaborative communication mechanism with neighborhood topologies of different shapes and sizes. We use Cluster B for the experiments in this section.

Section 7.6.1 shows that in the Moore neighborhood, the topology-aware approach provides a better or equal speedup as the topology-agnostic approach and at the same time, it has lower overhead and memory consumption (number of hyperedges). Random sparse graph has denser communications compared to Moore neighborhood. Therefore, in this topology graph, the efficiency gap between the topology-aware and topology-agnostic approaches is even more. More specifically, the overhead and memory consumption of the topology-agnostic approach make it non-feasible for dense topology graphs such as random sparse graph. So, in this section, we show the results just for the topology-aware approach. Note that the dense communication of random sparse graph makes this topology more sensitive to the parameter k. Therefore, in this section, we conduct the experiments with the more fine-grain k values. This way we could measure the impact of small variations of k on the performance. Moreover, the performance gain reaches a steady state for $k \geq 6$. Therefore, we show the results up to $k = 6$.

Number of Hyperedges

Figure 7.9 shows the number of hyperedges in random sparse graph for different k and δ. As can be seen in this figure, the number of hyperedges increases with increasing δ. This is because increasing the edge density increases the number of neighbors which provides the opportunity for the processes to make more friend groups. Another observation from this figure is that for $\delta = 0.05$, $k \geq 3$ results in no hyperedges. That is because of the small number of communications in $\delta = 0.05$ that results in no or a few common neighbors between groups of size greater than 3. The same thing happens for $k \geq 6$ when $\delta = 0.2$. For large edge densities ($\delta \geq 0.4$), increasing k increases the number of hyperedges. That

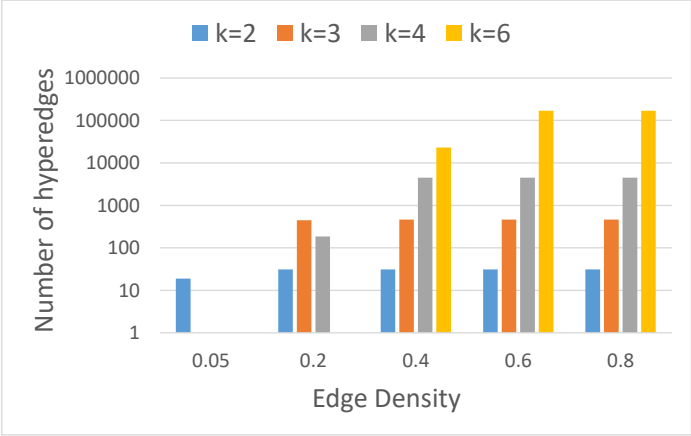


Figure 7.9: Number of hyperedges in Random sparse graph with topology-aware design and 1K processes on Cluster B. Missing bars represent a zero value

is because in dense topology graphs, the number of common neighbors is large enough to make friend groups of a large size ($k = 6$). Moreover, as we increase k, we can have more combinations of the friend processes in each friend group which results in more hyperedges.

Performance Improvement

Figure 7.10 shows the neighborhood allgather speedup for 4 byte message size with different k and δ. This figure shows that the k value in which provides the highest speedup increases with increasing the edge density. For example, for $\delta = 0.05$ and $\delta = 0.2$, $k = 2$ provides the highest speedup. For $\delta = 0.4$, $k = 3$ has the highest speedup and for $\delta = 0.6$ and $\delta = 0.8$, the speedup gained by $k = 4$ is the highest. To understand the reason behind this, we should consider the negative and positive impact of increasing k on the performance. The advantage of increasing k is that it provides the opportunity to perform message combining between larger group of friend processes. However, it reduces the number of common neighbors in the hyperedges. Therefore, increasing k is not beneficial unless the topology graph is dense enough to have sufficient number of common neighbors for large k. For example, when $\delta = 0.2$, we observe 2.6x performance improvement with $k = 2$. When we increase k to 3 and 4, the number of common neighbors reduces. With a few common neighbors, the

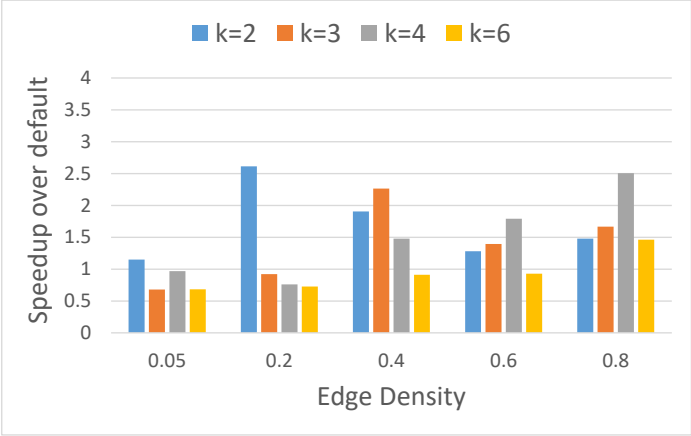
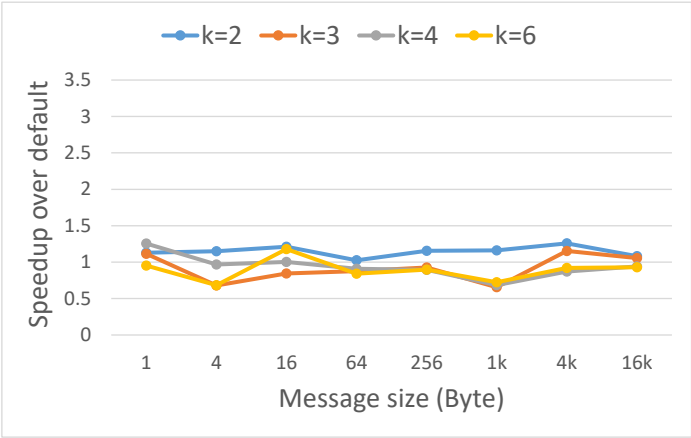


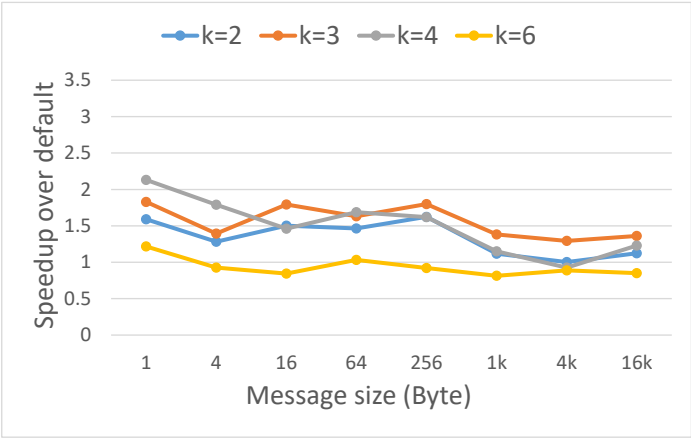
Figure 7.10: MPI_Ineighbor_allgather speedup in Random sparse graph with topology-aware design and 1K processes- 4 bytes message size on Cluster B

overhead of communication between k friend processes is not compensated and the speedup drops. For $k = 6$, there is no hyperedge (Figure 7.9) and we do not have any speedup. When we increase the edge density to $\delta = 0.4$, a group of $k = 3$ friend processes have sufficient common neighbors to gain speedup of 2.2x. However, for groups of size $k \geq 4$, the number of common neighbors reduces and the speedup drops. For denser topology graphs ($\delta = 0.6$ and $\delta = 0.8$), groups of $k = 4$ friend processes can have sufficient common neighbors to provide the maximum speedup.

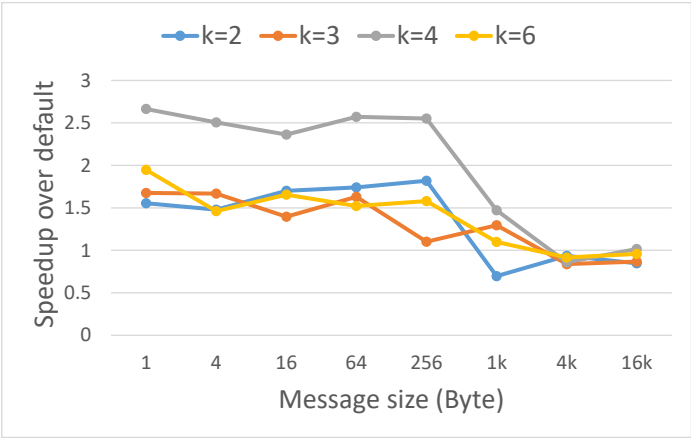
Figure 7.11 shows the neighbor allgather speedup for different message sizes and three different edge densities. Comparing Figure 7.11(a), 7.11(b) and 7.11(c), we can observe that the speedup increases with increasing δ. That is because as we increase δ the number of communications increases which provide more room for performance improvement. For $\delta = 0.05$, the speedup is around 1 for almost all cases. That is because of the small number of common neighbors in $\delta = 0.05$ that make the default approach good enough in this case. For $\delta = 0.6$ and $\delta = 0.8$, the speedup reduces with increasing the message size and it reaches around 1 for message sizes greater than 1KB. This is because of the bandwidth effect of the larger message sizes.



(a) δ=0.05



(b) δ=0.6



(c) δ=0.8

Figure 7.11: Neighbor allgather speedup for Random Sparse Graph with topology-aware design for different message sizes and 1K processes on Cluster B

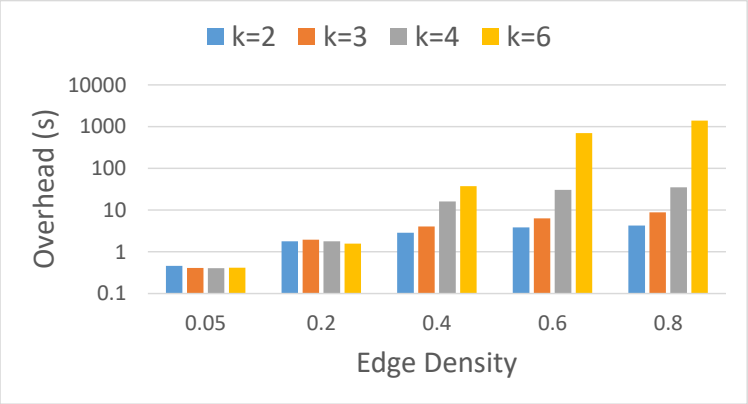


Figure 7.12: The overhead of Algorithm 1 for random sparse graph with topology-aware design for 1K processes on Cluster B

Overhead Analysis

Figure 7.12 shows the overhead of making the communication pattern for different k and δ. This overhead happens only once for each topology graph. The results in this figure is in concert with the results in Figure 7.9. As discussed in Section 7.6.1, the overhead increases with increasing the number of hyperedges. This is because it takes more time to make the communication pattern among a larger number of friend groups.

One interesting observation from Figure 7.10 and Figure 7.12 is that the k values that provide more speedup has considerably less overhead compared to larger k values with less speedup. For example, when $\delta = 0.4$, $k = 3$ provides the maximum speedup of 2.26x. In this case, the overhead of generating the communication pattern is 4 seconds. When we increase k to 4 or 6, the speedup reduces and at the same time we have more overhead. This shows that by choosing k appropriately we can gain the maximum speedup with less overhead and memory consumption (number of hyperedges).

Comparing Figure 7.12 with Figure 7.8(b), we can observe that the overhead of making the communication pattern in Moore neighborhood is considerably less than random sparse graph (1s compared to 30s to gain the maximum speedup). This is because Moore has more structured and localized neighborhood topology compared to random sparse graph.

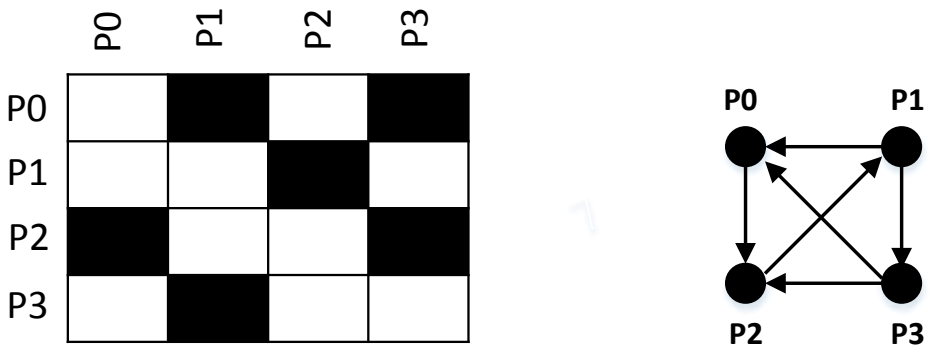
Considering the fact that many real applications have structured and localized neighborhood topologies, this shows the efficiency of the proposed topology-aware design.

7.6.3 Application Results and Analysis

In this section, we use a sparse matrix-matrix multiplication (SpMM) kernel to show the benefits and practicality of neighborhood collectives and our proposed design to optimize them. SpMM is an important kernel used in computational linear algebra and big data analytics [19]. The performance bottleneck of SpMM is the cost of communications that are performed in each iteration to gather specific rows/columns of the matrices that are distributed among all processes. We show how MPI neighborhood collectives can help to reduce the cost of such communications in SpMM. We also show how our proposed collaborative communication mechanism for neighborhood collectives can further improve the performance of SpMM.

The SpMM problem can be defined as $A_{ml} \times B_{ln} = C_{mn}$, where A is a sparse matrix of size $m \times l$ which should be multiplied by matrix B of size $l \times n$ to get the result matrix C of size $m \times n$. To parallelize the multiplication, we use a row-wise block-striped decomposition [77] of A and B to evenly distribute the rows of A and B among the set of all processes. The kernel will then consist of l iterations with a call to `MPI_Allgather` in each iteration. With p processes, each process will compute $\frac{m}{p}$ rows of the result matrix C.

The `MPI_Allgather` used in each iteration of SpMM is used to gather a column of B at all processes. However, we can take advantage of the sparsity of A to replace the costly global `MPI_Allgather` operation with `MPI_Neighbor_allgather`. This is because every process needs to gather only those elements of each column of B that have a corresponding non-zero value in the rows of A that are assigned to that process. Thus, we can use A to define a process topology over which the `MPI_Neighbor_allgather` is called in each iteration. Figure 7.13 provides an example of how this is done for a sparse matrix A of size 4×4. The 0 and 1 elements of the matrix are represented by white and black squares, respectively.



(a) Matrix A (b) Topology grap

Figure 7.13: An example of building the process topology graph for the SpMM kernel based on the non-zero elements of the input matrix A

As shown by the figure, the rows and columns of matrix A are evenly distributed among all processes. For the sake of clarity, in this example we use a number of processes that is equal to the number of rows and columns of A. Hence, process P_j (P_i) will be an incoming (outgoing) neighbor of process P_i (P_j) in the process topology graph if and only if $A_{ij} = 1$.

For performance evaluation, we use four matrices from the SuiteSparse Matrix Collection (formerly the University of Florida Sparse Matrix Collection) [39], namely 'ash292', 'human_gene2', 'journals' and 'dwt_193'. We choose these matrices to evaluate the impact of the proposed neighborhood collective optimization on matrices with different densities and dimensions. In each case, we use the same matrix to provide the input matrices for SpMM. In other words, we multiply each given matrix by itself. All experiments are conducted on Cluster B as described in Section 5.3.1.

Figure 7.14 shows the achieved performance improvement for SpMM when we use neighborhood collectives instead of ordinary collectives. As shown by the figure, for all input matrices, we have been able to speed up SpMM by using neighborhood collectives. In particular, we can see more than 100x speedup for 'human_gene2'. Moreover, we see higher speedup for 'ash292' and 'human_gene2' compared to 'journals' and 'dwt_193'. This is because 'ash292' and 'human_gene2' are more sparse than 'journals' and 'dwt_193', which

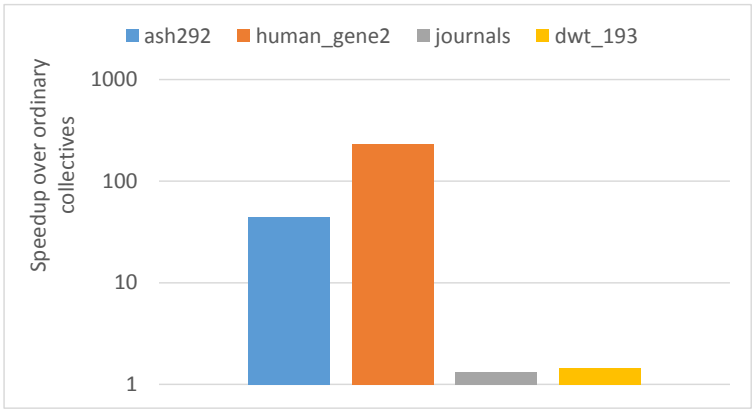


Figure 7.14: The speedup of SpMM for various input matrices with neighborhood collectives over ordinary collectives on Cluster B

provides more opportunity to decrease communication costs through neighborhood collectives. Higher sparsity of the input matrix results in fewer processes communicating with each other in the neighborhood collective.

Figure 7.15 shows how our proposed design for neighborhood collectives further improves the performance of SpMM. The figure shows the achieved speedup over the default neighborhood collective used in MVAPICH for different values of the friend group size k. We can see that in almost all cases, we have been able to achieve further speedup by using our proposed design for neighborhood collectives. The only exception is 'human_gene2' for which we get the same performance as the default neighborhood collective design. This is because the sparsity of the 'human_gene2' matrix is so high that its corresponding process topology graph does not provide enough common neighbors to take advantage of. However, for relatively denser matrices such as 'journals' and 'dwt_193', we can achieve high speedups.

7.7 Summary

In this chapter, we propose a collaborative communication mechanism between groups of k processes to improve the performance of MPI neighborhood collectives. We show that part

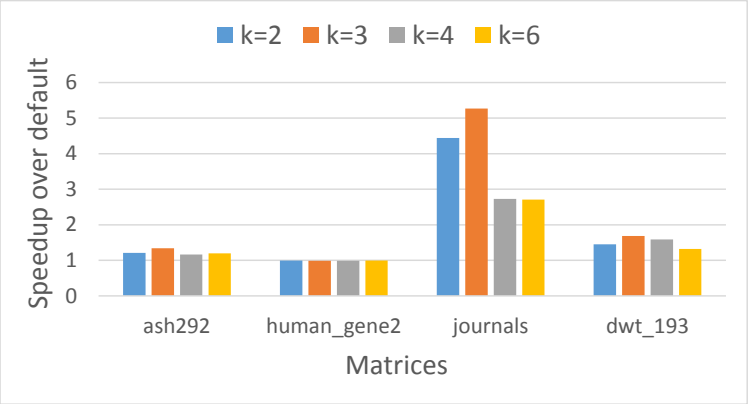


Figure 7.15: The speedup of SpMM for various input matrices with optimized topology-aware neighborhood collectives over the default neighborhood collectives on Cluster B

of the problem falls within the scope of maximum matching in weighted hypergraphs, where we seek to find a mutual grouping of the processes that have neighbors in common. For this, we propose a distributed algorithm that finds the maximum matching in a weighted hypergraph. We propose two different types of the collaborative communication mechanism: topology-agnostic and topology-aware. The topology-agnostic approach assumes a flat physical topology for the system. While the topology-aware approach considers the hierarchical physical topology of the system in the design. The evaluation results shows that in most cases the performance gain of the topology-aware approach is equal or even better than the topology-agnostic approach and at the same time, it has considerably less overhead and memory consumption. We also present the results with different values of k and show that by choosing k appropriately, we can gain considerable performance speedup (up to 8x).

Chapter 8

Conclusions and Future Work

8.1 Conclusion

Inter-process communication is one of the most important challenges in HPC and its efficiency significantly affects the performance of parallel applications. This is particularly important considering the fast pace at which the number of processing elements in HPC systems is increasing. In this dissertation, we consider different research approaches to improve the performance of communication in MPI and the applications that use them. In the first research direction, we proposed algorithms and mechanisms to improve the performance of message matching that is in the critical path of communication. For this, we considered both the behavior of applications and the hardware/software features of parallel computing systems. In the second research direction, we took advantage of the physical and virtual topology information to improve the performance of neighborhood collective communications in HPC. Physical topology represents the connections between the cores, chips, and nodes in the hardware; while virtual topology represents the way that MPI processes communicate. In the remainder of this section, we will highlight the contributions of the research in this dissertation.

Clustering-based Message Queue Data Structure

In Chapter 3, we proposed a new message matching mechanism that categorizes the

processes into groups based on the number of queue elements each process adds to the message queue at runtime. Then, a dedicated message queue is allocated to each group. We used two different algorithms for grouping the processes: K-means clustering and a heuristic algorithm. The proposed message queue mechanism provides two advantages. First, it parallelizes the search operation by using multiple queues. Moreover, it further speeds up the search operation by considering the message queue behavior of the peer processes in grouping the processes and allocating dedicated message queue to each group. The experimental evaluation showed that the proposed approach can successfully decrease the number of queue traversals. Moreover, it improves the queue search time and application runtime by up to 44.2x and 2.4x for the FDS application which has extreme message matching requirements, respectively.

Partner/Non-partner Message Queue Data Structure

Despite the fact that the clustering-based approach proposed in Chapter 3 improves the queue search time, it has some limitations. First, it requires a large amount of memory for maintaining the information about peer processes. Moreover, it is a static approach meaning that the application should be executed once to gather the profiling information. In Chapter 4, we proposed a new message queue mechanism based on the notion of partner non-partner traffic to address the above issues. This approach uses queue profiling information to categorize the processes into two groups: partners and non-partners. Partners are the processes that send/post a large number of messages/receives to the queues, while the non-partner processes only have a few elements in the queues. Each partner process has its own dedicated message queue, while the non-partner processes share a single queue. This approach deals with the memory scalability issue of the clustering-based approach by maintaining the information of just partner processes in a hash table rather than saving the information of all processes in an array.

We proposed both a static and a dynamic version of our design. The Static approach

works based on the information from a profiling stage, while the Dynamic approach utilizes the message queue characteristics at runtime. The Dynamic approach is more practical than the Static approach. However, we present a static version of our design to compare its performance with the Dynamic approach.

The experimental results showed that the proposed partner/non-partner message queue data structure can reduce the queue search time for long list traversals while maintaining the performance for short list traversals. More specifically, The evaluation results show that the queue search time and application runtime are improved by up to 28x and 5x for applications with long queue traversal, respectively. Moreover, it provides scalable memory consumption.

A Unified, Dedicated Message Matching Engine for MPI Communications

In Chapter 5, we looked at the message matching issue from a different perspective and we proposed a new queue data structure that considers the type of communication to further improve the message matching performance. This approach separates the queue elements based on their type of communication (point-to-point and collective). For queue elements that are coming from collective communications, we propose a message matching mechanism that dynamically profiles the impact of each collective call on message queue traversals and uses this information to adapt the message queue data structure (the COL approach). For the point-to-point queue elements, we use the partner/non-partner message queue data structure proposed in Chapter 4 (the PNP approach). We refer to the unified design as the COL+PNP approach.

We compare the COL+PNP performance gain with COL+LL. In the COL+LL approach, the COL approach is used for collective communications and a MVAPICH linked list queue data structure is used for point-to-point communications. The evaluation results show that the COL+PNP approach provides similar or better performance compared to COL+LL. Through experimental evaluations, we show that by allocating 194 queues

for point-to-point elements and 416 queues for collective elements, we can gain up to 5.5x runtime speedup for the FDS application.

Message Matching Improvement on Modern Architectures

Chapter 3 to Chapter 5 proposed mechanisms that work based on the application behavior. In contrast, Chapter 6 considers hardware/software features of the modern architectures to improve the performance of message queue operations. More specifically, this chapter proposed two techniques to take advantage of the vectorization capabilities of many-core processors to enhance the performance. In the first technique, we restructured the message queue design to linked list of arrays. This provides the opportunity to reduce the number of memory references compared to the default linked list data structure. Moreover, it uses special locality by allowing loading of queue elements into the cache. In the second approach, we consider linked list of vector technique to take advantage of vector operations such as Intel's AVX intrinsics to further improve the performance. Finally, we evaluated the performance gain on Intel Xeon Phi KNC coprocessor as well as Xeon processors. The evaluation results on three different microbenchmarks showed that the proposed approaches can improve the queue search time on Xeon Phi many-core coprocessor by up to 35.3x. Moreover, we could improve the queue search time and the application runtime by up to 4.5x, 2.72x for applications with long list traversals on the Xeon processors.

MPI Neighborhood Collective Optimization

In Chapter 7, we proposed an algorithm to improve the performance for neighborhood collective communications in MPI. Neighborhood collectives represent a relatively new type of communication in MPI that greatly increase the importance and employment of MPI process topologies. We showed that useful information can be extracted from a given process topology to optimize the performance of neighborhood collectives. More specifically, we took advantage of common neighborhoods in the process topology graph to design an

efficient communication pattern. For this, we proposed a distributed maximum matching algorithm in weighted hypergraph. The derived communication pattern is then used to build message-combining communication schedules for neighborhood collectives

We proposed a topology-agnostic as well as a topology-aware design. The topology-agnostic design assumes that the system has a flat physical topology while the topology-aware design considers the hierarchical physical topology of the system in the design.

Our experimental results with various process topologies showed that we can gain up to 8x reduction in the communication latency of the MPI_Ineighbor_allgather. Our results also showed up to 230x speedup for a sparse matrix-matrix multiplication kernel when we replace conventional collectives with neighborhood collectives. We further improve the performance by up to 5.6x by using collaborative communication mechanism for neighborhood collectives.

8.2 Future Work

Our future research plans revolve around developing designs that can tackle the major communication challenges in MPI. For this, we will consider the behavior of parallel applications and properties of the new parallel computing systems. In the following, we will discuss the opportunities to the algorithms, designs and mechanisms proposed in this dissertation.

Clustering-based Message Queue Data Structure

In Chapter 3, we used message queue profiling information to categorize the processes into some groups. We then assigned a dedicated message queue to each group. The proposed clustering approach in this chapter is more suitable for applications with a static communication profile and those that do not create additional communicating processes at runtime. We intend to extend this work to a dynamic clustering approach that could dynamically capture the application queue characteristics, use this information for clustering the processes and manage the message queues accordingly. As another direction for future

work, we would like to test the efficiency of this design with other clustering methods such as Support Vector Machine (SVM) clustering algorithm [104] and mean-shift clustering [37].

Partner/Non-partner Message Queue Data Structure

In Chapter 4, we took advantage of sparse communication pattern in the MPI applications to propose a new message matching design that allocates a dedicated message queue for the processes with high frequency of communications called partner processes. We used three different metrics for determining the partner processes: average, medium and outliers. We would like to evaluate the efficiency of this work on other applications and larger systems.

A Unified, Dedicated Message Matching Engine for MPI Communications

As another direction for future work, we intend to extend the proposed message queue design in Chapter 5 to support multi-threaded communications. Most of the traditional HPC applications use process-level parallelism [58] in which each process possesses a single thread. Processes can also be multithreaded in which case they host several threads and can exhibit parallelism internally. Recently, applications are moving towards fully-multithreaded runtime [28]. On the other hand, it has been shown that message matching becomes more of a challenge when using multiple threads [101]. However, most MPI message matching mechanisms are designed for single-threaded communications. Therefore, proposing a message matching mechanism for multi-threaded communications is of paramount importance.

Message Matching Improvement on Modern Architectures

In Chapter 6, we explored mechanisms to take advantage of vectorization capabilities of hardware architectures to improve the performance of MPI message matching. We presented the results on both Intel Xeon Phi many-core coprocessors and Xeon processors. In the future, we would like to do a sensitivity analysis to evaluate the impact of increasing the

array_size on performance.

As another direction for future work, we intend to take advantage of the hardware features of other popular hardware designs in HPC such as GPU (Graphics Processing Units) to improve message matching mechanism. GPUs, among other coprocessors and accelerators, have successfully established themselves in HPC clusters because of their high performance and energy efficiency. These factors are the key requirements of future supercomputers, thus paving the way for GPUs to be continually used in current petaflop and future exascale systems. GPUs consist of thousands of processing units. The application dataset is distributed among these processing units to parallelize and accelerate computation. GPUs are good candidate for offloading and accelerating compute-intensive portions of HPC applications due to their massive computational capabilities. Thus, we would like to evaluate the performance of message queue operations on GPUs and explore techniques that can be leveraged to improve message matching performance, and consequently, the application runtime. We would also like to investigate taking advantage of scatter/gather vector addressing instead of memory copying for making the vectors in linked list of vector data structure.

Note that the linked list of array data structure evaluated in this chapter is orthogonal to the proposals in Chapters 3 to 5. In other words, we can take advantage of a linked list of array instead of a linked list for all the allocated queues in Chapters 3 to 5. This will provide the opportunity to leverage the properties of the applications at the user level and the hardware features at lower layers at the same time.

MPI Neighborhood Collective Optimization

In Chapter 7, we proposed a collaborative communication mechanism to improve the performance of neighborhood communications. Neighborhood collectives are a relatively new feature added to MPI. Therefore, HPC applications are not yet ported. As a future work, we intend to port current applications so that they can take advantage of the many

benefits of the neighborhood collectives. More specifically, we plan to replace groups of point-to-point neighbor communications in current applications with appropriate calls to neighborhood collectives. This way, we would be able to evaluate the efficiency of the proposed collaborative design on the performance of real applications.

One important factor that has considerable impact on the performance of collaborative communication mechanism is the selection of the parameters k and θ. The advantage of increasing k is that it provides the opportunity to perform message combining between larger group of friend processes. However, the drawback of increasing k is that it enhances the overhead of generating friend groups in Algorithm 7.2. Moreover, increasing k potentially decreases the number of common neighbors. Therefore, there is a trade-off for selecting the parameter k.

In regard to parameter θ, the advantage of increasing it is that it reduces the number of friend groups for each process which results in less overhead in Algorithm 7.2 and 7.3. The drawback of increasing θ, however, is that it reduces the opportunities for message combining. A metric that can be used to guide the appropriate selection of the k and θ value is the density and structure of the topology graph. For dense topology graph, it might be better to use a higher value for k and θ. The reason is that a dense topology graph will result in a dense friendship graph where each node will have the opportunity to make more friend groups along heavy-weight edges. Therefore, we will have the chance to make friend groups of larger size (k) with considerable number of common neighbors (θ). In Section 7.3, we discussed how to choose θ for a given k to gain performance in the topology-agnostic and topology-aware designs. In the future, we intend to do more theoretical analysis to dynamically select the best values of k and θ in for a given topology graph.

In addition, we would like to propose an algorithm to improve the overhead of the proposed collaborative communication mechanism. This includes the overhead of making the friendship matrix in Algorithm 7.2, the communications in Algorithm 7.3 for mutual grouping of the processes, and also the memory consumption for saving the friend groups.

For example, one way to reduce the overhead of communication is to take advantage of the remote memory access (RMA) operations of MPI. This feature provides this opportunity to avoid a large fraction of the point-to-point communications that are performed between a process and its friends.

We also would like to evaluate the efficiency of the proposed design on other neighborhood collectives such as neighbor alltoall and neighbor alltoallw. Using the proposed approach for neighbor alltoallw is more challenging since it requires transferring type information among certain processes which potentially requires extra communications. We are also interested in evaluating performance and overhead of our design when it is used with other distributed maximum matching algorithms.

Another direction for future work is to take advantage of *cummulative combining* in the proposed collaborative communication mechanism. In the current design, each outgoing neighbor is considered in at most one message-combining round. This is shown in Line 23 of Algorithm 7.1. As can be seen, the offloaded and onloaded common neighbors at each message-combining iteration are removed from the set of active outgoing neighbors O_a. In cumulative combining, however, the onloaded neighbors are still considered for the message combining opportunities in further iterations. More specifically, in cummulative combining, just the offloaded neighbors are removed from O_a at each iteration. Such a nested combining will provide the opportunity to further decrease the number of individual communications that are performed by each process.

One direction for future work is to consider the physical topology of the system to reduce the number of communications for the processes that are far from each other (e.g. processes on different nodes). In each node, there is a leader process who is responsible to get the messages from incoming processes on other nodes and send it to the intra-node processes. This approach is beneficial for neighborhood allgather since in this neighborhood collective, the same message is sent to other processes. Moreover, it is beneficial for both small and large messages. However, it probably benefits large messages more since it reduces

bandwidth for inter-node communications.

Finally, we would like to improve the performance of deep learning frameworks by taking advantage of neighborhood collectives. For this, the framework should have two features: first, the communication between the neurons should have the potential to be modeled as allgather, alltoall or allreduce collective operations. Second, the communication between the neurons should be sparse. We intend to evaluate the specifications of deep learning frameworks to see if they can benefit from MPI neighborhood communications.

References

[1] Cori - cray xc40. https://www.top500.org/system/178924. Accessed: October 8, 2018.

[2] Generating perfect hash function. http://www.drdobbs.com/architecture-and-design/generating-perfect-hash-functions/184404506. Accessed: May 20, 2017.

[3] IBM Spectrum MPI. http://www-03.ibm.com/systems/spectrum-computing/products/mpi/index.html. Accessed: August 20, 2018.

[4] InfiniBand Trade Association. http://www.infinibandta.org. Accessed: September 16, 2018.

[5] Intel MPI Library. https://software.intel.com/en-us/intel-mpi-library. Accessed: January 28, 2019.

[6] Knights corner: Open source software stack. https://software.intel.com/en-us/blogs/2012/06/05/knights-corner-open-source-software-stack. Accessed: October 20, 2018.

[7] Mellanox HPC-X™ Software Toolkit. www.mellanox.com/products/hpcx/. Accessed: December 28, 2018.

[8] Message Passing Interface (MPI-3),. http://www.mpi-forum.org. Accessed September 20, 2018.

[9] Miniamr-a miniapp for adaptive mesh refinement. http://hdl.handle.net/2142/91046. accessed: February 4, 2019.

[10] MPICH: High-Performance Portable MPI. http://www.mpich.org. Accessed: October 27, 2018.

[11] MVAPICH2. http://www.mpich.org. Accessed: January 24, 2019.

[12] Oakforest-pacs. https://www.top500.org/system/178932. Accessed: October 2, 2018.

[13] Open fabric interfaces (OFI). https://ofiwg.github.io/libfabric/. Accessed: January 12, 2019.

[14] OPEN MPI. https://www.open-mpi.org. Accessed: October 23, 2018.

[15] PGAS Forum,. http://www.pgas.org/ Accessed November 3, 2018.

[16] RDMA Consortium. http://www.rdmaconsortium.org. Accessed:December 12, 2018.

[17] The OpenMP API specification for parallel programming,. https://www.openmp.org/ Accessed January 29, 2019.

[18] TOP500,. https://www.top500.org/. Accessed January 2, 2019.

[19] S. Acer, O. Selvitopi, and C. Aykanat. Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems. *Parallel Computing*, 59:71–96, 2016.

[20] G. Almási, P. Heidelberger, J. Archer, X. Martorell, C. Erway, J. Moreira, B. Steinmacher-Burow, and Y. Zheng. Optimization of MPI collective communication on bluegene/l systems. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 253–262. ACM, 2005.

[21] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, and G. Chen. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International Conference on Machine Learning*, pages 173–182, 2016.

[22] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, and S. Williams. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[23] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, R. Thakur, and J. Träff. MPI on millions of cores. *Parallel Processing Letters*, 21(01):45–60, 2011.

[24] P. Balaji, A. Chan, W. Gropp, R. Thakur, and E. Lusk. The importance of non-data-communication overheads in MPI. *International Journal of High Performance Computing Applications*, 24(1):5–15, 2010.

[25] B. W. Barrett, R. Brightwell, R. Grant, S. D. Hammond, and K. S. Hemmert. An evaluation of MPI message rate on hybrid-core processors. *The International Journal of High Performance Computing Applications*, 28(04):415–424, 2014.

[26] B. W. Barrett, R. Brightwell, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. B. Maccabe, and T. Hudson. The portals 4.1 network programming interface. *Sandia National Laboratories, Tech. Rep. SAND2017-3825*, 2017.

[27] M. Bayatpour, H. Subramoni, S. Chakraborty, and D. K. Panda. Adaptive and dynamic design for MPI tag matching. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10. IEEE, 2016.

[28] D. E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz, and G. R. Vallee. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience*, page e4851, 2017.

[29] G. Berti and J. L. Träff. What MPI could (and cannot) do for mesh-partitioning on non-homogeneous networks. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 293–302. Springer, 2006.

[30] R. Brightwell, S. Goudy, and K. Underwood. A preliminary analysis of the MPI queue characterisitics of several applications. *International Conference on Parallel Processing, 2005. ICPP 2005.*, pages 175–183, 2005.

[31] R. Brightwell, S. P. Goudy, A. Rodrigues, and K. D. Underwood. Implications of application usage characteristics for collective communication offload. *International Journal of High Performance Computing and Networking*, 4(3-4):104–116, 2006.

[32] R. Brightwell, K. Pedretti, and K. Ferreira. Instrumentation and analysis of MPI queue times on the seastar high-performance network. In *Proceedings of 17th International Conference on Computer Communications and Networks, 2008. ICCCN'08.*, pages 1–7. IEEE, 2008.

[33] R. Brightwell and K. D. Underwood. An analysis of NIC resource usage for offloading MPI. In *Proceedings of 18th International Conference on Parallel and Distributed Processing Symposium, 2004.*, page 183a. IEEE, 2004.

[34] R. Budruk, D. Anderson, and T. Shanley. *PCI express system architecture*. Addison-Wesley Professional, 2004.

[35] A. Castro, H. Appel, M. Oliveira, C. A. Rozzi, X. Andrade, F. Lorenzen, M. A. Marques, E. Gross, and A. Rubio. Octopus: a tool for the application of time-dependent density functional theory. *physica status solidi (b)*, 243(11):2465–2488, 2006.

[36] Yuk Hei Chan and Lap Chi Lau. On linear and semidefinite programming relaxations for hypergraph matching. *Mathematical programming*, 135(1-2):123–148, 2012.

[37] Y. Cheng. Mean shift, mode seeking, and clustering. *IEEE transactions on pattern analysis and machine intelligence*, 17(8):790–799, 1995.

[38] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with COTS HPC systems. In *International Conference on Machine Learning*, pages 1337–1345, 2013.

[39] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011.

[40] M. Dosanjh, S. M. Ghazimirsaeed, R. E. Grant, W. Schonbein, M. J. Levenhagen, P. G. Bridges, and A. Afsahi. The case for semi-permanent cache occupancy: Understanding the impact of data locality on network processing. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 73–84. ACM, 2018.

[41] M. Dosanjh, W. Schonbein, R. E. Grant, S. M. Ghazimirsaeed, and A. Afsahi. Fuzzy matching: Hardware accelerated mpi communication middleware. 19th Annual IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing (CCGrid), 2019.

[42] Paul Erdos and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[43] A. Faraj and X. Yuan. Automatic generation and tuning of MPI collective communication routines. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 393–402. ACM, 2005.

[44] I. Faraji and A. Afsahi. Design considerations for gpu-aware collective communications in MPI. *Concurrency and Computation: Practice and Experience*, 30(17):e4667, 2018.

[45] P Fischer, J Kruse, J Mullen, H Tufo, J Lottes, and S Kerkemeier. Nek5000: Open source spectral element CFD solver. *Argonne National Laboratory, Mathematics and*

Computer Science Division, Argonne, IL, see https://nek5000. mcs. anl. gov/index. php/MainPage, 2008.

[46] M. Flajslik, J. Dinan, and K. D. Underwood. Mitigating MPI message matching misery. International Conference on High Performance Computing, pages 281–299. Springer, 2016.

[47] E. W. Forgy. Cluster analysis of multivariate data: Efficiency vs. interpretability of classifications. *Biometrics*, 21:768–769, 1965.

[48] S. M. Ghazimirsaeed and A. Afsahi. Accelerating MPI message matching by a data clustering strategy. High Performance Computing Symposium (HPCS), in Lecture Notes in Computer Science (LNCS), 2017.

[49] S. M. Ghazimirsaeed, R. E. Grant, and A. Afsahi. Dynamic, unified design for dedicated message matching engines for collective and point-to-point communications. In *Elsevier International Journal of Parallel Computing (PARCO) (Invited paper - Under revision)*, 15 pages, 2019.

[50] S. M. Ghazimirsaeed, R. E. Grant, and A. Afsahi. A dedicated message matching mechanism for collective communications. In *12th International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, pages 26–36, 2018.

[51] S. M. Ghazimirsaeed, S. Mirsadeghi, and A. Afsahi. An efficient collaborative communication mechanism for MPI neighborhood collectives. In *Parallel & Distributed Processing, 2019. IPDPS 2019. IEEE International Symposium on*. IEEE, 12 pages, 2019.

[52] S. M. Ghazimirsaeed, S. Mirsadeghi, and A. Afsahi. Communication-aware message matching in MPI. Concurrency and Computation: Practice and Experience (CCPE)

journal, Presented in the 5th Workshop on Exascale MPI (ExaMPI 2017), 17 pages, 2017.

[53] R. L. Graham, R. Brightwell, B. Barrett, G. Bosilca, and J. Pješivac-Grbović. An evaluation of open mpi's matching transport layer on the cray xt. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 161–169. Springer, 2007.

[54] R. L. Graham and G. Shipman. MPI support for multi-core architectures: Optimized shared memory collectives. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 130–140. Springer, 2008.

[55] William Gropp. MPICH2: A new start for MPI implementations. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 7–7. Springer, 2002.

[56] F. Gygi. Large-scale first-principles molecular dynamics: moving from terascale to petascale computing. In *Journal of Physics: Conference Series*, volume 46, pages 268–277. IOP Publishing, 2006.

[57] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, and T. Peterka. HACC: Simulating sky surveys on state-of-the-art supercomputing architectures. *New Astronomy*, 42:49–65, 2016.

[58] N. Hjelm, M. Dosanjh, R. E. Grant, T. Groves, P. Bridges, and D. Arnold. Improving MPI multi-threaded RMA communication performance. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 58–69. ACM, 2018.

[59] T. Hoefler, R. Rabenseifner, H. Ritzdorf, B. R. de Supinski, R. Thakur, and J. Träff. The scalable process topology interface of MPI 2.2. *Concurrency and Computation: Practice and Experience*, 23(4):293–310, 2011.

[60] T. Hoefler and T. Schneider. Optimization principles for collective neighborhood communications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 98–108. IEEE Computer Society Press, 2012.

[61] T. Hoefler and J. Traff. Sparse collective operations for MPI. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[62] J. Hoepman. Simple distributed weighted matchings. *arXiv preprint cs/0410047*, 2004.

[63] C. U. Ileri and O. Dagdeviren. Performance evaluation of distributed maximum weighted matching algorithms. In *DICTAP*, pages 103–108, 2016.

[64] G. Inozemtsev and A. Afsahi. Designing an offloaded nonblocking MPI_Allgather collective using core-direct. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 477–485. IEEE, 2012.

[65] J. Jeffers and J. Reinders. *Intel Xeon Phi coprocessor high performance programming*. Newnes, 2013.

[66] J. Jeffers, J. Reinders, and A. Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[67] W. Jia, C. Zhang, and J. Chen. An efficient parameterized algorithm for m-set packing. *Journal of Algorithms*, 50(1):106–117, 2004.

[68] K. Kandalla, A. Buluç, H. Subramoni, K. Tomko, J. Vienne, L. Oliker, and D. K. Panda. Can network-offload based non-blocking neighborhood MPI collectives improve communication overheads of irregular graph algorithms? In *Cluster Computing*

Workshops (CLUSTER WORKSHOPS), 2012 IEEE International Conference on, pages 222–230. IEEE, 2012.

[69] K. Kandalla, H. Subramoni, A. Vishnu, and D. K. Panda. Designing topology-aware collective communication algorithms for large scale infiniband clusters: Case studies with scatter and gather. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[70] R. Keller and R. L. Graham. Characteristics of the unexpected message queue of MPI applications. In *Proceedings of the 17th European MPI users' group meeting conference on Recent advances in the message passing interface EuroMPI*, pages 179–188. Springer, 2010.

[71] B. Klenk, H. Fröening, H. Eberle, and L. Dennison. Relaxations for high-performance message passing on massively parallel SIMT processors. In *2017 IEEE International on Parallel and Distributed Processing Symposium (IPDPS)*, pages 855–865. IEEE, 2017.

[72] C. Koufogiannakis and N. E. Young. Distributed algorithms for covering, packing and maximum weighted matching. *Distributed Computing*, 24(1):45–63, 2011.

[73] R. Kumar, A. R. Mamidala, M. J. Koop, G. Santhanaraman, and D. K. Panda. Lock-free asynchronous rendezvous design for MPI point-to-point communication. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 185–193. Springer, 2008.

[74] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, and J. Ratterman. The deep computing messaging framework: generalized scalable message passing on the Blue Gene/P supercomputer. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 94–103. ACM, 2008.

[75] S. Kumar, P. Heidelberger, D. Chen, and M. Hines. Optimization of applications with non-blocking neighborhood collectives via multisends on the Blue Gene/P supercomputer. In *IPDPS International Parallel and Distributed Processing Symposium. IPDPS (Conference)*, pages 1–11, 2010.

[76] S. Kumar, A. Mamidala, P. Heidelberger, D. Chen, and D. Faraj. Optimization of MPI collective operations on the IBM Blue Gene/Q supercomputer. *The International Journal of High Performance Computing Applications*, 28(4):450–464, 2014.

[77] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing: design and analysis of algorithms*. Benjamin/Cummings Redwood City, 1994.

[78] X. Lapillonne, O. Fuhrer, P. Spörri, C. Osuna, A. Walser, A. Arteaga, T. Gysi, S. Rüdisühli, K. Osterried, and T. Schulthess. Operational numerical weather prediction on a GPU-accelerated cluster supercomputer. In *The European Geosciences Union (EGU) General Assembly Conference Abstracts*, volume 18, page 13554, 2016.

[79] S. Levy and K. B. Ferreira. Using simulation to examine the effect of MPI message matching costs on application performance. In *Proceedings of the 25th European MPI Users' Group Meeting*, pages 16–27. ACM, 2018.

[80] S. Li, T. Hoefler, and M. Snir. NUMA-aware shared-memory collective communication for MPI. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 85–96. ACM, 2013.

[81] A. Lingas and C. Di. Near approximation of maximum weight matching through efficient weight reduction. In *International Conference on Theory and Applications of Models of Computation*, pages 48–57. Springer, 2011.

[82] S. Lloyd. Least squares quantization in PCM. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[83] C. Lomont. Introduction to Intel advanced vector extensions. *Intel White Paper*, pages 1–21, 2011.

[84] Z. Lotker, B. Patt-Shamir, and S. Pettie. Improved distributed approximate matching. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 129–136. ACM, 2008.

[85] Z. Lotker, B. Patt-Shamir, and S. Pettie. Improved distributed approximate matching. *Journal of the ACM (JACM)*, 62(5):38, 2015.

[86] F. D. Lübbe. Micro-benchmarking MPI neighborhood collective operations. In *European Conference on Parallel Processing*, pages 65–78. Springer, 2017.

[87] T. Ma, T. Herault, G. Bosilca, and J. J. Dongarra. Process distance-aware adaptive MPI collective communications. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 196–204. IEEE, 2011.

[88] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.

[89] K. McGrattan, S. Hostikka, R. McDermott, J. Floyd, C. Weinschenk, and K. Overholt. Fire dynamics simulator, user's guide. *the National Institute of Standards and Technology (NIST) special publication*, 1019:6th Edition, 2013.

[90] S. H. Mirsadeghi and A. Afsahi. Topology-aware rank reordering for MPI collectives. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1759–1768. IEEE, 2016.

[91] S. H. Mirsadeghi, J. L. Träff, P. Balaji, and A. Afsahi. Exploiting common neighborhoods to optimize MPI neighborhood collectives. In *High Performance Computing (HiPC), 2017 IEEE 24th International Conference on*, pages 348–357. IEEE, 2017.

[92] C. Obrecht, F. Kuznik, B. Tourancheau, and J. Roux. Scalable lattice boltzmann solvers for cuda gpu clusters. *Parallel Computing*, 39(6-7):259–270, 2013.

[93] A. Ovcharenko, D. Ibanez, F. Delalondre, O. Sahni, K. E. Jansen, C. D. Carothers, and M. S. Shephard. Neighborhood communication paradigm to increase scalability in large-scale dynamic scientific applications. *Parallel Computing*, 38(3):140–156, 2012.

[94] O. Parekh. Iterative packing for demand and hypergraph matching. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 349–361. Springer, 2011.

[95] Simone Pellegrini, Torsten Hoefler, and Thomas Fahringer. On the effects of CPU caches on MPI point-to-point communications. In *2012 IEEE International Conference on Cluster Computing*, pages 495–503. IEEE, 2012.

[96] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance analysis of MPI collective operations. *Cluster Computing*, 10(2):127–143, 2007.

[97] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics*, 117(1):1–19, 1995.

[98] S. Plimpton, R. Pollock, and M. Stevens. Particle-mesh ewald and rRESPA for parallel molecular dynamics simulations. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing (PPSC)*, 1997.

[99] M. J. Rashti, R. E. Grant, A. Afsahi, and P. Balaji. iWARP redefined: Scalable connectionless communication over high-speed ethernet. In *High Performance Computing (HiPC), 2010 International Conference on*, pages 1–10. IEEE, 2010.

[100] J. Reinders. AVX-512 instructions. *Intel Corporation*, 2013.

[101] W. Schonbein, M. Dosanjh, R. E. Grant, and P. G. Bridges. Measuring multithreaded message matching misery. In *European Conference on Parallel Processing*, pages 480–491. Springer, 2018.

[102] G. Seber. *Multivariate observations*, volume 252. John Wiley & Sons, 2009.

[103] H. Shan, J. P. Singh, L. Oliker, and R. Biswas. Message passing and shared address space parallelism on an SMP cluster. *Parallel computing*, 29(2):167–186, 2003.

[104] J. Suykens and J. Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[105] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[106] T. Toffoli and N. Margolus. *Cellular automata machines: a new environment for modeling*. MIT press, 1987.

[107] J. Traff. Implementing the MPI process topology mechanism. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 28–28. IEEE, 2002.

[108] J. Träff, A. Carpen-Amarie, S. Hunold, and A. Rougier. Message-combining algorithms for isomorphic, sparse collective communication. *arXiv preprint arXiv:1606.07676*, 2016.

[109] J. Träff, F. D. Lübbe, A. Rougier, and S. Hunold. Isomorphic, sparse MPI-like collective communication operations for parallel stencil computations. In *Proceedings of the 22nd European MPI Users' Group Meeting*, pages 10–20. ACM, 2015.

[110] K. D. Underwood and R. Brightwell. The impact of MPI queue usage on message latency. In *IEEE International Conference on Parallel Processing*, pages 152–160, 2004.

[111] K. D. Underwood, K. S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell. A hardware acceleration unit for MPI queue processing. In *Proceedings of 19th IEEE International Conference on Parallel and Distributed Processing Symposium, 2005.*, page 96b. IEEE, 2005.

[112] M. Wattenhofer and R. Wattenhofer. Distributed weighted matching. In *International Symposium on Distributed Computing*, pages 335–348. Springer, 2004.

[113] T. S. Woodall, R. L. Graham, R. H. Castain, D. J. Daniel, M. W. Sukalski, G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, and J. J. Dongarra. TEG: A high-performance, scalable, multi-network point-to-point communications methodology. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 303–310. Springer, 2004.

[114] U. M. Yang and V. E. Henson. BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(1):155–177, 2002.

[115] D. Zhou, J. Huang, and B. Schölkopf. Learning with hypergraphs: Clustering, classification, and embedding. In *Advances in neural information processing systems*, pages 1601–1608, 2007.

[116] J. A. Zounmevo and A. Afsahi. A fast and resource-conscious MPI message queue mechanism for large-scale jobs. *Future Generation Computer Systems*, 30:265–290, 2014.

[117] J. A. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi. Using MPI in high-performance computing services. In *Proceedings of the 20th European MPI Users' Group Meeting*, pages 43–48. ACM, 2013.

1 Appendix: K-means Clustering

Clustering is the process of organizing objects into groups in a way that objects in the same group are similar to each other, and those from different groups are dissimilar. There are various clustering algorithms for different use cases. Among them, K-means clustering [102] is one of the most widely used clustering algorithm in literature. In the following, we briefly discuss how the K-means clustering works. Consider a given data set $\{x_1, , x_{\mathrm{m}}\}$, $x_{\mathrm{i}} \in R^{\mathrm{d}}$ for $i = 1, \cdots, m$ that represents a feature d-dimensional space. Suppose we want to divide these data points in k clusters. To do so, we need to determine two sets of variables: a cluster center for each cluster, μ_{j}, $j = 1, \cdots, k$, and indicator variables (cluster membership), ρ_{lj}, which are defined as follows:

$$\rho_{\mathrm{lj}} = 1 \text{ if the data point } x_{\mathrm{l}} \text{ belongs to cluster } j$$

$$\rho_{\mathrm{lj}} = 0 \text{ if the data point } x_{\mathrm{l}} \text{ does not belong to cluster } j$$

where $l = 1, ..., m$ and $j = 1, ..., k$

The variables μ_{j} and ρ_{lj} can be determined by solving the optimization problem 1 to minimize the total distance between the data points and their cluster centers,

$$min \sum_{l=1}^{m} \sum_{j=1}^{k} \rho_{\mathrm{lj}} ||x_{\mathrm{l}} - \mu_{\mathrm{j}}||^2 \tag{1}$$

where the optimization variables are $\rho_{\mathrm{lj}} \in \{0, 1\}$ and μj. Furthermore, $||x_{\mathrm{l}} - \mu_{\mathrm{j}}||^2 = \sum_{t=1}^{d} (x_{\mathrm{lt}} - \mu_{\mathrm{jt}})^2$ is the Euclidean distance between the data points and the clustering points. Note that it is possible to use other measures of similarity instead of the Euclidean distance.

There are various algorithms to solve this optimization problem [47, 82, 88, 102]. It should also be mentioned that the number of clusters, k, should be given as an input to the

algorithm for solving the above optimization problem.