Scalable Communication and I/O Algorithms for High Performance Computing Systems

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Engineering

By

Qiao Kang

EVANSTON, ILLINOIS

December 2020
ABSTRACT

The era of big data creates opportunities for carrying out scientific simulations at exascale. With increasing data size, the complexity of the design and execution of scientific applications demand the use of high-level tools, namely workflow systems on supercomputers. The performance of workflow systems has paramount importance since the goal of running scientific applications on a supercomputing system is to improve the end-to-end performance. The first part of this dissertation focuses on developing algorithms for inter-group All-to-All broadcast communication, which is an important data exchange pattern between workflow components. The second part of this thesis focuses on improving the two-phase I/O algorithm, which allows faster parallel file I/O operations by workflow components. The rest of this thesis contains miscellaneous works that apply machine learning approaches for improving applications’ prediction accuracy and execution performance.
Acknowledgements

I would like to thank Dr. Alok Choudhary, Dr. Wei-keng Liao, and Dr. Ankit Agrawal for advising me during the program. They taught me how to become a researcher. I would like to thank Northwestern University and all other funding sources for supporting my entire Ph.D. study. I would like to thank my family members for their continuous supports in the past five years.
Table of Contents

ABSTRACT 3

Acknowledgements 4

Table of Contents 5

List of Tables 11

List of Figures 14

Chapter 1. Introduction 25

Chapter 2. Scalable Algorithms for MPI Intergroup Allgather and Allgatherv 30

2.1. Background 33

2.1.1. Problem Definition 34

2.1.2. Optimal Transfer Time 35

2.2. Related Work 36

2.2.1. The Root Gathering Algorithm 37

2.3. The Allgather Algorithm 39

2.3.1. The Allgatherv Algorithm 46

2.4. Experimental Results 55

2.5. Conclusion 61
### Chapter 3. Optimal Algorithms for Half-Duplex Inter-Group All-to-All Broadcast on Fully Connected and Ring Topologies

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1. Background and Related work</td>
<td>65</td>
</tr>
<tr>
<td>3.1.1. Collective Communication</td>
<td>66</td>
</tr>
<tr>
<td>3.2. Algorithms for Fully Connected Topology</td>
<td>68</td>
</tr>
<tr>
<td>3.2.1. Comparison with Root Gathering Algorithm</td>
<td>72</td>
</tr>
<tr>
<td>3.3. Algorithms for Ring Topology</td>
<td>73</td>
</tr>
<tr>
<td>3.3.1. Message availability</td>
<td>76</td>
</tr>
<tr>
<td>3.3.2. Bidirectional communication constraint</td>
<td>78</td>
</tr>
<tr>
<td>3.3.3. Completeness</td>
<td>79</td>
</tr>
<tr>
<td>3.3.4. Optimality</td>
<td>81</td>
</tr>
<tr>
<td>3.3.5. Imbalanced Graph</td>
<td>81</td>
</tr>
<tr>
<td>3.4. Experimental Results</td>
<td>88</td>
</tr>
<tr>
<td>3.4.1. Fully Connected Topology</td>
<td>90</td>
</tr>
<tr>
<td>3.4.2. Ring Topology</td>
<td>91</td>
</tr>
<tr>
<td>3.5. Conclusions</td>
<td>92</td>
</tr>
</tbody>
</table>

### Chapter 4. Improving MPI Collective I/O for High Volume Non-contiguous Requests With Intra-node Aggregation

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1. Background</td>
<td>97</td>
</tr>
<tr>
<td>4.2. Design</td>
<td>99</td>
</tr>
<tr>
<td>4.2.1. Intra-node Aggregation</td>
<td>101</td>
</tr>
<tr>
<td>4.2.2. Inter-node Aggregation</td>
<td>106</td>
</tr>
<tr>
<td>4.2.3. I/O Phase</td>
<td>108</td>
</tr>
</tbody>
</table>
Chapter 6. Spatiotemporal Real-Time Anomaly Detection for Supercomputing Systems

6.1. Background and Related Work 162

6.2. Design 166

6.2.1. Identifying Independent Failures 167

6.2.2. Feature Construction 168

6.2.3. Realtime Anomaly Detection 171

6.3. Experimental Results 174

6.3.1. General Failure Detection 176

6.3.2. Critical Failure Detection 180

6.4. Summary 183

Chapter 7. Predicting Resource Requirement in Intermediate Palomar Transient Factory Workflow

7.1. Background 186

7.2. Design 188

7.2.1. Removing I/O anomalies 190

7.2.2. Prediction Target 191

7.2.3. Feature selection 193

7.2.4. Prediction model 200

7.3. Experimental Results 201

7.3.1. Baselines 202

7.3.2. Predicting stage 36 processing time 203

7.3.3. Predicting the number of objects extracted 206
7.4. Conclusion

Chapter 8. A Filtering-based Clustering Algorithm for Improving Spatiotemporal Kriging Interpolation Accuracy

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Kriging Algorithm</strong></td>
<td>211</td>
</tr>
<tr>
<td>8.1.</td>
<td>Spatio-temporal Kriging</td>
<td>212</td>
</tr>
<tr>
<td>8.1.2.</td>
<td>Clustering-based Kriging</td>
<td>213</td>
</tr>
<tr>
<td>8.2.</td>
<td>Design</td>
<td>214</td>
</tr>
<tr>
<td>8.2.1.</td>
<td>Filtering Phase</td>
<td>216</td>
</tr>
<tr>
<td>8.2.2.</td>
<td>Reinforcement Phase</td>
<td>217</td>
</tr>
<tr>
<td>8.2.3.</td>
<td>Convergence Analysis</td>
<td>218</td>
</tr>
<tr>
<td>8.2.4.</td>
<td>Complexity Analysis</td>
<td>220</td>
</tr>
<tr>
<td>8.3.</td>
<td>Implementation</td>
<td>221</td>
</tr>
<tr>
<td>8.3.1.</td>
<td>Caching for Kriging Interpolation</td>
<td>221</td>
</tr>
<tr>
<td>8.3.2.</td>
<td>Heuristics for Reinforcement</td>
<td>222</td>
</tr>
<tr>
<td>8.4.</td>
<td>Experimental Results</td>
<td>223</td>
</tr>
<tr>
<td>8.4.1.</td>
<td>Quality Measurements</td>
<td>223</td>
</tr>
<tr>
<td>8.4.2.</td>
<td>SOCR Data</td>
<td>224</td>
</tr>
<tr>
<td>8.4.3.</td>
<td>IGRA Data</td>
<td>225</td>
</tr>
<tr>
<td>8.5.</td>
<td>Summary and Future Work</td>
<td>228</td>
</tr>
</tbody>
</table>

Chapter 9. A Hybrid Training Algorithm for Recurrent Neural Network Using Particle Swarm Optimization-based Preprocessing and Temporal Error Aggregation

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Particle Swarm Optimization-based Preprocessing and Temporal Error Aggregation</strong></td>
<td>229</td>
</tr>
</tbody>
</table>
9.1. Background and Related Works

9.1.1. Back Propagation

9.1.2. Particle Swarm Optimization

9.2. Design

9.2.1. Exploration Phase

9.2.2. Exploitation Phase

9.2.3. Complexity Analysis

9.3. Experimental Results

9.3.1. Enhanced Recurrent Back Propagation

9.3.2. Hybrid Algorithm

Chapter 10. Conclusion and Future Work

References
List of Tables

3.1 Comparison of theoretical communication cost for the proposed algorithm and the root gathering algorithm when the number of senders is $p$ and the number of receivers is $q$ on a fully connected network. 73

3.2 Comparison of communication cost for the proposed algorithm and intra-group Allgather algorithm when the number of senders is $p$ and the number of receivers is $q$ on ring network. 81

4.1 This table summarizes the terms used in Chapter 4. 100

4.2 Datasets used in our evaluation in Chapter 4. For E3SM F and G benchmarks, the non-contiguous requests are collected from production runs using 21600 and 9600 MPI processes, respectively. As we present the strong-scaling results, the non-contiguous requests are partitioned among all $p$ processes used in our experiments. BTIO and S3D-IO have more significant numbers of non-contiguous requests, and the numbers increase as the number of processes $p$. For S3D-IO, $x$, $y$, and $z$ are the number of processes used to partition $X$, $Y$, and $Z$ dimensions. $x \cdot y \cdot z$ is the total number of processes. 109
5.1 Datasets used in Chapter 5. The second column shows the total number of noncontiguous requests. For the E3SM F benchmark, the noncontiguous requests are collected from production runs using 21,600 processes. We present the strong-scaling results, so the noncontiguous requests partitioned among all $p$ processes are used in our experiments.

BTIO has more significant numbers of noncontiguous requests, and the number increases as the number of processes $p$.

6.1 Number of events used for training and testing. Events for training part have timestamps from 2013 to 2018. Events for testing part have timestamps in 2018.

6.2 Case study for the Mira dataset using proposed methods. A prediction model $\hat{f}$ is trained with data from 2013 to 2018 by using features extracted by Algorithm 12. We apply Algorithm 13 with the trained model to predict failures in 2018.

6.3 Case study for the Mira dataset using period-based approach proposed in [1] with training data from 2013-2017. We apply Algorithm 13 with the trained model to predict failures in 2018.

6.4 Case study for the Mira dataset using proposed methods for coolant and power related events. A prediction model $\hat{f}$ is trained with data from 2013 to 2018 by using features extracted by Algorithm 12. We apply Algorithm 13 with the trained model to predict failures in 2018.
7.1 This table shows the earliest checkpoint that an image attribute is available.

7.2 Accuracy results of iPTF prediction models for predicting checkpoint 36 timing.

7.3 Accuracy results of iPTF prediction models for predicting the number of objects extracted.

8.1 SOCR Dataset Results

9.1 Chapter 9 datasets characteristics.

9.2 The improvements from new error definition for rBP gradient descent.

9.3 The improvements from new error definition for ADAM gradient descent.

9.4 MSE percentage reductions of hybrid algorithm (including all phases) for rBP and ADAM gradient descent.
List of Figures

2.1 Example execution of Algorithm 1 for $p = 6$ and $q = 2$. 40

2.2 Example execution Algorithm 1 for $p = 8$ and $q = 3$. 53

2.3 An illustration for the intergroup message exchange in Algorithm 2 for $p = 6$ and $q = 2$. (a) The send operations in Lines 12-14. (b) The send operations in line 22-24. 54

2.4 Allgather results for settings 1-4. Legend “Algorithm 1” refers to Algorithm 1 of this chapter, legend “EuroMPI” to Algorithm 2 in [2], and legend “Root Gathering” to the MPI library native root gathering algorithm. The plots are labeled with titles of the form “Number of processes (setting number)”. Settings for $p$, $q$, $k_A$, and $k_B$ have been defined in the text. The x-axis gives the maximum message size per node in Bytes, and the y-axis gives the completion time in milliseconds. The data type used is MPI_INT. 58

2.5 Algorithm 1 versus the MPI library native root gathering algorithm for settings 5-8. The plots are labeled with titles of the form “Number of processes (setting number)”. Settings for $p$, $q$, $k_A$, and $k_B$ have been defined in the text. The x-axis gives the maximum message size per
node in Bytes, and the y-axis gives the completion time in milliseconds.

The data type used is $\text{MPI\_INT}$. 59

2.6 Improvement of the proposed algorithms (Algorithm 1 for Allgather and Algorithm 2 for Allgatherv) ($T_{\text{new}}$) compared with the root gathering algorithm adopted by the MPI library ($T_{\text{old}}$). Improvements are computed as $\frac{T_{\text{old}}}{T_{\text{new}}}$. Each curve represents the improvement for one of the settings with different message sizes. The number in titles refers to the total number of processes. 60

3.1 Illustration of proposed Allgather algorithms for $p \geq q$ and $p < q$. The labels on arrows indicate message transferred. (a) First step for $p \geq q$. (b) Second step for $p \geq q$. (c) First step for $p < q$. (d) Second step for $p < q$. 69

3.2 An example (left to right and top to bottom) of the proposed algorithm for the ring topology all-to-all broadcast when $p = 4$ and $q = 6$. Message available at every step is labeled above/below a process. Arrows illustrate messages transferred. The total number of steps is 7. 82

3.3 An example (left to right and top to bottom) of the proposed algorithm for the ring topology all-to-all broadcast when $p = 6$ and $q = 4$. Message available at every step is labeled above/below a process. Arrows illustrate messages transferred. The total number of steps is 8. 83

3.4 These figures illustrate the comparison between the proposed algorithm and the benchmark algorithm in terms of communication cost. The
title format is "no. senders/no. receivers, topology". The unit of horizontal axis is number of bytes per message. The unit of vertical axis is the communication cost in seconds.

4.1 Examples of placement for local and global aggregators on distributed compute nodes. (a) The placement of 12 local and three global aggregators on three compute nodes while 8 MPI processes are running on each node. This configuration illustrates the case when the number of nodes equals the number of global aggregators. (b) The placement of 16 local and two global aggregators on six nodes in the case the number of nodes is more than the global aggregators. Blue circles represent global aggregators. Green circles represent local aggregators. A global aggregator can also serve as a local aggregator. The placement policy for local and global aggregators is to spread them out evenly among the available resource to prevent possible communication contention.

4.2 Illustrations of communication pattern in a collective write operation for two-phase I/O in (a) and TAM in (b). Each square box represents a compute node, and circles are MPI processes. The six green circles are local aggregators, and the three blue ones are global aggregators. Inter-node and intra-node communications are blue and green arrows, respectively. A one-to-one mapping connects the global aggregators and the file servers. In (a), there is no local aggregation phase. The all-to-many (24-to-3) communication pattern shows a potential contention at the global aggregators. In (b), the communication clearly
shows a reduced contention (6-to-3) at the global aggregators with the help of local aggregation phase.  

4.3 The I/O bandwidth comparisons of TAM and two-phase I/O in the strong-scaling evaluation from 256 to 16K processes on KNL and Haswell nodes. Haswell nodes have 32 physical cores, and KNL nodes have 64 physical cores. We set the number of processes per node equal to the number of available physical cores. Setting $p_l = p$ automatically degrades TAM to two-phase I/O. For our TAM in these comparisons, we set $p_l = 512$ (or $p_l = p$ if $P < 512$) based on empirical results. For I/O patterns with a large number of non-contiguous I/O requests, this strategy works well. We explain why such $p_l$ yields better performance later.  

4.4 The degree of contention for all datasets is denoted as the average number of receive operations for metadata exchange at global aggregators. TAM has 512 local aggregators in the comparisons.  

4.5 Timing breakdown for E3SM F case with different number of local aggregators. The right-most bar in every subfigure is the two-phase I/O timing ($p = p_g$). (a)-(d): KNL write. (e)-(h): Haswell write. (i)-(l): KNL read. (m)-(p): Haswell read.  

4.6 Timing breakdown for E3SM G case with different number of local aggregators. The right-most bar in every subfigure is the two-phase I/O timing ($p = p_g$). (a)-(d): KNL write. (e)-(h): Haswell write. (i)-(l): KNL read. (m)-(p): Haswell read.
4.7 Timing breakdown for BTIO with different number of local aggregators. The right-most bar in every subfigure is the two-phase I/O timing \((p = p_g)\). (a)-(d): KNL write. (e)-(h): Haswell write. (i)-(l): KNL read. (m)-(p): Haswell read.

4.8 Timing breakdown for S3D-IO with different number of local aggregators. The right-most bar in every subfigure is the two-phase I/O timing \((p = p_g)\). (a)-(d): KNL write. (e)-(h): Haswell write. (i)-(l): KNL read. (m)-(p): Haswell read.

5.1 This figure illustrates the communication workload of all processes in the spread-out and balanced spread-out algorithms in the first round. The red circles represent processes that are receivers. The blue circles represent the rest of the processes. An arrow between two circles represent a point-to-point communication. (a) The communication workload is imbalanced in the spread-out algorithm. Rank 3 receives data from 3–7 in the first round. (b) The communication workload is balanced in the balanced spread-out algorithm with virtual rank list \(\{0, 4, 1, 5, 2, 6, 3, 7\}\).

5.2 Two-phase I/O implementation for the Lustre and GPFS drivers. Different colors represent file domains owned by different I/O aggregators. We label the file domains handled by different rounds of two-phase I/O. (a) Lustre driver. (b) GPFS driver.
5.3 These figures illustrate the differences in the implementations we used for evaluation. We use the blue circles to denote nonaggregators. The red circles are I/O aggregators, and the green circles are local aggregators in TAM language. In this example, we have 2 I/O aggregators, 4 local aggregators, and 8 processes. We label their ranks next to them. The arrows indicate point-to-point communications.

(a) Default ROMIO implementation. All communication requests are posted simultaneously. (b) The balanced spread-out algorithm with throttling limit 4. The difference between the original and balanced spread-out algorithm has been illustrated in Figure 5.1. With throttling, an I/O aggregator receives data from a batch of 4 processes at a time. After receiving from a batch is finished, it receives from the next batch. This figure shows the first round of communication. (c) Pairwise algorithm. An I/O aggregator receives data from 1 process at a time. After receiving data from rank $x$, it receives data from $x + 1 \mod p$. This figure shows the first round of communication. (d) Two-layer aggregation method (TAM). An arbitrary process sends data to 1 local aggregator. Then all local aggregators exchange data with all I/O aggregators.

5.4 (a)–(d) illustrate the communication time of the E3SM-IO F case using the balanced spread-out algorithm with different throttling threshold. The figure titles identify the dataset, number of nodes, and computing system. The data communication pattern in E3SM-IO is all-to-many.
The x-axis depicts the upper bound of the number of concurrent MPI_Irecv in any MPI_Waitall call.

5.5 (a)–(l) are breakdown timings of different communication kernels on Cori and Summit. The figure titles indicate the dataset, number of nodes, and computing system. Cray-mpich and IBM Spectrum MPI libraries are not open-sourced, so we present their end-to-end time only. “ROMIO default” means that the current version of ROMIO is used. “Hybrid” means that we use the ROMIO default for metadata exchange and the balanced spread-out algorithm for data exchange.

(a) E3SM-IO F case on 64 Cori KNL nodes. (b) E3SM-IO F case on 256 Cori KNL nodes. (c) E3SM-IO F case on 128 Summit nodes. (d) E3SM-IO F case on 512 Summit nodes. (e) FLASH/IO on 64 Cori KNL nodes. (f) FLASH/IO on 256 Cori KNL nodes. (g) FLASH/IO on 128 Summit nodes. (h) FLASH/IO on 512 Summit nodes. (i) BTIO on 64 Cori KNL nodes. (j) BTIO on 256 Cori KNL nodes. (k) BTIO on 128 Summit nodes. (l) BTIO on 512 Summit nodes.

5.6 The partition of ROMIO Lustre collective write driver FLASH/IO data. There are 24 checkpoint variables, each with contiguous offsets represented in 3D blocks. The colors represent the file domains of I/O aggregators. An I/O aggregator receives metadata and data from ranks labeled with its color. Metadata exchange is in one round, and data exchange is in multiple rounds. Each round of data exchange spans three rows of file domains (blue, brown, and green) within a variable
in this example. Both metadata and data communication patterns are not all-to-many.

5.7 3D-layout of BTIO. The ROMIO Lustre collective write driver partitions BTIO data with three permutations (along the third dimension) represented by (a), (b), and (c). The colors represent the file domains of I/O aggregators. An I/O aggregator receives metadata and data from ranks labeled with its color. Metadata exchange is in one round, and data exchange is in multiple rounds. Each round of data exchange spans a file domain (blue, brown, and green) of three rows of the 2D array. The metadata communication pattern is all-to-many, but the data communication pattern is not.

6.1 This figure illustrates an example for feature construction as mentioned in Algorithm 13. We set $n_{\text{event}} = 4$. The first feature $x_1$ consists of the first 4 RAS events. The second feature $x_2$ consists of 5 RAS events. The condition at Line 7 is triggered for building $x_2$. After constructing $x_2$, Line 12 is triggered to jump the starting index of next RAS event to the beginning of $x_3$.

6.2 Illustration of convolutional neural network architecture used for training. The 2D convolutional layer has kernel size $2 \times 15$ and stride size $2 \times 1$. The pooling layer followed by the 2D convolutional layer has kernel size $1 \times 15$. The 1D convolutional layers have kernel size 3. The pooling layers followed by the 1D convolutional layers have kernel size 3.
size 2. Padding is applied to all convolutional+pooling layers. Each of
the fully connected layers has 2,048 hidden nodes.

6.3 (a) Log scale (based 10) number of events for 2013–2018. (b) Number
of RAS events with INFO severity across all five years. (c) Number of
RAS events with WARN severity across all five years. (d) Number of
RAS events with FATAL severity across all five years.

6.4 The frequency of all fatal events in logarithm (10) scale.

7.1 (a) Pearson correlation coefficients for pair-wise processing time
of stages before removing I/O anomalies. (b) Pearson correlation
coefficients for pair-wise processing time of stages after removing
I/O anomalies. (c) Spearman correlation coefficients for pair-wise
processing time of stages before removing I/O anomalies. (d) Spearman
correlation coefficients for pair-wise processing time of stages after
removing I/O anomalies.

7.2 Stage processing time of all stages after removing all I/O anomalies.

7.3 (a) Average stage 36 processing time in 24 hours period. (b) Average
objects extracted in 24 hours period. (c) Average stage 36 processing
time in spatial on right ascension (horizontal) and declination (vertical)
scale. (d) Average objects extracted in spatial on right ascension
(horizontal) and declination (vertical) scale. Note the processing
time roughly correlates with the position on the sky following the
overdensity of stars in our own galaxy.
7.4 iPTF prediction model feature summary. Type 1 features are the stage processing time of current images prior to checkpoint 36. Type 2 features are the attributes of current image. Category 2a is the number of objects extracted and category 2b contains other attributes of the images shown in Table 7.1. Type 3 features are the features provided by images in the spatiotemporal nearest neighborhood.

7.5 iPTF online prediction illustration. The circles indicate checkpoints of the workflow pipeline. The stages are denoted as arrows. The prediction of the number of objects is at checkpoint 1. The prediction of the stage 36 processing time is at checkpoint 36.

7.6 iPTF prediction model feature Bayesian network. Features that are directly connected to the red prediction targets are the one that should be used by the prediction models.

8.1 An illustration of filter algorithm. The left side contains clusters that are filtered and reinforced. The right side shows how a cluster is filtered and reinforced. The remaining points at the end of the process enter the next round.

8.2 (a) IGRA data variogram in four directions. The vertical axis shows the averaged covariance between points at a particular distance. Different lines show the variograms in different directions. (b) January IGRA data variogram. The vertical axis shows the averaged covariance between points at a particular distance. (c) Fitted model of the
variogram curve bounded by distance 30. (d) The horizontal axis represents time in hours. The vertical axis shows the $\chi^2$ IGRA Kriging error for Kriging with clustering and Kriging without clustering. The $\chi^2$ statistics of clustering-based Kriging is lower than Kriging without clustering across all time stamps.

9.1 An illustration of Elman neural recurrent network.

9.2 Unfolded Elman neural network.

9.3 MSE of testing data against epoch under rBP. (a) Space shuttle puppet withdraw (b) Space shuttle puppet pulled out. (c) QT ECG database record 0606. (d) MGH/MF database record 0003.

9.4 MSE of testing data against epoch under ADAM. (a) Space shuttle puppet withdraw (b) Space shuttle puppet pulled out. (c) QT ECG database record 0606. (d) MGH/MF database record 0003.
CHAPTER 1

Introduction

The era of big data creates opportunities for carrying out scientific simulations at exascale. With increasing data size, the complexity of the design and execution of scientific applications demand the use of high-level tools, namely workflow systems on supercomputers. Workflow components are systematic schedulings of operations for processing data. A workflow system is used to define, create, and execute workflows [3].

The performance of workflow systems has paramount importance since the goal of running scientific applications on a supercomputing system is to improve the end-to-end performance. Modern supercomputing system parallelizes computational tasks into different computing resources, such as computing cores and nodes. Later, the computing resources execute the workflows independently and merge their results via shared memory or interprocess communication frameworks. Therefore, the time cost for running an application can be divided into two categories. The first category is computational cost, which includes time cost for local computations and shared file I/O operations. The second category is communication cost, which is the time cost for transferring data among workflow components [4]. Improving collective communication and I/O algorithms can significantly benefit scientific workflow application performance. We present algorithms for both intergroup collective communications among workflow components and improvements of two-phase I/O strategy [5], which has become the implementation backbone for collective I/O in almost all MPI libraries.
The first part of this dissertation focuses on intergroup All-to-All broadcast communication, which in MPI corresponds to the `MPI_Allgather` and `MPI_Allgatherv` functions when using MPI inter-communicators. With an inter-communicator as an input argument, `MPI_Allgather` and `MPI_Allgatherv` exchange data between the two disjoint process groups, such that every process in one group receives messages from all processes in the other group. MPICH [6], MVAPICH [7], and OpenMPI [8], the three most widely-used MPI implementations in the parallel processing community, implement these intergroup collectives (as well as most other intergroup collectives) based on the “root gathering” algorithm paradigm. The underlying principle of the root gathering algorithm is single-process accumulation per group followed by a pair-wise exchange of the gathered messages followed by one-to-all broadcasts within the two groups. However, this approach does not achieve optimal communication time under the single-port communication constraint, because the complete data to be exchanged are sent multiple times. The paper [2] presents an algorithm for `MPI_Allgather` that outperforms the root gathering algorithm by a large factor and achieves optimal communication transfer time under a reasonable system (and message size) assumptions. The algorithm divides messages into segments in order to fully utilize the available communication links between the two groups (as well as within each of the groups), achieving optimal transfer time. An extension work further improves the formulation of the algorithm proposed for `MPI_Allgather` in [2] by combining the two-stage intragroup Allgather communication into a single stage with the same communication time [9]. In addition, the implementation strategy for handling the case of indivisible process numbers is described in details. Moreover, we extend the message segmentation concept to design an algorithm for intergroup `MPI_Allgatherv`. The
extension generalizes the solution for balancing the message size exchanged between two groups for intergroup MPI_Allgather. We show that the proposed algorithm for intergroup MPI_Allgatherv significantly improves the end-to-end execution time compared with the root gathering algorithm for large message sizes. Finally, the paper [10] present optimal algorithms for half-duplex inter-group all-to-all broadcast (Allgather) under bidirectional communication constraint for any number of senders and receivers on both fully connected and ring topologies. Chapter 2 and 3 are the full papers [9] and [10].

The second part of this thesis focuses on improving the two-phase I/O algorithm. In [11], we present an improvement for MPI collective I/O, denoted as the two-layer aggregation method (TAM), which adds an intra-node request aggregation layer so that the communication in the two-phase strategy consists of two layers of request aggregations. In the intra-node aggregation layer, MPI processes running on the same compute nodes perform a request aggregation to a subset of processes, denoted as local aggregators. In contrast to local aggregators, we denote the I/O aggregators in the original two-phase I/O as the global aggregators. As communication takes place among processes on the same node, the cost is expected to be relatively small. This intra-node aggregation performs on all compute nodes independently and concurrently. Once receiving the requests from local processes, the local aggregators coalesce them into fewer contiguous requests. After coalescing non-contiguous I/O requests, the local and global aggregators across compute nodes enter the traditional two-phase I/O to complete collective read or write operations. In Chapter 4, we refer to the communication between local and global aggregators as the inter-node aggregation. An advantage of TAM is the reduction of the number of inter-node communications at global aggregators since the number of local aggregators is much
less than the number of processes. In the traditional two-phase I/O, each global aggregator may receive requests from all other MPI processes, potentially causing communication contention at the global aggregators for large-scale applications with a significant number of non-contiguous I/O requests. The intra-node aggregation alleviates such a problem by breaking the all-to-many communication into two layers so that the global aggregators receive requests only from the local aggregators. Realizing the importance of communication cost in two-phase I/O, we explore all-to-many personalized communication algorithms: pairwise and spread-out for improving two-phase I/O communication kernel performance. Metadata and communication kernels of two-phase I/O in ROMIO are replaced with different all-to-many personalized communication algorithms. We extend the spread-out algorithm adopted by the implementations of personalized all-to-all in major MPI production libraries to adapt to the all-to-many communication pattern by reducing the straggler effect resulting from an unbalanced communication workload. Throttling techniques are additionally applied to the asynchronous MPI point-to-point implementation to reduce communication contention. Chapter 4 and 5 are the full papers [11] and [12].

In addition, we also target on improving the reliability of the system that workflow components are running. As the scale of supercomputer systems grows, system failures, which can negatively affect system performance [13], become critical. If knowledge about failures can be predicted via heuristics, one can deploy an autonomous system that can schedule resources and actions in order to maximize the overall system performance, as argued in [14]. For example, failure recovery measures such as checkpoint saving can be used to reduce the cost from system failures [15], which in turn reducing local
computational cost caused by repeated execution of the same job. We present a machine learning algorithm for detecting spatiotemporal system anomalies [16] in Chapter 6. To build up the necessary knowledge on machine learning and anomaly detection, this thesis includes several topics on using machine learning for solving application problems. We present a filtering-based clustering algorithm that improves Krigging accuracy [17] in Chapter 8. Chapter 9 contains an algorithm for training a recurrent neural network. A recurrent neural network can be used for semi-supervised anomaly detection [18]. Finally, Chapter 7 constructs features from history log data from Intermediate Palomar Transient Factory (iPTF) workflow and predicts future resource requirements of iPTF workflow on supercomputers. The prediction of resource requirement helps supercomputer schedulers for better allocation of resources, which in turn improves the overall throughput.
CHAPTER 2

Scalable Algorithms for MPI Intergroup Allgather and Allgatherv

MPI [19] intergroup collective communication defines message transfer patterns between two disjoint groups of MPI processes, with communication taking place between groups but not inside groups. In contrast to the intragroup collective communication counterparts, where all processes in an Allgather communication receive the same messages from all other processes, intergroup collectives can have expressive and performance advantages for coupled scientific applications and workflows systems.

Intergroup collective communication patterns occur frequently in modern parallel frameworks for scientific applications. As discussed in [2], a scientific application workflow system consists of different groups of processes, where the results computed by one process are sent to another via intergroup All-to-All broadcast (Allgather) and/or All-to-All personalized communication. In [20], the authors proposed a parallel data transfer framework for a computationally intensive weather prediction system, SCALE-LETKF [21]. Communication among workflow components is achieved using MPI intergroup Allgather. An advantage of using intergroup communication is achieving a higher degree of fault tolerance. Examples discussed in [22] include applications such as DNA sequencing, graphics rendering, and searching for extraterrestrial intelligence that relies on a manager/worker model can benefit from the encapsulation provided by inter-communicators. Researchers
have designed sophisticated approaches that reduce the data size communicated among workflow components and in turn improving communication performance. For example, Zhang et al. [23] have proposed a distributed framework for maximizing on-chip data exchange in order to reduce the amount of communication among workflow components. Docan et al. [24] have designed an Active Spacing strategy that reduces the size of message transfers among workflow components by moving programs to staging areas. Existing literature that focuses on reducing the communication message size and frequency has successfully improved the communication performance between workflow components. Nevertheless, intergroup communications with large data size seem unavoidable, and applications can benefit from optimal MPI implementations of the intergroup collective communication.

In this chapter, we focus on intergroup All-to-All broadcast communication, which in MPI corresponds to the `MPI_Allgather` and `MPI_Allgatherv` functions when using MPI inter-communicators. With an inter-communicator as an input argument, `MPI_Allgather` and `MPI_Allgatherv` exchange data between the two disjoint process groups, such that every process in one group receives messages from all processes in the other group. MPICH [6], MVAPICH [7], and OpenMPI [8], the three most widely-used MPI implementations in the parallel processing community, implement these intergroup collectives (as well as most other intergroup collectives) based on the “root gathering” algorithm paradigm. The underlying principle of the root gathering algorithm is single-process accumulation per group followed by a pair-wise exchange of the gathered messages followed by one-to-all broadcasts within the two groups. However, this approach does not achieve
optimal communication time under the single-port communication constraint, because the complete data to be exchanged are sent multiple times.

In our previous work [2], we presented an algorithm for MPI_Allgather that outperforms the root gathering algorithm by a large factor and achieves optimal communication transfer time under a reasonable system (and message size) assumptions. The algorithm divides messages into segments in order to fully utilize the available communication links between the two groups (as well as within each of the groups), achieving optimal transfer time.

In this chapter, we further improve the formulation of the algorithm proposed for MPI_Allgather in [2] by combining the two-stage intragroup Allgather communication into a single stage with the same communication time. In addition, the implementation strategy for handling the case of indivisible process numbers is described in detail. Moreover, we extend the message segmentation concept to design an algorithm for intergroup MPI_Allgatherv. The extension generalizes the solution for balancing the message size exchanged between two groups for intergroup MPI_Allgather. We show that the proposed algorithm for intergroup MPI_Allgatherv significantly improves the end-to-end execution time compared with the root gathering algorithm for larger message sizes.

We implement the proposed algorithms for intergroup MPI_Allgather and MPI_Allgatherv using MPI point-to-point and collective communication functions, namely intergroup send/receive, intragroup MPI_Allgather, and intragroup MPI_Allreduce. We conduct experiments on Cori, a Cray XC40 supercomputer at the National Energy Research Scientific Computing Center (NERSC). Direct comparisons with the MPI native
library function **MPI_Allgather** that implements the root gathering algorithm are presented. Although the communication network topology on Cori is pseudo-fully connected (dragon-fly), we show that our ideas are applicable to hierarchically organized processes by evaluating and comparing our algorithms against the MPI library installed on Cori with different process settings. With message sizes per compute node ranging from 64KBytes to 8MBytes per node, the experiments show a significant performance improvement achieved by our algorithms, which are up to 23.67 times on 256 compute nodes faster than the production MPI libraries.

The rest of this chapter is structured like the following. In Section 2.1 we define the communication model assumptions and problem notations. In Section 2.2 we discuss related work for MPI collective communications, including detailed descriptions for the root gathering algorithm with the formulation of its communication cost. In Section 2.3 we propose the algorithms for intergroup **MPI_Allgather** and **MPI_Allgatherv**. In Section 2.4 we present the experimental results. Finally, we summarize this chapter in Section 2.5.

### 2.1. Background

For design and analysis, we employ the single-port communication model as defined in [25]. A set of ranked processes \{0, 1, \ldots, p − 1\} can communicate as specified by a communication graph representing the communication network. Processes are nodes in the graph, and edges represent communication links, such that pair-wise processes connected by an edge can communicate directly. The basic communication operations are send and receive operations initiated by the processes. When a send or receive operation is
called by a process, the function is blocked until the communication has been completed. Therefore, a process cannot send to multiple processes simultaneously. Likewise, a process cannot receive from multiple processes at the same time.

Communication cost is determined by a startup time term $t_s$ and a per-byte transfer time term $t_w$. If there is a link between a sender $x$ and a receiver $y$, sending a message of size $k$ bytes from $x$ to $y$ has a communication transmission time of $t_s + kt_w$ time units. We refer to the $t_s$ term as the startup time and $kt_w$ as the transfer time. In this model, the objective of designing collective communication algorithms is to minimize communication transfer time. Modern networks are typically bidirectional, and a process can be involved in both a (non-blocking) receive and send operation at a time. Our algorithms exploit this when exchanging messages between processes in the two groups. Modern networks may be able to sustain more communication operations at a time. Our algorithms can also exploit such capabilities, but we leave this optimization as an implementation detail and use MPI non-blocking send and receive operations in our concrete implementations.

We assume that there are communication links between all processes in the two groups. This assumption is strong, and may not hold for all applications of the MPI intergroup collectives. For instance, coupled applications running on different parts of a heterogeneous system may be only weakly connected with only a few communication edges between the two process groups.

2.1.1. Problem Definition

Let $A = \{a_0, ..., a_{p-1}\}$ and $B = \{b_0, ..., b_{q-1}\}$ be two disjoint groups of processes. Group $A$ has size $p$ and group $B$ has size $q$. Initially, every $a_i \in A$ has a unique message $m_{A,i}$.
of size $k_A$ bytes and every $b_j \in B$ has a unique message $m_{B,j}$ of size $k_B$ bytes. We term $m_A = \{m_{A,i} | \forall 0 \leq i < p \}$ and $m_B = \{m_{B,i} | \forall 0 \leq i < q \}$ the full messages of the two groups. The goal of the All-to-All broadcast (Allgather) operation is to let every $b_j \forall 0 \leq j < q$ receive $m_A = m_{A,i} \forall 0 \leq i \leq p - 1$ and every $a_j \forall 0 \leq j < p$ receive $m_B = m_{B,i} \forall 0 \leq i \leq q - 1$, that is, to exchange the full messages between all processes in the two groups.

We refer to $k_A$ and $k_B$ as the block sizes, and $pk_A + qk_B$ as the (total) problem size. In the following, we will assume without loss of generality that $q \leq p$.

### 2.1.2. Optimal Transfer Time

Consider a single process $a \in A$, which has to receive all messages from the group $B$ in the end. Thus, process $a$ has to receive $qk_B$ bytes. By the single-port constraint, it takes at least $qk_B t_w$ time units for $a$ to receive all messages. Hence the lower bound on the transfer time is $qk_B t_w$. The same argument can be applied to each process $b \in B$, giving a lower bound on the message transfer time of $pk_A t_w$ time units. By the argument, it follows that

$$M = \max (qk_B, pk_A) t_w$$

is a lower bound on the transfer time for intergroup Allgather. We prove that this lower bound is the largest asymptotic lower bound by proposing an algorithm with transfer time converging to it given large $p$ and $q$. 
2.2. Related Work

In MPI, the collective communication operations come in two flavors: Intragroup and intergroup operations [19, Chapter 5]. MPI distinguishes between the two cases by the communicator argument, which can be either an intra-communicator or an inter-communicator. The actual interfaces are the same in the two cases, but the underlying communication patterns are different. Inter-communicators in MPI can be viewed as objects that contain two intra-communicators representing the two disjoint process groups. In our implementations, we can reuse the efficient MPI intragroup collectives for communication inside the two process groups.

Intragroup collective communication algorithms have been thoroughly studied in existing literature [26]. Johnson et al. [27] have implemented the Allgather function under a single-port communication constraint on hypercube topology. The algorithm of Bruck et al. [28] addresses the case of non-power of two number of processes for Allgather and All-to-All. Thakur et al. [29] optimized intragroup Allgather for MPI using recursive doubling and the Bruck algorithm [28] for non-power number of processes. Träff [30] has proposed an algorithm for intragroup Allgatherv using a ring topology that is efficient for larger data sets. Like the root gathering algorithm, our new algorithms benefit from efficient implementations of these algorithms for intragroup collectives, in particular, implementations that incorporate to the hierarchical structure of modern, parallel systems.

Intergroup collective communication, on the other hand, is still in need of further research. For intergroup all-to-all broadcast (MPI_Allgather), MPICH [6], MVAPICH [7] and OpenMPI [8], the most widely-used MPI implementations in the HPC community,
adopt the “root gathering algorithm” which does, by a significant factor, not achieve
optimal transfer time. Also closed source, vendor libraries like Intel MPI seem to use
some variant of the root gathering paradigm.

2.2.1. The Root Gathering Algorithm

The root gathering algorithm, adopted by popular MPI production libraries such as
MPICH, MVAPICH, and OpenMPI, is discussed in [31]. To the best of our knowledge,
this algorithm is the state-of-art implementation of MPI inter-group Allgather. For each
of the two groups, a root process gathers all messages from other processes in the local
group and broadcasts the gathered message to all processes in the remote group. The root
gathering algorithm can be readily implemented using existing MPI functionality without
creating any intermediate communicators. However, it does not meet the lower bound on
communication transfer time under the single-port communication constraint by a large
factor.

The best version of the root gathering algorithm has three stages. Let $a_0$ and $b_0$ be
chosen root processes in groups $A$ and $B$. In the first stage, root process $a_0$ gathers the full
message $m_A$ from the other $p-1$ processes in the group $A$. Root process $b_0$ gathers the full
message $m_B$ from the other $q-1$ processes in the group $B$. The overall communication
cost of this stage is

$$\max (\log (p) t_s + (p - 1) k_A t_w, \log (q) t_s + (q - 1) k_B t_w)$$
using standard, optimal, binomial-tree algorithms, see, e.g. [26]. In the second stage, the two roots exchange their aggregated messages, which takes

\[ \max (t_s + pk_A t_w, t_s + qk_B t_w) \]

(time units. In the third stage, root process \(a_0\) broadcasts \(m_B\) of size \(qk_B\) to the other \(p - 1\) processes in its group. Root process \(b_0\) broadcasts \(m_A\) of size \(pk_A\) to the other \(q - 1\) processes within its group. This stage takes time

\[ \max (\log (p) t_s + qk_B t_w, \log (q) t_s + pk_A t_w) \]

(plus lower order terms) using an optimal algorithm for the broadcast operations [32, 33].

Let \(T\) be total the communication cost of the three stages. A lower bound for the communication cost of the root gathering algorithm can be bounded as expressed by Equation (2.1), with \(\alpha = (1 + 2 \log (p))\).

\[
T \leq \alpha t_s + (\max ((p - 1) k_A, (q - 1) k_B)) t_w + 2M \\
= \alpha t_s + \max ((3p - 1) k_A, (3q - 1) k_B) t_w \\
< 2 \log (p) t_s + 3M \quad \text{as } p, q \to \infty
\]  

The time of this version of the root gathering algorithm is about a factor three larger than the lower bound.

MPI libraries like MPICH, MVAPICH, and OpenMPI do not implement optimal algorithms for broadcast, but algorithms that are a factor of two off in the transfer time
Furthermore, in their implementations, the intergroup exchange of gathered messages at Stage 2 and the one-to-all broadcasts at Stage 3 are all performed sequentially. Thus, the implementation of the root gathering algorithm in MPI production libraries has communication cost $T$ roughly as shown in Equation (2.2), where 

$$\beta = (2 + 2 \log (p) + \log (q))$$

This cost is a baseline for evaluating our new algorithm.

$$T \leq \beta t_s + \max ((p - 1) k_A,(q - 1) k_B) + 3 p k_A + 3 q k_B) t_w$$

\[
(2.2) \quad \leq 3 \log (p) t_s + M + 3 (p k_A + q k_B) t_w \quad \text{as } p, q \to \infty
\]

### 2.3. The Allgather Algorithm

<table>
<thead>
<tr>
<th>Algorithm 1: Full-duplex Intergroup All-to-All Broadcast with Message segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 for $i \in \left[0, \ldots, \frac{p}{q} - 1\right]$ do</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7 end</td>
</tr>
<tr>
<td>8 # Group $A$ Allgather received messages from group $B$.</td>
</tr>
<tr>
<td>9 Intra-Allgather $a_j, m_{B,j,\frac{p}{q}j} \forall {j : 0 \leq j &lt; p}$</td>
</tr>
<tr>
<td>10 # Group $B$ Allgather received messages from group $A$</td>
</tr>
<tr>
<td>11 Intra-Allgather $b_j, m_{A,qi+j} \forall (0 \leq j &lt; q, 0 \leq i &lt; \frac{p}{q})$</td>
</tr>
</tbody>
</table>

We present our new algorithm for the intergroup Allgather operation that achieves the transfer time lower bound asymptotically. In [2], we have presented an algorithm for intergroup Allgather with optimal transfer time for large numbers of processes. The
algorithm for intergroup Allgather proposed in this chapter simplifies the previous work by reducing the two-stage intragroup Allgather operations to one stage. We show that the transfer time of the proposed algorithm is also optimal for large numbers of processes.
Overall, the idea of the proposed algorithm is to exploit as many communication links between the two groups concurrently as possible in order to exchange the full messages $m_A$ and $m_B$ between the two groups. After that, concurrent intragroup Allgather operations in each of the groups collect the received messages to achieve the objective of intergroup Allgather. Our algorithm keeps $q$ pairs of processes active throughout the exchange (recall that we assumed that $q \leq p$), and the exchange is done over $\lceil \frac{p}{q} \rceil$ communication rounds.

We divide the messages from the smaller group $B$ into segments such that the total number of segments in the group $B$ is $p$. From that, the correctness of the algorithm is clear by construction: Each process in group $A$ will receive a segment from a process in the group $B$, and each process in the group $B$ will receive one or more blocks from processes in group $A$. All blocks from group $A$ will have been transferred to processes in group $B$, and all segments from the group $B$ will likewise have been transferred to processes group $A$. The concurrent Allgather operations in the two groups collect the blocks and the segments together, such that each process will have the full message from the processes of the other group. The partner processes in the exchange are chosen such that blocks and segments are received in rank order over the processes, and an Allgather operation without any block or segment reordering therefore suffices. The same idea can also be applied to the `MPI_Allgatherv` operation as described in Section 2.3.1.

Algorithm 1 formally shows the proposed intergroup Allgather operation. It consists of a message exchange step between the two groups of processes, followed by concurrent intragroup Allgather operations within each of the groups. We first assume that $q$ divides $p$. Each process $i$ in group $A$ sends its message $m_{A,i}$ to process $j = \lfloor i / q \rfloor$, as shown in Line 4. The messages in group $B$ are segmented and sent across the groups in $\frac{p}{q}$ communication
rounds, as shown in Line 5. For now, we also assume that $k_B$ is divisible by $\frac{p}{q}$. Each process $j$ in group $B$ sends a part of its message $m_{B,j}$ of size $k_B/\frac{p}{q}$ to process $j\frac{p}{q} + i$ for $i = 0, \ldots, \frac{p}{q} - 1$. After this exchange in $\frac{p}{q}$ communication rounds, it is obvious that all messages from group $A$ have arrived at processes in the group $B$, and that all segments of the messages in the group $B$ are likewise present at processes in the group $A$. To complete the All-to-All broadcast operation, each of the two groups performs intragroup Allgather operations over messages received from the remote group. These two Allgather operations can take place concurrently. The intragroup Allgather operations over the group $A$ for messages received from the group $B$ is in Line 9. The intragroup Allgather operations over the group $B$ for messages received from group $A$ is in Line 11. Figure 2.1 shows an execution of the algorithm with $p = 6$ and $q = 2$.

To handle the case where $\frac{p}{q}$ is not an integer, we divide processes in group $A$ into $q$ disjoint subgroups $\{s_0, \ldots, s_{q-1}\}$ such that all ranks in $s_i$ are smaller than ranks in $s_j$ if $i < j$. We assign the size of the subgroups in the following way. Let $r = p \mod q$. For $i < r$, $s_i$ has size $\lceil \frac{p}{q} \rceil$. Otherwise $s_i$ has size $\lfloor \frac{p}{q} \rfloor$. Since $p = \lceil \frac{p}{q} \rceil r + \lfloor \frac{p}{q} \rfloor (q - r)$, the union of the subgroups is $A$.

We modify Algorithm 1 in the following way. Every $b_i \in B$ exchanges messages with all processes in the subgroup $s_i$. Then, both group $A$ and group $B$ perform intragroup Allgather operations for messages received from their remote groups. Since every subgroup of the group $A$ has almost the same size, the segmented message received by the processes in group $A$ also has almost the same size. Likewise, every process in group $B$ also has a similar size of messages received from the group $A$. With the balanced send sizes, the final intragroup Allgather operations can be performed using the largest send size.
of any process without having significant completion time overhead, or by an Allgatherv operation. Figure 2.2 gives an example with \( p = 8 \) and \( q = 3 \). We have \( s_0 = \{0, 1, 2\} \), \( s_1 = \{3, 4, 5\} \), and \( s_2 = \{6, 7\} \). The final intragroup Allgather for group \( B \) has a send size of \( 3k_A \). The final intragroup Allgather for group \( B \) has a send size of \( \frac{k_B}{2} \).

If the message size \( k_B \) of the group \( B \) is not divisible by \( \frac{p}{q} \), some processes from \( B \) will send a smaller, potentially empty message to some processes in the group \( A \). The algorithm is still correct in the sense that all segmented messages from group \( B \) are present in the group \( A \) after the exchange. Again, the Allgather step is done either by using the larger send size or by an Allgatherv operation.

The segmentation for messages transferred from the group \( B \) to group \( A \) is necessary, especially when \( \frac{p}{q} \) is large. In \cite{2}, we analyzed the transfer time of an algorithm similar to Algorithm 1 but without message segmentation. We showed that the transfer time of this algorithm is not bounded by any multiple of the optimal transfer time lower bound when \( pk_A < qk_B \) as \( \frac{p}{q} \to \infty \).

The message segmentation algorithm proposed in \cite{2} differs from Algorithm 1 in the way that processes in the group \( A \) gather all messages from the group \( B \) via two steps of intragroup Allgather operations. With the homogeneous communication model cost assumed in this chapter, the two-step intergroup Allgather operation is unnecessary, since the one-step version in Algorithm 1 achieves the task with the same transfer time. However, the two-step implementation of \cite{2} could have advantages on hierarchical systems with non-homogeneous costs in case the MPI_Allgather operation is not properly aware of the processor hierarchy. For instance, one of the Allgather operations can operate entirely inside processor nodes. We prefer the formulation of Algorithm 1 and assume that
the MPI\textunderscore\texttt{Allgather} operation is properly implemented to be efficient over any assignment of MPI processes to processors in the compute nodes.

In Algorithm 1 Lines 1-7 initiate $\frac{p}{q}$ operations, so the startup time is $\frac{p}{q}t_s$. For Line 9, the Allgather can be finished in $\log(p)$ communication rounds using the Bruck algorithm \cite{28}. For Line 11, the number of communication rounds is $\log(q)$. Therefore, the overall startup time for Lines 9-11 is $\log(p)t_s$ since $q \leq p$ by assumption.

\textbf{Theorem 1.} \textit{The transfer time term of Algorithm 1 is bounded by}

\begin{equation}
(2.3) \quad (\max(pk_A, qk_B) + k_B) t_w = M + k_B t_w
\end{equation}

\textbf{Proof.} The total transfer time of Lines 1-7 is

$$\max\left(\frac{k_A p}{q}, \frac{k_B p}{q}\right) t_w = \max\left(\frac{k_A p}{q}, k_B\right) t_w.$$ 

Line 9 has a total transfer time of $\left(q - \frac{q}{p}\right) k_B t_w$, since it is an intragroup Allgather operation of $\frac{qk_B}{p}$ message size over $p$ processes. Line 11 has transfer time $\left(p - \frac{p}{q}\right) k_A t_w$. The intragroup Allgather at Line 11 runs parallel to Line 9. It follows that the overall transfer time can be bounded using Equation (2.4).
\[(2.4) \quad t_w \leq \left( \max \left( k_A \frac{p}{q}, k_B \right) + \max \left( k_A \left( p - \frac{p}{q} \right), k_B \left( q - \frac{q}{p} \right) \right) \right) t_w \]

Theorem 1 gives a lower bound for the transfer time of Algorithm 1. This lower bound justifies the optimality of transfer time for Algorithm 1 asymptotically with respect to \( p \) and \( q \).

We summarize the correctness, namely that the full messages \( m_A \) and \( m_B \) are gathered at all processes in groups \( B \) and \( A \) as required in Theorem 2.

**Theorem 2.** Algorithm 1 correctly implements the intergroup Allgather operation.

Theorem 2 follows from the description of Algorithm 1. In Figure 2.1, it is clear that messages transferred from group \( A \) to group \( B \) do not overlap and have a total size of \( pk_A \). Likewise, messages transferred from group \( B \) to group \( A \) are also disjoint and have a total size of \( qk_B \). Since all messages received from the other group are contributed to each of the Allgather operations, Algorithm 1 correctly implements the intergroup Allgather.

Algorithm 1 does not utilize all links possible between the \( p + q \) processes. We note that optimal transfer time requires much fewer links than the \( (p + q)(p + q - 1) \) links of
a fully connected system. The intragroup Allgather only requires a ring topology to reach optimal completion time \([26, 30]\). For Algorithm 1 Lines 1-7 require \(p\) links since every process in the group \(B\) sends messages to \(\frac{p}{q}\) processes in the group \(A\) and every process in the group \(A\) sends messages to only one process in the group \(B\). Line 9 requires \(p\) processes connected in a ring to perform the Allgather. Likewise, Line 11 requires \(q\) processes connected in a ring to perform the Allgather. Therefore, the minimum number of links to maintain optimal transfer time is less than \(p + q + \frac{p}{q}\) and linear in the number of processes.

2.3.1. The Allgatherv Algorithm

The MPI\_Allgatherv operation generalizes the MPI\_Allgather collective. Instead of every process in each group having the same block size, MPI\_Allgatherv allows every process to contribute a block of different size. Therefore, a process receives messages of different size from every process in its remote group.

Let \(k_{A,i}\) be the message size of \(m_{A,i}\) sent by process \(a_i\) and \(k_{B,j}\) be the message size of \(m_{B,j}\) sent by process \(b_j\). The goal of intergroup MPI\_Allgatherv is to let every \(b_j\) \(\forall 0 \leq j < q\) receive \(m_A = m_{A,i}\) \(\forall 0 \leq i \leq p - 1\) and every \(a_j\) \(\forall 0 \leq j < p\) receive \(m_B = m_{B,i}\) \(\forall 0 \leq i \leq q - 1\). By the same argument as for Allgather in Section 2.1.2 it follows that a lower bound for the optimal transfer time is

\[
M = \max \left( \sum_{i=0}^{p-1} k_{A,i}, \sum_{i=0}^{q-1} k_{B,i} \right) t_w.
\]

The algorithm for the MPI\_Allgatherv operation follows the same principle as for MPI\_Allgather. The processes in each group compute the total message sizes \(K_A =\)
\[ \sum_{i=0}^{p-1} k_{A,i} \] and \( K_B = \sum_{i=0}^{q-1} k_{B,i} \), respectively, which can be done by an Allreduce operation concurrently within the two groups. The messages to be sent from group \( A \) to group \( B \) are then segmented into blocks of size roughly \( \frac{K_A}{q} \), and the blocks from group \( B \) to group \( A \) into blocks of size roughly \( \frac{K_B}{p} \). Thus, processes having large blocks will send these over a number of send operations, while small blocks will be sent in a single operation.

The overall time for the exchange step will thus be determined by the largest blocks, that is \( \max(k_{A,i}, k_{B,j}) \). This exchange step ensures that all segments from group \( B \) will be present at processes in group \( A \) in segments of roughly the same size, and that all segments from processes in group \( A \) will likewise be present in group \( B \), such that final Allgather operations can collect the full messages at all processes in the two groups, as required. For process \( a_i \) in group \( A \), to compute which processes in group \( B \) it has to send its segments, it only has to compute the size of blocks at the processes \( a_0, a_1, \ldots, a_{i-1} \) with a lower rank than \( a_i \). Call this prefix sum which can be computed by an Exscan operation for \( R_{A,i} \). The first process in group \( B \) to receive a segment from \( a_i \) is then \( R_{A,i}/\frac{K_A}{q} \). The same applies to process \( b_j \) in \( B \). This ensures that all segments will be present in rank order in each of the groups, such that an Allgather operation can indeed complete the algorithm without any need for reordering. The cases where the block size is not exactly divisible can be handled as described for Algorithm 1. It is important to notice that the total sizes of segments received by a process are almost the same for all processes in the group, such that an Allgather operation with the same block sizes can be used.

Algorithm 2 describes our algorithm for intergroup Allgatherv more formally. To make the segmentation easier to describe, we assign a unique, global index for every byte of the messages in each of the groups, ordered based on the rank of processes.
Algorithm 2: Full-duplex Intergroup Allgatherv

1. \( k_A' \leftarrow \) Intra-group Allreduce (Sum) \( a_j \forall(0 \leq j < p) \) over \( k_{A,j} \)
2. \( k_B' \leftarrow \) Intra-group Allreduce (Sum) \( b_j \forall(0 \leq j < q) \) over \( k_{B,j} \)
3. \( r_A \leftarrow k_A \mod q \)
4. \( r_B \leftarrow k_B \mod q \)
5. for \( i \in [0, q-1] \) do
   6.     if \( i < r_A \) then
      7.         \( s_{A,i} \leftarrow \{ j : j \in [\lceil k_A' q \rceil, (i + 1)\lceil k_A' q \rceil) \} \)
   8.     else
      9.         \( s_{A,i} \leftarrow \{ j : j \in [\lceil k_A' q \rceil, (i - r_A)\lceil k_A' q \rceil, r_A\lceil k_A' q \rceil + (i + 1 - r_A)\lceil k_A' q \rceil) \} \)
   10.   end
11. end
12. for \( i \in [0, p-1] \) do
   13.     \( a_i \) send \( m_{A,i,j} \) to \( b_k \forall j \) such that \( m_{A,i,j}.index \in s_{A,k} \forall 0 < k < q \land m_{A,i}.index \cap s_{A,k} = \phi \)
14. end
15. for \( i \in [0, p-1] \) do
   16.     if \( i < r_B \) then
      17.         \( s_{B,i} \leftarrow \{ j : j \in [\lceil k_B' p \rceil, (i + 1)\lceil k_B' p \rceil) \} \)
   18.     else
      19.         \( s_{B,i} \leftarrow \{ j : j \in [\lceil k_B' p \rceil, (i - r_B)\lceil k_B' p \rceil, r_B\lceil k_B' p \rceil + (i + 1 - r_B)\lceil k_B' p \rceil) \} \)
   20.   end
21. end
22. for \( i \in [0, q-1] \) do
23.     \( b_i \) send \( m_{B,i,j} \) to \( b_k \forall j \) such that \( m_{B,i,j}.index \in s_{B,k} \forall 0 < k < p \land m_{B,i}.index \cap s_{B,k} \neq \phi \)
24. end
25. Intra-Allgather \( a_j \forall(0 \leq j < p) \) over \( m_B \)
26. Intra-Allgather \( b_j \forall(0 \leq j < q) \) over \( m_A \)

Formally, the global index of byte \( y \) in a message \( m_{A,x} \) is denoted by \( m_{A,x,y} \) and defined as \( m_{A,x,y}.index = \sum_{i=0}^{x-1} k_{A,i} + y \). We define the set of indices associated with message \( m_{A,x} \) as \( m_{A,x}.index = \{ \sum_{i=0}^{x-1} k_{A,i} + y \forall 0 \leq y < k_{A,x} \} \). For example, if \( m_{A,0} \) has a size of \( x \) bytes and \( m_{A,1} \) has a size of \( y \) bytes, \( m_{A,0,i}.index = i \) and \( m_{A,1,j}.index = x + j \) for \( 0 \leq i < x \)
and $0 \leq j < y$. $m_{A,0}.index = [0, x)$ and $m_{A,1}.index = [x, x + y)$. The same definition applies to $m_B$.

Initially, each of the groups sums up the total message size of their local groups in Lines 1-2. In Lines 5-24, both groups exchange their messages. Lines 5-14 and Lines 15-24 are similar. The first one is from group $A$ to group $B$ and the other is from group $B$ to group $A$. The definition of $s_{A,i}$ at Lines 7 and 9 ensure that the message received by every process in group $A$ at Line 13 is unique and has size either $\lfloor \frac{k_A'}{q} \rfloor$ or $\lceil \frac{k_A'}{q} \rceil$. Similarly, the definition of $s_{A,i}$ at Lines 17 and 19 ensure that every process in group $B$ receives unique messages of size either $\lfloor \frac{k_B'}{p} \rfloor$ or $\lceil \frac{k_B'}{p} \rceil$ at Line 23. Figure 2.3 gives an example of this stage. The balancing of the message sizes received by receivers at this stage is necessary for the MPI_Allgather operations in the next stage. Lines 25 and 26 perform intragroup Allgather for messages received from the remote group of each group.

At Lines 13 and 23 of Algorithm 2, the send operations are run in parallel. However, by the single-port communication constraint, a sender can send to one receiver and a receiver can receive from one sender at a time. Therefore, there may be delays of send/receive operations that can compromise the transfer time of the algorithm at Line 13 and 23. For example, in Figure 2.3, $a_3$ sends messages to $b_1$ and $b_2$. $b_2$ receives messages from $a_3$ and $a_4$. To achieve the best possible transfer time with the algorithm, it is crucial that sending and receiving is done in such a way that sending processes can be busy throughout.

We order the send/receive operations in the following way. If a sender sends messages to multiple receivers, the sender always sends its messages in the order of its receivers’ ranks. We call the senders that send to multiple receivers multi-target senders. A receiver decides the order of senders it receives messages from and senders should wait
for the receiving channel of the receiver to be idle in order to initiate the send operation. It is obvious that one receiver has at most two multi-target senders by the design of Algorithm 2. The receivers receive messages from the multi-target sender with higher rank, followed by senders with ranks between the two multi-target senders, followed by the multi-target sender with a lower rank. For example, in Figure 2.3 both $a_2$ and $a_3$ try to send messages to $b_1$. In such a case, the receiver receives messages from $a_3$ with higher priority, since $a_3$ is a multi-sender with the highest rank among all its senders. Another example is $b_2$, both $a_3$ and $a_4$ send messages to $b_2$. Moreover, both $a_3$ and $a_4$ are multi-target senders. Since $a_4$ has higher rank, $b_2$ receives messages from $a_4$ with higher priority. Theorem 3 presents an upper bound for the transfer time of Algorithm 2.

**Definition 1.** A receiver receives messages from a set of senders compactly if the total transfer time is equal to the senders’ total messages size times $t_w$.

**Theorem 3.** The transfer time term of Algorithm 2 is bounded by

\[
\begin{align*}
&\max \left( \sum_{i=0}^{p-1} k_{A,i} + \max(k_A), \sum_{i=0}^{q-1} k_{B,i} + \max(k_B) \right) t_w + \log(q) t_w \\
&\leq M + (\max(\max(k_A), \max(k_B)) + \log(q)) t_w
\end{align*}
\]

**Proof.** The Allreduce operations at Lines 1-2 have transfer time $\log(p) t_w + \log(q) t_w$.

Consider a multi-target sender $a_j$, let $b_i,..,b_{i+y}$ for $y > 1$ be its receivers. Process $b_i$ receives the message from $a_j$ at the beginning, since $a_j$ is the multi-target sender of $b_{i+y}$ with highest rank by the design of $S_{A,i+y}$. The processes $b_{i+z}$ for $0 < z < y$ receive messages exclusively from $a_j$, so the send operations are also not delayed. Now we consider process
$b_{i+y}$. Let $a_j, ..., a_{j+c}$ be its senders. If $a_{j+c}$ is a multi-target sender, $b_{i+y}$ receives messages in order of $\{a_{j+c}, a_{j+1}, ..., a_{j+c-1}, a_j\}$. The processes $\{a_{j+1}, ..., a_{j+c}\}$ are senders that only send to $b_{i+y}$ and $b_{i+y}$ is the lowest rank among all receivers of process $a_{j+c}$. If process $a_{j+c}$ is not a multi-target sender, $b_{i+y}$ receives messages in order of $\{a_{j+1}, ..., a_{j+c}, a_j\}$. Processes $\{a_{j+1}, ..., a_{j+c}\}$ are senders that send messages to $b_{i+y}$ exclusively. Therefore, $b_{i+y}$ has been compactly receiving messages when it starts to receive messages from $a_j$ in either of the cases. If the send operation from process $a_j$ to process $b_{i+y}$ is not delayed, the total transfer time of the send operations for $a_j$ is $k_{A,j}t_w \leq \max(k_A) t_w$. If the send operation from $a_j$ to $b_{i+y}$ is delayed, the total transfer time of the send operations for $a_j$ is less than $\left\lceil \frac{\sum_{i=0}^{p-1} k_{A,i}}{q} \right\rceil t_w$, since $a_j$ is the last sender of the receiver $b_{i+y}$ and $b_{i+y}$ can receive at most $\left\lceil \frac{\sum_{i=0}^{p-1} k_{A,i}}{q} \right\rceil \leq \max(k_A)$ messages by design of $S_{A,i+y}$.

If a sender $a_j$ is not a multi-target sender, it is obvious (from previous argument for $b_{i+y}$) that its only receiver $b_i$ receives messages compactly until receiving messages from $a_j$. Since $b_i$ can receive at most $\left\lceil \frac{\sum_{i=0}^{p-1} k_{A,i}}{q} \right\rceil \leq \max(k_A)$ messages by design of $S_{A,i}$, the transfer time of the send operation for $a_j$ is less than $\max(k_A) t_w$.

Line 25 is an Allgather operation of messages received from Lines 12-14 for processes in group $B$. The send size is $\text{size}_A = \left\lceil \frac{\sum_{i=0}^{p-1} k_{A,i}}{p} \right\rceil$. Therefore, the transfer time is $p(\text{size}_A - 1)t_w \leq \sum_{i=0}^{p-1} k_{A,i} t_w$. Consequently, $\left( \max(k_A) + \sum_{i=0}^{p-1} k_{A,i} \right) t_w$ is the total transfer time of Lines 12-14 and 25.

We can apply the same argument to derive that $\left( \max(k_B) + \sum_{i=0}^{q-1} k_{B,i} \right) t_w$ is the total transfer time of 22-24 and 26. Lines 12-14 and 25 run in parallel with Lines 22-24 and 26. Thus, the optimal transfer time bound in the theorem is proven. □
Theorem 3 shows that the message transfer ordering achieves optimal transfer time asymptotically. The term $\log(pq)t_w$ is negligible compared with $M$ for large message sizes and the term $\max(\max(k_A), \max(k_B))t_w$ is negligible for large $p$ and $q$. Therefore, Algorithm 2 along with the ordering proposed for the send operations, converges to the lower bound of the optimal transfer time $M$.

**Theorem 4.** Algorithm 2 correctly implements the intergroup Allgatherv operation.

Theorem 4 recapitulates the correctness of Algorithm 2 and follows from the explanation given above.
Figure 2.2. Example execution Algorithm 1 for $p = 8$ and $q = 3$. 
Figure 2.3. An illustration for the intergroup message exchange in Algorithm 2 for $p = 6$ and $q = 2$. (a) The send operations in Lines 12-14. (b) The send operations in line 22-24.
2.4. Experimental Results

To study the actual improvements achieved over the intergroup collective implementations in standard MPI libraries, a number of experiments have been performed on Cori, a Cray XC40 supercomputer at the National Energy Research Scientific Computing Center (NERSC). We have used the Cray MPI compiler(Cray-mpich/7.6.0), which is based on MPICH [6]. We have implemented our algorithms as described in the previous sections, using intra-communicator `MPI_Allgather` and `MPI_Allgatherv` as building blocks. The creation of new communicators is not necessary since an inter-communicator gives access to the intra-communicator that a given process belongs to. The `MPI_Allgather` and `MPI_Allgatherv` interfaces give the information on block sizes as required by the algorithms to perform the message segmentation.

For our experiments, we fill all message buffers with random numbers. We measure the `MPI_Allgather` 8 times and report the average completion time. As completion time for each individual measurement, we pick the maximum time among all processes. We implement Algorithm 1 and Algorithm 2 using MPI functions and make direct comparisons with the root gathering algorithm by calling the MPI library native `MPI_Allgather` and `MPI_Allgatherv` operations on the proper inter-communicators.

Algorithm 1 is equivalent to Algorithm 2 presented in [2] in terms of transfer time. Therefore, we expect the same performance. In this section, we present additional experimental results for cases where $p$ and $q$ are pairwise indivisible. We present four parameter settings, numbered from 1 to 4. Setting 1 is when $p = q$ and all processes are sending exactly the same data size. Setting 2 is when $\frac{p}{q} = \frac{25}{7}$ and all processes are sending exactly the same data size. This setting evaluates an imbalanced number of processes in the two
groups. Moreover, the sizes of the two groups are pairwise indivisible. Setting 3 is when $\frac{p}{q} = \frac{25}{7}$ and group A sends larger messages than group B. Setting 4 is when $\frac{p}{q} = \frac{25}{7}$ and group B sends larger messages than group A. Settings 3 and 4 evaluate imbalanced message sizes for the two groups.

We use four parameter settings, numbered from 5 to 8, on 64 nodes and 256 nodes to evaluate Algorithm 2 for MPI_Allgatherv. Setting 5 is when $p = q$ and all processes are sending exactly the same data size. This setting has the same end result as the first setting used to evaluate Algorithm 1. Setting 6 is when $p = q$ and processes in each of the groups are sending messages with block sizes forming an arithmetic sequence, e.g., rank $i$ sends a message of size $ki$ for some a chosen, fixed $k$. This setting evaluates extreme cases of imbalanced send sizes. Setting 7 is when $\frac{p}{q} = \frac{25}{7}$ and all processes are sending equal-sized messages. This setting has the same end result as the fourth setting used for evaluating Algorithm 1. Finally, setting 8 is when $\frac{p}{q} = \frac{25}{7}$ and the send sizes form an arithmetic sequence. This setting evaluates the case of imbalanced process numbers and irregular message sizes.

We experiment with the different parameter settings described on three MPI process configurations. For the first two configurations, one process is assigned to every node of 64 nodes and 256 nodes. Hence the total number of processes is 64 and 256 respectively. These two setups emulate a homogeneous communication cost model where a data transfer between any two processes has the same communication cost. For hybrid MPI and OpenMP applications that use shared memory at the same node and process communication across nodes, this setup is reasonable. For the last setup, we assign 32 processes to every node of the 64 nodes. Thus, the total number of processes is 2048. This
setup emulates applications that fully utilize cores with MPI. We perform experiments with a maximum message size from 64KBytes to 8MBytes per node. Figure 2.4 shows the run time comparison between Algorithm 1 and the MPI library native root gathering algorithm. Figure 2.5 shows the run time comparison between Algorithm 2 and the MPI library native root gathering algorithm.

As explained in Section 2.2.1, the version of the root gathering algorithm implemented by MPICH has transfer time \((M + 3pk_A + 3qk_B)t_w\) by Equation (2.2) for \(M = \max(pk_A, qk_B)\). The transfer time of Algorithm 1 is \(\max(pk_A, qk_B)t_w\). The expected improvement is therefore

\[
(2.6) \quad \frac{M + 3pk_A + 3qk_B}{M}.
\]

Figure 2.4 illustrates the end-to-end execution time of the root gathering algorithm, Algorithm 2 in [2], and Algorithm 1 from message size 64KBytes to 8MBytes per process with settings 1-4 and three process configurations mentioned previously. Since the theoretical transfer time of Algorithm 1 is equivalent to that of Algorithm 2 presented in [2], we expect a similar performance of these two algorithms for all settings when message size is large. The results in Figure 2.4 match this expectation. Setting 2, 3, and 4 provide additional results for settings that are not presented in [2], where \(p\) and \(q\) are indivisible of each other.

Figure 2.5 illustrates the end-to-end execution time of the root gathering algorithm and Algorithm 2 from message size 64KBytes to 8MBytes per node with settings 5-6 and three process configurations mentioned previously. The proposed algorithm shows a much better
performance than expected with 256 processes running on 256 nodes. This observation can be explained by the MPICH implementation that implements the intergroup MPI_Allgatherv slightly different from intergroup MPI_Allgather. The gathering operations

Figure 2.4. Allgather results for settings 1-4. Legend “Algorithm 1” refers to Algorithm 1 of this chapter, legend “EuroMPI” to Algorithm 2 in [2], and legend “Root Gathering” to the MPI library native root gathering algorithm. The plots are labeled with titles of the form “Number of processes (setting number)”. Settings for $p$, $q$, $k_A$, and $k_B$ have been defined in the text. The $x$-axis gives the maximum message size per node in Bytes, and the $y$-axis gives the completion time in milliseconds. The data type used is MPI_INT.
of messages to roots are implemented using a ring algorithm. If the communication
cost between nodes is the same, the transfer time of the ring-based intragroup Gather
algorithm achieves the lower bound of intragroup Gather. However, as mentioned before,
Cori is pseudo-fully connected, so every step of the ring algorithm is dominated by the

Figure 2.5. Algorithm 1 versus the MPI library native root gathering algo-

rithm for settings 5-8. The plots are labeled with titles of the form “Number
of processes (setting number)”. Settings for \( p, q, k_A, \) and \( k_B \) have been de-
fined in the text. The \( x \)-axis gives the maximum message size per node in
Bytes, and the \( y \)-axis gives the completion time in milliseconds. The data
type used is \texttt{MPI\_INT}. 

### Algorithm 2
largest communication cost between nodes with adjacent ranks, which can lead to poor performance as the number of nodes increases. As a result, the overall performance of the root gathering algorithm is worse than its theoretical performance on 256 nodes.

Figure 2.6 illustrates the improvements of Algorithm 1 and Algorithm 2 over the root gathering algorithm for all eight settings and three process configurations. When message sizes are small, the performance of the proposed algorithms is not necessarily better than
the performance of the root gathering algorithm, because the $t_w$ term is relatively small compared with $t_s$ which represents the communication cost independent of the message size. Discussion for small message sizes is beyond the scope of this chapter since our objective is to optimize the $t_w$ term. For large message sizes, we can observe that the performance of the proposed algorithms is many times faster than the performance of the root gathering algorithm. According to Equation 2.6, improvement curves for setting 1, 2, 3, 4, 5, 6, 7, and 8 should converge to 7.0, 4.84, 4.21, 6.68, 7.0, 7.0, 4.84, and 4.84 respectively as the message size increases. Since Cori is not strictly fully connected, the homogeneous communication cost assumption is not necessarily valid. Thus, the improvements may not converge to the exact ratios, since different pairs of arbitrarily selected processes can have different communication cost. Nonetheless, the significant improvements with large message size experiments demonstrate the robustness of the proposed algorithms under the weak assumption for communication cost for different process configurations and communication patterns.

2.5. Conclusion

In this chapter, we proposed scalable algorithms for the full-duplex intergroup All-to-All broadcast collective operations that can readily be used to implement the MPI_Allgather and MPI_Allgatherv collectives of MPI. Under single-port communication assumptions, the proposed algorithms achieve the lower bound for optimal transfer time asymptotically. Theoretical analysis shows that the proposed algorithms can achieve much lower transfer times than the root gathering algorithm that is adopted by many modern MPI production libraries. We have implemented the proposed algorithms. Experimental
results are in line with our theoretical analysis under a broad selection of configurations (numbers of processes, relative block sizes, process placement), and the practical gains correspond to the model predictions. In addition, these results show that the relative improvements are robust also for hierarchical systems where our homogeneous cost model does not apply. Consequently, we suggest using these implementations in MPI libraries instead of the now commonly used root gathering algorithm.

Intergroup collective operations embody communication patterns that cannot easily be expressed with the ordinary, intragroup collectives, namely bipartite patterns between processes in two disjoint groups where all communication is between groups and never within groups.

Although it is easily possible to improve over the root gathering algorithm also for the intergroup MPI_Bcast, MPI_Gather, MPI_Gatherv, MPI_Scatter, and MPI_Scatterv operations, partly by the ideas presented in this chapter, partly by reusing ideas from algorithms for the corresponding intragroup collectives, the full-duplex All-to-All operations still pose interesting challenges. While MPI can only express bipartite patterns over two disjoint process groups, it might be worthwhile to study multi-partite patterns both for use in concrete application and to gain insights into more general collective communication operations.
CHAPTER 3

Optimal Algorithms for Half-Duplex Inter-Group All-to-All
Broadcast on Fully Connected and Ring Topologies

Half-duplex inter-group collective communications in Message Passing Interface (MPI) are bipartite message transfer patterns such that the processes in a sender group transfer messages to the processes in a receiver group \[19\]. Compared with intra-group collective communication, where all processes are both senders and receivers, half-duplex inter-group collective communication does not enforce the senders to receive any messages and the receivers to send any messages. Implementation of inter-group collective communication in theory should have smaller communication cost compared with intra-group collective communication given the same total number of processes since the number of compulsory senders and receivers is less.

Argued in \[2\], inter-group collective communication is a basic communication pattern in scientific application workflows. For example, Liao et al. \[20\] propose a framework that allows parallel data transfer among workflow components in order to improve the performance of weather prediction systems SCALE-LETKF \[21\]. Hardware/Hybrid Accelerated Cosmology Code (HACC) \[34\], also has a demand for processing and transferring petabyte sized data in real-time \[35\]. To improve the performance of such systems, existing literature has focused on reducing the data size of communications. For instance, Zhang et al. \[23\] propose a distributed framework that maximizes on-chip data exchange,
which in turn reduces the communication frequency among components. Docan et al. 
[24] adopt an ActiveSpacing approach that moves programs to staging areas in order to reduce data exchange. Although reducing communication size and frequency can improve the performance of workflow systems, scientific application performance can also benefit from optimal inter-group collective communication algorithms.

The most widely-used algorithm for inter-group communication is the root gathering algorithm, summarized in [31] and [36]. The root gathering algorithm has two stages: Single-process accumulation followed by one-to-all broadcast/scatter. MPICH [6] and OpenMPI [8], the most widely used MPI implementations in the parallel processing community, adopt this strategy. However, the root gathering algorithm is not optimal because the receiving channels of the receivers are idle in at some stages.

In [2], we have proposed a full-duplex inter-group all-to-all broadcast algorithm on fully connected topology. In this chapter, we present optimal algorithms for half-duplex inter-group all-to-all broadcast (Allgather) under bidirectional communication constraint for any number of senders and receivers on both fully connected and ring topologies. Unlike the root gathering algorithm, the proposed algorithms reduce the idle time of the receiving channels for the receivers. Moreover, the proposed algorithm for fully connected topology has a more concise formulation and smaller startup latency compared with the full-duplex algorithm proposed in [2]. We provide detailed descriptions of algorithmic correctness and optimality.

For performance evaluation, we implement the proposed algorithms for inter-group all-to-all broadcast using MPI communication functions. Experiments are conducted on
Cori, a Cray XC40 supercomputer at the National Energy Research Scientific Computing Center (NERSC). The inter-group all-to-all broadcast can be achieved by the function MPI_Allgather using the inter-group communicator. For a fully connected topology case, direct comparisons with the function MPI_Allgather that adopts the root gathering algorithm are made. The communication network topology on Cori is pseudo fully connected (dragon-fly). We prove the point by evaluating and comparing our algorithms against the MPI library installed on Cori. For the ring topology case, we implement our proposed algorithms using the MPI_Sendrecv function to emulate a ring topology environment. The proposed algorithms are compared with the intra-group Allgather algorithm emulated under the same environment. The message size used for evaluations ranges from 32KB to 4MB. The proposed algorithm for the fully connected topology is up to 5 times faster than the root gathering algorithm. The proposed algorithm for the ring topology is up to 1.4 times faster than the intra-group Allgather algorithm.

3.1. Background and Related work

The communication model used in this chapter is based on the assumptions presented in [25], which are summarized below. Studies [26] have shown that these assumptions are widely assumed by collective communication algorithm designs.

(1) **Parallel architecture**: An undirected and connected graph represents a network, where processes are vertices and links are edges. Processes can only send/receive messages to/from other processes if there are direct links between them.
(2) **Bidirectional communication constraint:** When send or receive function at a process is called, the function is locked until the function returns. A process can receive and send messages at the same time.

(3) **Communication cost:** Let $t_w$ be communication transfer time per word and $t_s$ be communication startup time. Sending a message of size $k$ words from a sender to a receiver has communication cost $t_s + kt_w$. The term $t_s$ is called startup latency and the term $kt_w$ is called bandwidth.

### 3.1.1. Collective Communication

Collective communications defined by the MPI standard [19] have two categories: intra-group communication and inter-group communication. MPI communicator has an attribute that distinguishes these two categories.

Intra-group communication means that all processes are both senders and receivers. All processes in intra-group collective communication are symmetric. A process receives messages aggregated from the rest of the processes in the end, though intermediate messages received by an individual process may differ depending on the algorithm and topology.

Optimal intra-group communication algorithms have been well-studied [26]. Bertsekas [37] has proposed an optimal algorithm for all-to-all broadcast and total exchange on a hypercube topology. Thakur et al. [29] optimize intra-group Allgather using recursive doubling and Bruck algorithm [28] for non-power-of-2 number of processes.
Though intra-group communication is sufficient for supporting classical parallel computations such as matrix operations and prefix sums, demands for inter-group communication exists. Inter-group communication address the problem of bipartite message transfer. Instead of having a symmetric group of processes, processes are separated into two disjoint groups: one is the sender group and the other is the receiver group. The goal of inter-group communication is to deliver messages from the senders to the receivers. Although intra-group communication functions can achieve this goal by treating all processes as senders and receivers with the help of dummy messages, inter-group communication algorithms can have less communication cost compared with intra-group communication.

Existing works such as [31] and [36] present inter-group all-to-all broadcast (Allgather) implementations by extending MPI intra-group communication. All these works are based on the root gathering strategy: Single-process accumulation followed by one-to-all broadcast. MPICH [6] and OpenMPI [8], the most widely used MPI implementations in the parallel processing community, use this strategy. However, the root gathering algorithm is not optimal because the receiving channels of the receivers are idle in some stages. Thus, algorithms that allow inter-group all-to-all broadcast without using a root process can be very useful.

Optimal communication cost for inter-group communication depends on the topology of processes. We present algorithms for two important topologies: Full connected and ring. Fully connected topology is a reasonable assumption for modern supercomputers. For example, the supercomputer Cori at NERSC is pseudo fully connected (dragon-fly), which means that the shortest distance between any two remote computing nodes is a constant value. Moreover, algorithms that use binary indexing frequently assume fully
connected topology. For example, the multiple message broadcasting algorithm [38] assumes fully connected topology. Ring topology connects processes with a small number of edges. It has the advantage of handling non-power of two number of processes. For example, intra-group Allgather and intra-group one-to-all broadcast functions implemented by MPICH adopt a ring-based algorithm for handling non-power of two number of processes. Moreover, some systems are configured as multi-dimensional rings (N-dimensional torus). The optimal algorithm for ring topology can be readily extended to a multi-dimensional ring.

3.2. Algorithms for Fully Connected Topology

We first define the mathematical notations in this chapter. Let \( A = \{ a_0, \ldots, a_{p-1} \} \) and \( B = \{ b_0, \ldots, b_{q-1} \} \) be two disjoint arrays of processes. \( A \) denotes the set of sender ranks of size \( p \). \( B \) denotes the set of receiver ranks of size \( q \). Processes in \( A \) are labeled with ranks from 0 to \( p - 1 \). Processes in \( B \) are labeled with ranks from \( p \) to \( p + q - 1 \). Initially, every \( a_i \in A \) has a unique message \( m_i \) of size \( k \) words. The goal is to let \( b_j \forall 0 \leq j < q \) receive \( m_i \forall 0 \leq i < p \).

The theoretical lower bound for bandwidth can be established using the minimum time required for a single receiver to receive all messages. A receiver can receive \( p \) number of messages in a single step or multiple steps. Nevertheless, the total transfer time for bipartite communication takes at least \( pkt_w \). The time taken for all processes to receive required messages is longer than the time taken for a specific process to receive the required messages, so \( pkt_w \) is a theoretical lower bound for bandwidth.
Figure 3.1. Illustration of proposed Allgather algorithms for $p \geq q$ and $p < q$. The labels on arrows indicate message transferred. (a) First step for $p \geq q$. (b) Second step for $p \geq q$. (c) First step for $p < q$. (d) Second step for $p < q$. 
The theoretical lower bound for startup latency can be established using one-to-all broadcast from one sender to all receivers and all-to-one gather from all senders to one receiver, which are both sub-problems of inter-group all-to-all broadcast. Algorithms for one-to-all broadcast require at least \( \log (q + 1) \) steps. Since \( \log (q + 1) \leq \log (2q) = 1 + \log (q) \), \( 1 + \log (q) \) is a tight lower bound for startup latency. Alternatively, a receiver receives messages aggregated from \( p \) senders. Aggregating messages from \( p \) processes, requires at least \( \log (p + 1) \) steps, so \( \log (p) + 1 \) is another tight lower bound for optimal startup latency. Achieving either of the lower bound for startup latency would justify the optimality of startup latency of the proposed algorithms. Since \( p = q \) and the proposed algorithm has \( \log (p) + 1 \) number of steps, the startup time is optimal.

Suppose \( p = q \), the proposed algorithm can be divided into two phases. Message \( m_j \) is transfered from \( a_j \) to \( b_j \) \( \forall j \in [0, p - 1] \). This phase has communication cost \( (t_s + kt_w) \). Then, the receivers perform intra-group Allgather over messages \( m_j \) \( \forall j \in [0, p - 1] \). This phase has communication cost \( \log (p) t_s + (p - 1) kt_w \). The total communication cost is \( (1 + \log (p)) t_s + pk t_w \). It is obvious that all receivers receive messages from all senders.

In real-world applications, the number of senders and receivers are not necessarily equal, so strategies for handling \( p \neq q \) should be proposed.

When \( p > q \), \( q \) number of processes in group \( A \) concurrently send messages to group \( B \). The communication cost is \( t_s + kt_w \). The remaining \( p - q \) number of processes in the sender group joins the \( q \) processes in the receiver group, forming a group of size \( p \). The \( p \) processes spanned across two groups perform intra-group Allgather. The communication cost of this process is \( \log (p) t_s + (p - 1) kt_w \). The total communication cost is \( (1 + \log (p)) t_s + pk t_w \), which is the same as \( p = q \) case. As a result, the algorithm is optimal. Figure 3.1a
and 3.1b illustrate an example when \( p = 6 \) and \( q = 2 \). Although processes in group \( A \) are aggregating unnecessary messages to themselves during intra-group Allgather, the aggregated messages save the number of steps for the receivers to receive all messages. To elaborate on the necessity of using the sender channels, we consider an alternative algorithm that passes all messages from the senders to the receivers with \( \frac{p}{q} \) number of steps. Then the receivers perform intra-group Allgather by themselves. Although this algorithm does not send redundant messages, it has at least \( \frac{p}{q} \) of startup latency, which is not optimal for small message sizes.

When \( p < q \), the algorithm consists of two stages. In the first stage, the processes \( \{a_i, b_{i+pj} : 0 \leq j < \frac{q}{p}\} \forall i \in [0, p - 1] \) form broadcast groups with root \( a_i \). In every broadcast group, the root broadcasts its message to all other processes. In the second stage, the processes \( \{b_{ip+j} : 0 \leq j < p\} \forall i \in [0, \frac{q}{p} - 1] \) form subgroups of the receiver group, each with \( p \) number of processes. All subgroups of the receiver group execute intra-group Allgather independently. The one-to-all broadcasting has communication cost \( 2 \log \left(\frac{q}{p} + 1\right) t_s + 2kt_w \) using the multiple message broadcasting algorithm [38]. This communication cost for one-to-all broadcast is also claimed by the latest version of MPICH. The subgroup all-to-all broadcast has communication cost \( \log (p) t_s + (p - 1) kt_w \). Total communication cost is \( \left(\log (p) + 2 \log \left(\frac{q}{p} + 1\right)\right) t_s + (p + 1) kt_w \). When \( p \) and \( q \) are large enough, the total communication cost converge to \( \log (q) t_s + pkt_w \). Hence the total communication cost is optimal. Figure 3.1c and 3.1d illustrate an example when \( p = 2 \) and \( q = 6 \).
3.2.1. Comparison with Root Gathering Algorithm

The root gathering algorithm for inter-group all-to-all broadcast, adopted by both MPICH and OpenMPI, consists of three stages. In the first stage, processes in the sender group use intra-group Gather function to accumulate all messages to a root sender. Without loss of generality, we use $a_0$ to denote this root sender. This stage has communication cost $\log(p) t_s + (p - 1) k t_w$. In the second stage, $a_0$ send $p$ messages aggregated in the previous stage to the root $b_0 \in B$. This stage has communication cost $t_s + p k t_w$. Finally, root $b_0$ uses one-to-all broadcast function to pass accumulated messages to all receivers. Since the message size of broadcast is $p k$, this stage has communication cost $2 \log(q + 1) t_s + 2 p k t_w$. Therefore, the total communication cost is $(\log(p) + 2 \log(q)) t_s + 4 p k t_w$ for large $p$ and $q$.

Table 3.1 summarizes the communication cost of the proposed algorithm and the root gathering algorithm. When $p \geq q$, the startup latency of the proposed algorithm is $2 \log(q)$ steps less than the root gathering algorithm. When $p < q$, the startup latency of the proposed algorithm is $\log(q) + \log(p)$ steps less than the root gathering algorithm. The bandwidth of the proposed methods is approximately 4 times faster than the root gathering algorithm for any $p, q$. Since $q + 1$ and $\frac{p}{q} + 1$ can be non-power of two, the exact difference depends on the implementation strategies. Nevertheless, the proposed algorithm is a constant time faster than the root gathering algorithm.
Table 3.1. Comparison of theoretical communication cost for the proposed algorithm and the root gathering algorithm when the number of senders is $p$ and the number of receivers is $q$ on a fully connected network.

<table>
<thead>
<tr>
<th>Method</th>
<th>Startup Latency/$t_s$</th>
<th>Bandwidth/$kt_w$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root Gathering</td>
<td>$\log(p) + 2\log(q)$</td>
<td>$4p$</td>
</tr>
<tr>
<td>Proposed $p \geq q$</td>
<td>$\log(p)$</td>
<td>$p$</td>
</tr>
<tr>
<td>Proposed $p &lt; q$</td>
<td>$\log(p)$</td>
<td>$p$</td>
</tr>
</tbody>
</table>

3.3. Algorithms for Ring Topology

Ring topology connects processes with a small number of links. Rank $i$ has two edges: One is connected to rank $(i - 1) \mod (p + q)$ and the other is connected to rank $(i + 1) \mod (p + q)$. We use ranks $0, \ldots, p - 1$ to denote the senders and ranks $p, \ldots, p + q - 1$ to denote the receivers. One important assumption is that $p$ and $q$ are both even numbers for the convenience of our formulation. If they are not, a dummy process can be used to make up to the next even number.

The proposed optimal inter-group all-to-all broadcast algorithms on ring topology contain three stages. Every stage consists of multiple steps. The steps are send/receive functions executed in parallel. The entire algorithm is the concatenation of Algorithms 3, 4, and 5 describes all stages and steps of the algorithm when $p = q$. The $i$ loops represent the stages. The $j$ loops are the parallel steps. The concurrent labels in the algorithm refer to the ID of the message sequence defined as the following.

We briefly discuss the high-level ideas of Algorithms 3, 4, and 5 for transferring messages from the senders to the receivers. Like the intra-group Allgather algorithm for ring topology, where messages are circulated in a "pipeline" movement for $p + q - 1$ steps, the proposed algorithm creates multiple 'pipelines' of messages in different directions.
However, naive pipelining of messages from both sides (via process 0 and \( p - 1 \)) of the ring will break the bidirectional communication constraint mentioned in section 2, due to the limited number of connections. The problem is more complicated for \( p \neq q \) cases. The design of the proposed algorithms solves this issue by introducing the concept of a message sequence in Definition 2, a formal description of pipeline movements of messages. These message sequences are controlled by the proposed algorithms so that their movements do not violate the bidirectional communication constraint using interleaved indexing. Nevertheless, they can fulfill the objectives of half-duplex all-to-all broadcast with optimal communication costs.

**Definition 2.** A message sequence over an array of messages \( m_{i},...,m_{i+k} \) in a parallel communication algorithm is defined to be parallel steps such that the list of receivers for every element of the message array has either increasing or decreasing index order for all steps.

In the rest of this section, we present complete details of theoretical analysis for all cases. We suggest readers seeking for high-level insights to go through Figures 3.2 and 3.3 first. Algorithms 3, 4, and 5 describe the three stages of a ring algorithm for half-duplex intergroup Allgather. Algorithms 7, 8, and 9 discuss the three stages for handling \( p > q \). Section 4.1-4.4 and the second half of 4.5 should be the interest of people who favor comprehensive proofs of algorithms algebraically.

We prove that the three stages presented by Algorithms 3, 4, and 5 is feasible with respect to three constraints. The first constraint, referred as message availability, is that any message sent is available at the sender. The second constraint is the bidirectional
Algorithm 3: Optimal inter-group all-to-all broadcast on Ring Topology  
\( (p = q) \) stage 1.

1 for \( i \in \{0, \ldots, \frac{p}{2} - 1\} \) do
2     #Concurrent #1
3     for \( j \in \{\frac{p}{2}, \ldots, p - 1\} \) do
4         \( j + i \) send \( m_j \) to \( j + i + 1 \)
5     end
6     #Concurrent #2
7     for \( j \in \{0, \ldots, \frac{p}{2} - 1\} \) do
8         \((j - i) \mod (p + q)\) send \( m_j \) to \((j - i - 1) \mod (p + q)\)
9     end
10 #Concurrent #3
11 for \( j \in \{0, \ldots, i - 1\} \) do
12     \( \frac{p}{2} - i + 2j \) send \( m_{\frac{p}{2}-i+j} \) to \( \frac{p}{2} - i + 2j + 1 \)
13 end
14 end

Algorithm 4: Optimal inter-group all-to-all broadcast on Ring Topology  
\( (p = q) \) stage 2.

1 for \( i \in \{0, \ldots, \frac{p}{2} - 1\} \) do
2     #Concurrent #1
3     for \( j \in \{\frac{p}{2}, \ldots, p - 1\} \) do
4         \( \frac{p}{2} + j + i \) send \( m_j \) to \( \frac{p}{2} + j + i + 1 \)
5     end
6     #Concurrent #3
7     for \( j \in \{0, \ldots, \frac{p}{2} - 1\} \) do
8         \( 2j + i \) send \( m_j \) to \( 2j + i + 1 \)
9     end
10 end

communication constraint mentioned in section 2. The third constraint is message completeness, which means that every receiver receives all sender messages in the end.
Algorithm 5: Optimal inter-group all-to-all broadcast on Ring Topology
$(p = q)$ stage 3.

1 for $i \in \{0, \ldots, \frac{p}{2} - 1\}$ do
2 #Concurrent #1
3 for $j \in \{\frac{p}{2}, \ldots, p - 1\}$ do
4 if $p + j + i + 1 < 2p$ then
5 $p + j + i$ send $m_j$ to $p + j + i + 1$
6 end
7 end
8 #Concurrent #2
9 for $j \in \{1, \ldots, i\}$ do
10 $3p - i + 2j - 1$ send $m_{j-1}$ to $3p - i + 2j - 2$
11 end
12 #Concurrent #3
13 for $j \in \{0, \ldots, \frac{p}{2} - 1\}$ do
14 $\frac{p}{2} + i + 2j$ send $m_j$ to $\frac{p}{2} + i + 2j + 1$
15 end
16 end

Algorithm 6: Message Continuity

Data: $k \in \{1, -1\}, x, a, b, c \in \mathbb{Z}^+$
1 for $i \in \{0, \ldots, \frac{p}{2} - 1\}$ do
2 for $j \in \{0, \ldots, x\}$ do
3 $P(a + cj + ki) \mod (p+q)$ send $m_{(j+b) \mod (p+q)}$ to $P(a + cj + k(i+1)) \mod (p+q)$
4 end
5 end

3.3.1. Message availability

Theorem 5. Algorithm satisfies message availability constraint if process $P(a + cj + ki) \mod (p+q)$ has message

$m_{(j+b) \mod (p+q)}$ at the beginning $\forall j \in \{0, \ldots, x\}$. (Message continuity).

Proof. The sender always becomes the receiver in the next iteration of $i$ loop for any $j$. Hence message availability constraint is satisfied. □
In the three stages represented by Algorithms 3, 4, and 5, there are three message sequences (labeled with #). It is possible to show that they all obey message availability constraint.

For sequence #1 and #2, Theorem 5 justifies the message availability constraint by concatenating the i loops.

For sequence #3, we can examine all i loops. For the first i loop, let \( m_c \) be any arbitrary constant rank of message in message array of sequence #3. We have a linear system \( c = \frac{p}{2} - i + j \) with variables \( i \) and \( j \). Let \( i_0 \) be the very first iteration of i loop when \( m_c \) is transer in message sequence #3. \( i_0 = \frac{p}{2} - c \) is minimum when \( j = 0 \). Processor \( \frac{p}{2} - i_0 \) has \( m_{\frac{p}{2} - i_0} \) by definition. Thus, the base case for \( m_c \) is established. When \( m_c \) is sent at the \( i_k \geq i_0 \) iteration, the receiver of \( m_c \) is rank \( \frac{p}{2} - i_k + 2j_k + 1 = c + j_k + 1 \) for \( j = j_k \) in this iteration. In the \( i_k + 1 \) iteration, \( j = j_k + 1 \) is solved from the definition of \( c \). The sender of \( m_c \) is exactly process \( \frac{p}{2} - (i_k + 1) + 2(j_k + 1) = c + j_k + 1 \). Hence the induction of message availability of \( m_c \) is completed. Consider the receiver ranks in the last iteration of the first stage, \( \frac{p}{2} - \left( \frac{p}{2} - 1 \right) + 1 + 2j = 2j + 2 \) and \( j \) takes values from 0 to \( \frac{p}{2} - 2 \), so the senders at the beginning of the second stage are the receivers in the last iteration of first stage loop except process 0. By definition, process 0 has message \( m_0 \). Hence all messages sent at the start of the second stage are available. Message continuity theorem can be used to show that the second i loop fulfills the message availability constraint. For the third i loop, it continues the second loop for the range of i from \( \frac{p}{2} \) to \( p - 1 \). According to Theorem 5, sequence #3 is message complete.
3.3.2. Bidirectional communication constraint

Single-port communication constraint is not violated if the following two conditions are true. Every sender is unique within an \( i \) loop. Moreover, every receiver is unique within an \( i \) loop. The uniqueness across multiple \( j \) loops within an \( i \) loop can be verified by computing the ranges of sender/receiver ranks. If the ranges of sequence ranks do not overlap, the senders and receivers must be different for any \( i \). The ranges of sequence \#1 and sequence \#2 in the first \( i \) loop do not overlap. The ranges of sequence \#3 and sequence \#2 in the first \( i \) loop do not overlap.

When the ranges of senders and receivers in two sequences under an \( i \) loop overlap, proof by contradiction can be used to show uniqueness of ranks given the same arbitrary value of \( i \). Suppose a sender sends a message to different processes within an \( i \) loop. There exist two integers \( j_1 \) and \( j_2 \) within the ranges of \( j \) loops such that the ranks of senders are equal given the same \( i \). The same reasoning can be applied to receivers. We can enumerate the rest pairs of sequences.

Consider sequence \#1 and sequence \#3 in the first \( i \) loop. For senders, we have \( j_1 + i = \frac{p}{2} - i + 2j_2 \). It follows that \( j_1 = \frac{p}{2} - 2i + 2j_2 \). However, \( j_2 < i \) by definition. Hence \( j_1 < \frac{p}{2} \), which is outside its range \([\frac{p}{2}, p - 1] \). The same argument works for receivers because receiver ranks are sender ranks plus 1.

Consider sequence \#1 and sequence \#3 in the second \( i \) loop. For senders, we have \( \frac{p}{2} + j_1 + i = 2j_2 + i \). It follows that \( j_1 = 2j_2 - \frac{p}{2} \). \( j_2 \) takes values from 0 to \( \frac{p}{2} - 1 \). Hence \( j_1 < \frac{p}{2} \), which is outside the range \([\frac{p}{2}, p - 1] \). The same argument works for receivers because receiver ranks are sender ranks plus 1.
Consider sequence #1 and sequence #2 in the third $i$ loop. For senders, we have $p + j_1 + i = \frac{3p}{2} - i + 2j_2 - 1$. It follows that $j_1 = \frac{p}{2} + 2j_2 - 2i - 1$. $j_2 \leq i$ implies that $j_1 \leq p - 1$. However, $j_1 \geq \frac{p}{2}$ by definition. Therefore, no solution exists for $j_1$. For receivers, we have $p + j_1 + i + 1 = \frac{3p}{2} - i + 2j_2 - 2$, it follows that $j_1 = \frac{p}{2} + 2j_2 - 2i - 3$. $j_2 \leq i$ implies that $j_1 \leq p - 3$. However, $j_1 \geq \frac{p}{2}$ by definition. Therefore, solution for $j_1$ does not exist.

Consider sequence #1 and sequence #3 in the third $i$ loop. For senders, we have $p + j_1 + i = \frac{p}{2} + i + 2j_2$. It follows that $j_1 = 2j_2 - \frac{p}{2}$. $j_2$ takes values from range 0 to $\frac{p}{2} - 1$, so $j_1 \leq p - 1$. However, $j_1$ takes values in $[\frac{p}{2}, p - 1]$. The same argument works for receivers because receiver ranks are sender ranks plus 1.

Consider sequence #2 and sequence #3 in the third $i$ loop. For senders, we have $\frac{3p}{2} - i + 2j_1 - 1 = \frac{p}{2} + i + 2j_2$. It follows that $2(j_2 - j_1 + i) = p - 1$. By assumption, $p$ is an even number and $i$ is an integer, so integer solutions for $j_1$ and $j_2$ do not exist. For receivers, we have $\frac{3p}{2} - i + 2j_1 - 2 = \frac{p}{2} + i + 2j_2 + 1$. It follows that $2(j_2 - j_1 + i) = p - 3$. Integer solutions for $j_1$ and $j_2$ do not exist.

3.3.3. Completeness

Completeness means that all messages from senders are received by all receivers when the algorithm finishes.

**Theorem 6.** Let $M_1$ be message sequence over message array $m_i, \ldots, m_{i+c}$ and $M_2$ be message sequence over message array $m_{i+c}, \ldots, m_i$ such that the receivers of $M_1$ and $M_2$ have reverse index order. If $\forall j \in [i, i + c]$, $m_j$ in both sequences is received by some pair of
adjacent receivers (\(m_j\) intersects in both sequences), process set \(P'\) that contains processes between the front senders of \(M_1\) and \(M_2\) is message complete.

\[\textbf{Proof.}\] Suppose that \(\forall j \in [i, i + c]\), \(m_j\) is received by adjacent receivers \(v - 1\) and \(v\) in both sequences. Without loss of generality, suppose \(M_1\) has receivers in increasing index order at all steps and \(M_2\) has receivers in decreasing index order at all steps. By definition of message sequence, when \(v - 1\) receives \(m_j\) in \(M_1\), all processes in \(P'\) with ranks less than \(v - 1\) have received message \(m_j\). When \(v\) receives \(m_j\) in \(M_2\), all processes in \(P'\) with ranks greater than \(v\) have received message \(m_j\). Therefore, \(P'\) is message complete over the message array. \(\Box\)

For sequence \#1, it is clear that it travels through all the receivers by combining all the three stages. Hence receivers are message complete for messages \(\{m_j : j \in \left[\frac{p}{2}, p - 1\right]\}\).

For messages \(\{m_j : j \in \left[0, \frac{p}{2} - 1\right]\}\), we can show that the message \(m_j\) in sequence \#2 and sequence \#3 intersects at some receiver. Because all receivers are between the front senders of the two sequences, the theorem justifies message completeness. There is no stop condition in both sequences, so all messages in the sequence are transferred without delay. Therefore, \(m_j\) moves exactly \(\frac{p}{2} + \frac{p}{2} - 1 - j = p - 1 - j\) hops in sequence \#2 to decreasing index direction. The final receiver of \(m_j\) in sequence \#2 is \((j - (p - 1 - j)) \mod 2p = 2j + p + 1\).

For message sequence \#3, message \(m_j\) at the start of stage 2 travels \(p\) hops in increasing direction. The final receiver of \(m_j\) in sequence \#3 is \(2j + p\). Hence the final receiver differs by 1 index, which means that the sequences intersect. It follows from Theorem 6 that all receivers have received messages \(\{m_j : j \in \left[0, \frac{p}{2} - 1\right]\}\).
Table 3.2. Comparison of communication cost for the proposed algorithm and intra-group Allgather algorithm when the number of senders is \( p \) and the number of receivers is \( q \) on ring network.

<table>
<thead>
<tr>
<th>Method</th>
<th>Startup latency/( t_s )</th>
<th>Bandwidth/( kt_w )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intra-Allgather</td>
<td>( p + q - 1 )</td>
<td>( p + q - 1 )</td>
</tr>
<tr>
<td>Proposed</td>
<td>( p + \frac{q}{2} )</td>
<td>( p + \frac{q}{2} )</td>
</tr>
</tbody>
</table>

### 3.3.4. Optimality

The optimal algorithm for inter-group all-to-all broadcast on ring topology that contains \( p \) senders with lower ranks and \( q \) receivers with higher ranks has at least \( k \left( p + \frac{q}{2} - 1 \right) t_w \) bandwidth. This statement can be proved by considering the receiver at rank \( p + \frac{q}{2} - 1 \) and \( p + \frac{q}{2} \). Both of them need to receive \( p \) messages. Moreover, it takes at least \( \frac{q}{2} - 1 \) steps for the first message to arrive at either of them. As a result, the communication cost is at least \( \left( p + \frac{q}{2} - 1 \right) (kt_w + t_s) \).

The total communication cost of Algorithms 3, 4, and 5 is the number of \( i \) iterations multiplied by \( t_s + t_w \). Hence the communication cost is \( \left( p + \frac{q}{2} \right) (t_s + t_w) k \), which converges to optimal bandwidth bound \( \left( p + \frac{q}{2} - 1 \right) kt_w \) for large \( p, q \).

Table 3.2 summarizes the complexities of communication cost compared with intra-group Allgather algorithm. The improvement of communication cost is \( \frac{p+q-1}{p+\frac{q}{2}} \). Based on this ratio, we expect that the improvement increases as the number of receivers increases.

### 3.3.5. Imbalanced Graph

In this section, we discuss methods for handling imbalanced numbers of senders and receivers. The three main stages of the Algorithms 3, 4, and 5 remain unchanged.
Figure 3.2. An example (left to right and top to bottom) of the proposed algorithm for the ring topology all-to-all broadcast when \( p = 4 \) and \( q = 6 \). Message available at every step is labeled above/below a process. Arrows illustrate messages transferred. The total number of steps is 7.

When \( p > q \), the principle is the same as \( p = q \) case. The directions for all three sequences remain the same. The length of sequence #1 is \( p - \frac{q}{2} \) over message array \( \{m_{\frac{q}{2}}, ..., m_{p-1}\} \). The length of sequence #2 and #3 is \( \frac{q}{2} \) over message array \( \{m_0, ..., m_{\frac{q}{2}-1}\} \). The first and third stage have \( \frac{q}{2} \) steps. The second stage has \( p - \frac{q}{2} \) steps. The previous proofs based on the three sequences are valid. The total communication cost is \((p + \frac{q}{2})(t_s + t_w)k\). Hence the algorithm is optimal. Figure 3.3 illustrates an example when \( p = 4 \) and \( q = 6 \).

When \( p < q \), as long as the process \( p + q - 1 \) receives any messages delivered by sequence #1, there is at least a \( q(t_s + t_w)k \) component in the total communication cost. If \( q \) is significantly larger than \( p \), the communication cost is not optimal. Optimal communication cost can be reached by adding a sequence #4 that is symmetric to sequence
Figure 3.3. An example (left to right and top to bottom) of the proposed algorithm for the ring topology all-to-all broadcast when \( p = 6 \) and \( q = 4 \). Message available at every step is labeled above/below a process. Arrows illustrate messages transferred. The total number of steps is 8.

#3 over messages \( \{m_{\frac{q}{2}} - 1, \ldots, m_0\} \) in the opposite direction at every stage. Algorithms 7, 8, and 9 describes the three stages for handling \( p < q \) case. For message availability, the proof for sequence #1, #2, and #3 in previous section can be reused. Sequence #4 is symmetric to sequence #3 in opposite direction, so message availability is ensured. Figure 3.2 illustrates an example when \( p = 6 \) and \( q = 4 \).

For bidirectional constraint, we show that the new sequence does not conflict with existing sequences. Sequence #4 is divided into two parts in the second \( i \) loops. The computing of \( i \) takes ceiling of the result. The range of sequence #4-1 does not overlap the ranges of sequence #1 and #2 in the second stage. Moreover, the range of sequence #3 does not overlap the ranges sequence #2 and sequence #4-2 in the second stage.

Consider sequence #1 and sequence #4 in the first \( i \) loop. For senders, we have 

\[
j_1 + i = 2j_2 - i - \frac{p}{2} + 1.
\]

It follows that 

\[
j_1 = 2j_2 - 2i - \frac{p}{2} + 1.
\]

Since \( j_2 \leq \frac{p}{2} + i - 1 \),
Algorithm 7: Optimal inter-group all-to-all broadcast on Ring Topology (p<q) stage 1

1 for $i \in \{0, ..., \frac{p}{2} - 1\}$ do
2     #Concurrent #1
3     for $j \in \{\frac{p}{2}, ..., p - 1\}$ do
4         $j + i$ send $m_j$ to $j + i + 1$
5     end
6     #Concurrent #2
7     for $j \in \{0, ..., \frac{p}{2} - 1\}$ do
8         $(j - i) \mod (p + q)$ send $m_j$ to $(j - i - 1) \mod (p + q)$
9     end
10    #Concurrent #3
11    for $j \in \{0, ..., i - 1\}$ do
12        $\frac{p}{2} - i + 2j$ send $m_{\frac{p}{2} - i + j}$ to $\frac{p}{2} - i + 2j + 1$
13    end
14    #Concurrent #4
15    for $j \in \{\frac{p}{2}, ..., \frac{p}{2} + i - 1\}$ do
16        $2j - i - \frac{p}{2} + 1$ send $m_j$ to $2j - i - \frac{p}{2}$
17    end
18 end

$j_1 \leq \frac{p}{2} - 1$, which is out of the range of $j$ in sequence #1. For receivers, we have $j_1 + i + 1 = 2j_2 - i - \frac{p}{2}$, it follows that $j_1 = 2j_2 - 2i - \frac{p}{2} - 1$. Since $j_2 \leq \frac{p}{2} + i - 1$, $j_1 \leq \frac{p}{2} - 3$, which is out of the range of $j$ in sequence #1. The same argument works for receivers because receiver ranks are sender ranks plus 1.

Consider sequence #2 and sequence #4 in the first $i$ loop. For senders, we have $j_1 - i = 2j_2 - i - \frac{p}{2} + 1$. It follows that $j_1 = 2j_2 - \frac{p}{2} + 1$. Since $j_2 \geq \frac{p}{2}$, $j_1 \geq \frac{p}{2} + 1$, which is out of the range of $j$ in sequence #2. The same argument works for receivers because receiver ranks are sender ranks minus 1.

Consider sequence #3 and sequence #4 in the first $i$ loop. For senders, we have $\frac{p}{2} - i + 2j_1 = 2j_2 - i - \frac{p}{2} + 1$. It follows that $2j_1 = 2j_2 - p + 1$. By assumption, $p$ is even. Hence $2j_2 - p + 1$ is odd. Integer solutions for $j_1$ do not exist. For receivers, we have
\textbf{Algorithm 8:} Optimal inter-group all-to-all broadcast on Ring Topology
(p<q) stage 2

\begin{algorithm}
\begin{algorithmic}
\State for $i \in \{0, \ldots, \frac{q}{2} - 1\}$ do
\State \#Concurrent #1
\For{$j \in \{\frac{p}{2}, \ldots, p - 1\}$}
\State $\frac{p}{2} + j + i$ send $m_j$ to $\frac{p}{2} + j + i + 1$
\EndFor
\State \#Concurrent #2
\For{$j \in \{0, \ldots, \frac{p}{2} - 1\}$}
\If{$i < \frac{q}{2} - \frac{p}{2}$}
\State $\frac{p}{2} + q - i + j$ send $m_j$ to $\frac{p}{2} + q - i + j - 1$
\EndIf
\EndFor
\State \#Concurrent #3
\For{$j \in \{0, \ldots, \frac{p}{2} - 1\}$}
\State $2j + i$ send $m_j$ to $2j + i + 1$
\EndFor
\State \#Concurrent #4-1
\For{$j \in \{\frac{p}{2} + \frac{i}{2}, \ldots, p - 1\}$}
\State $2j - i - p + 1$ send $m_j$ to $2j - i - p$
\EndFor
\State \#Concurrent #4-2
\For{$j \in \{\frac{p}{2}, \ldots, \frac{p}{2} + \frac{i}{2} - 1\}$}
\If{$2j > 2i + \frac{p}{2}$}
\State $2j - i + q + 1$ send $m_j$ to $2j - i + q$
\EndIf
\EndFor
\EndFor
\end{algorithmic}
\end{algorithm}

$\frac{p}{2} - i + 2j_1 + 1 = 2j_2 - i - \frac{p}{2}$. It follows that $2j_1 = 2j_2 - p - 1$. By assumption, $p$ is even. Hence $2j_2 - p - 1$ is odd. Integer solutions for $j_1$ and $j_2$ do not exist.

Consider sequence #3 and sequence #4-1 in the second $i$ loop. For senders, we have $2j_1 + i = 2j_2 - i - p + 1$. It follows that $2(j_2 - j_1) = 2i + p - 1$. By assumption, $p$ is even. Hence $2i + p - 1$ is odd. Integer solutions for $j_1$ and $j_2$ do not exist. For receivers, we have $2j_2 - i - p = 2j_1 + i + 1$. It follows that $2(j_2 - j_1) = 2i + p + 1$. By assumption, $p$ is even. Hence $2i + p + 1$ is odd. Integer solutions for $j_1$ and $j_2$ do not exist.
Consider sequence #1 and sequence #4-2 in the second $i$ loop. For senders, we have $p_j + j + i = 2j_2 - i + q + 1$. It follows that $j_1 = 2j_2 - 2i + q - \frac{p_j}{2} + 1$. The condition enforces $2j_2 > 2i + q - \frac{p_j}{2} > 2i + \frac{p_j}{2}$, so $j_1 > p + 1$, which is out of the range of $j$ in sequence #1. For receivers, we have $p_j + j_1 + i + 1 = 2j_2 - i + q$. It follows that $j_1 = 2j_2 - 2i + q - \frac{p_j}{2} - 1$. The condition branch enforces $2j_2 > 2i + q - \frac{p_j}{2} > 2i + \frac{p_j}{2}$, so $j_1 > p - 1$, which is out of the range of $j$ in sequence #1.

Consider sequence #2 and sequence #4-2 in the second $i$ loop. For senders, we have $p_j + q - i + j_1 = 2j_2 - i + q + 1$. It follows that $j_1 = 2j_2 - \frac{p_j}{2} + 1$. Since $j_2 \geq \frac{p_j}{2}$, $j_1 \geq \frac{p_j}{2} + 1$, which is outside the range of $j$ in sequence #2. The same argument works for receivers because receiver ranks are sender ranks minus 1.

Consider sequence #2 and sequence #1 in the second $i$ loop. For senders, we have $p_j + q - i + j_1 = \frac{p_j}{2} + i + j_2$. It follows that $j_2 = q - 2i + j_1$. Since $i < \frac{q}{2} - \frac{p_j}{2}$ by the conditional branch and $i$ is an integer, $2i < q - p - 1$. It follows that $j_2 > p + j_1 + 1$.
However, \( j_1 \geq 0 \), so \( j_2 > p + 1 \), which is out of range of \( j \) in sequence \#1. For receivers, we have \( \frac{p}{2} + q - i + j_1 - 1 = \frac{p}{2} + i + j_2 + 1 \). It follows that \( j_2 = q - 2i + j_1 - 2 \). Since \( i < \frac{q}{2} - \frac{p}{2} \) by the conditional branch and \( i \) is an integer, \( 2i < q - p - 1 \). It follows that \( j_2 > p + j_1 - 1 \). However, \( j_1 \geq 0 \), so \( j_2 > p - 1 \), which is out of range of \( j \) in sequence \#1.

Proof for message completeness can be divided into two parts: \( \{m_j : j \in \left[0, \frac{p}{2} - 1\right]\} \) and \( \{m_j : j \in \left[\frac{p}{2}, p\right]\} \).

For messages \( \{m_j : j \in \left[0, \frac{p}{2} - 1\right]\} \), we can show that the message \( m_j \) in sequence \#2 and sequence \#3 intersects at some receiver. \( m_j \) moves exactly \( \frac{p}{2} + \left(q - \frac{p}{2} - \frac{p}{2} - 1 - j\right) = \frac{p}{2} + \frac{q}{2} - 1 - j \) hops in sequence \#2 to decreasing index direction. \( m_j \) moves exactly \( j + \frac{q}{2} + \frac{p}{2} \) hops in sequence \#3 to increasing index direction. The final receiver of \( m_j \) in sequence \#2 is \( \left( j - \left(\frac{p}{2} + \frac{q}{2} - 1 - j\right) \right) \mod (p + q) = 2j + \frac{p}{2} + \frac{q}{2} + 1 \). The final receiver of \( m_j \) in sequence \#3 is \( 2j + \frac{q}{2} + \frac{p}{2} \). Hence the final receiver differs by 1 index, which means that all messages intersect. It follows from Theorem 5 that all receivers have received messages \( \{m_j : j \in \left[0, \frac{p}{2} - 1\right]\} \).

For \( \{m_j : j \in \left[\frac{p}{2}, p - 1\right]\} \), we can show the message \( m_j \) in sequence \#1 and sequence \#4 intersects at some receiver for any \( j \in \left[0, \frac{p}{2} - 1\right] \) or \( m_j \) in sequence \#1 reaches \( p + q - 1 \) at the end of the algorithm. If \( m_j \) is forced to stop at some receiver by the condition in sequence \#4-2, it would intersect message \( m_{p-1} \) in the second \( i \) loop. Because there are \( \frac{p}{2} \) steps that sequence \#1 will move forward in stage 3, \( m_j \) in sequence \#1 must intersect \( m_j \) in sequence \#4. On the other hand, if the condition in sequence \#4-2 is not triggered for \( m_j \), \( m_j \) travels as far as \( \frac{q}{2} + \frac{p}{2} - \left(j - \frac{p}{2} + 1\right) \) hops, so the final receiver is \( \left( j - \left(\frac{q}{2} + \frac{p}{2} - \left(j - \frac{p}{2} + 1\right) \right) \right) \mod (p + q) = 2j + \frac{q}{2} + 1 \). The final receiver of \( m_j \) in sequence \#1 is \( \min \left(p + q - 1, j + p + \frac{q}{2}\right) \). If the final receiver of \( m_j \) in sequence \#1 is
$p+q-1$, the case is finished. Otherwise the final receiver is $j + p + \frac{q}{2}$. Since $j \in \left[\frac{p}{2}, p-1\right]$, $j + p + \frac{q}{2} \geq 2j + 1 + \frac{q}{2}$. Hence $m_j$ in both sequences must have intersected. It follows from Theorem 6 that all receivers have received messages $\{m_j : j \in \left[\frac{p}{2}, p-1\right]\}$.

3.4. Experimental Results

We implement the proposed algorithms for inter-group all-to-all broadcast on fully connected topology using MPI_Send, MPI_Recv, and MPI_Allgather with intra-group communicators as building blocks. To make a fair comparison, we emulate the root gathering algorithm using the MPI_Bcast and MPI_Allgather with intra-group communicators as building blocks. The timing performance of this implementation for the root gathering differs from the performance of calling MPI_Allgather with an inter-group communicator by less than 3%. Thus, comparing the proposed algorithms with the emulation of the root gathering algorithm is also a direct comparison. The root gathering algorithm is designed for topologies that supports binary indexing, so it is highly inefficient on a ring topology. Thus, we use the intra-group Allgather as the benchmark for the ring topology. We implement the proposed algorithms using MPI_Sendrecv function under ring topological constraints. This implementation is compared with the intra-group Allgather algorithm under ring topological constraints implemented using the same MPI_Sendrecv function. Improvement of communication cost is computed as the benchmark communication cost divided by the proposed algorithm communication cost.

The experiments are performed on Cori, a Cray XC40 supercomputer at the National Energy Research Scientific Computing Center (NERSC). We use the Cray MPI compiler (cray-mpich/7.6.0) for MPI functions, which is based on MPICH. We adopt two process
Figure 3.4. These figures illustrate the comparison between the proposed algorithm and the benchmark algorithm in terms of communication cost. The title format is 'no. senders/no. receivers, topology'. The unit of horizontal axis is number of bytes per message. The unit of vertical axis is the communication cost in seconds.

configurations. The first configuration has 256 nodes. Every node is assigned with a single process. For hybrid MPI and OpenMP applications that use shared memory at the same node and process communications among nodes, this setup is reasonable. The second
configuration has 128 nodes. Every node is assigned with 8 processes. MPI programs that utilize cores spanning across nodes. The messages are randomly generated data with size from 32KB to 4MB. If message size smaller than 32KB is used, the bandwidth is trivial, so the proposed algorithm does not improve performance significantly. All timing results are averaged over eight trials of experiments.

### 3.4.1. Fully Connected Topology

Figure 3.4a illustrates the comparisons of communication cost with 512 senders and 512 receivers running on two groups of 64 compute nodes, respectively. The communication cost improvement is 1.8 with 4MB message size. Figure 3.4d illustrates the comparisons of communication cost with 128 senders and 128 receivers running on two groups of 128 compute nodes, respectively. The communication cost improvement is 5.3 with 4MB message size. As the message size increases, the difference of total communication cost becomes significant because the bandwidth starts to dominate the total communication cost. Moreover, the improvement increases as the number of processes becomes larger, since the total message size received by a process increases.

Figure 3.4b illustrates the comparisons of communication cost with 256 senders and 768 receivers running on two groups of 32 and 96 compute nodes, respectively. The communication cost improvement is 2.0 with 4MB message size. Figure 3.4e illustrates the comparisons of communication cost with 64 senders and 192 receivers running on two groups of 64 and 192 compute nodes, respectively. The communication cost improvement is 5.1 with 4MB message size.
Figure 3.4c illustrates the comparisons of communication cost with 768 senders and 256 receivers running on two groups of 96 and 32 compute nodes, respectively. The communication cost improvement is 1.2 with 4MB message size. Figure 3.4f illustrates the comparisons of communication cost with 192 senders and 64 receivers running on two groups of 192 and 64 compute nodes, respectively. The communication cost improvement is 4.4 with 4MB message size.

The communication cost improvements are much better for the configuration that assign one process per node. When multiple processes are assigned to the same node, the communication cost between processes are not uniform, since a processes within the same node communicate with each other faster than processes located in remote nodes. This configuration violates the assumption we made in section 2. Nevertheless, we demonstrate that the proposed algorithm still run faster than the root gathering algorithm.

3.4.2. Ring Topology

Figure 3.4g illustrates the comparisons of communication cost with 512 senders and 512 receivers running on two groups of 64 compute nodes, respectively. Figure 3.4j illustrates the comparisons of communication cost with 128 senders and 128 receivers running on two groups of 128 compute nodes, respectively. The communication cost improvements are 1.7 and 1.9 with 4MB message size.

Figure 3.4h illustrates the comparisons of communication cost with 256 senders and 768 receivers running on two groups of 8 and 24 compute nodes, respectively. Figure 3.4k illustrates the comparisons of communication cost with 64 senders and 192 receivers running on two groups of 2 and 6 compute nodes, respectively. The communication cost
improvements are both 1.9 with 4MB message size. Compared with $p = q$, the improvements with large message size is larger. This improvement matches the complexity comparison in Table 3.2 because the difference of communication cost between the intra-group Allgather and the proposed algorithm is proportional to the number of receivers.

Figure 3.4 illustrates the comparisons of communication cost with 768 senders and 256 receivers running on two groups of 24 and 8 compute nodes, respectively. Figure 3.4 illustrates the comparisons of communication cost with 192 senders and 64 receivers running on two groups of 6 and 2 compute nodes, respectively. The communication cost improvements are 1.8 and 1.4 with 4MB message size. The communication cost improvement is less compared with the previous two cases, because the difference of communication cost between the intra-group Allgather and the proposed algorithm is positively related to the number of receivers in Table 3.2.

The theoretical communication cost improvements for the proposed ring algorithms are 1.5, 1.14, and 1.6 for $p = q$, $p = 2q$, and $2p = q$ when there are $p$ senders and $q$ receivers according to Table 3.2. We expect to observe communication cost improvement close to the theoretical values when the total number of processes is large enough. This observation can be made as we increase the total number of processes from 256 to 1024.

3.5. Conclusions

Inter-group communication is a fundamental communication pattern for scientific workflow systems. In this chapter, we propose optimal algorithms of inter-group half-duplex all-to-all broadcast designed for both ring topology and fully connected topologies.
Both theoretical and experimental results have shown that the proposed algorithms outperform the benchmarks. In the future, it is possible to extend our proposed algorithms from ring topology to torus topology.
CHAPTER 4

Improving MPI Collective I/O for High Volume Non-contiguous Requests With Intra-node Aggregation

The message passing interface (MPI) standard defines a set of programming interfaces for parallel shared-file access, commonly denoted as MPI-IO [19]. Many large-scale scientific applications adopt MPI-IO directly or indirectly through parallel I/O libraries to obtain high I/O performance [39, 40, 41, 42, 43]. There are two types of MPI-IO functions: collective and independent. The collective functions require all processes that collectively open the same shared file to participate in the calls. Such a requirement provides an opportunity for an MPI-IO implementation to coordinate activities between processes to achieve better performance. A well-known example is the two-phase I/O strategy [5], which has become the implementation backbone for collective I/O in almost all MPI libraries.

Two-phase I/O conceptually consists of a communication phase and an I/O phase. A subset of the MPI processes, defined as I/O aggregators, act as I/O proxies for the rest of the processes. The aggregate access file region of a collective I/O call is divided among the aggregators into nonoverlapping regions, called file domains. In the communication phase, all processes send their I/O requests to the aggregators based on their file domain assignments. In the I/O phase, aggregators make system calls to read from or write the received requests to the file. The two-phase I/O strategy has delivered high performance
on parallel machines in the past two decades. The success of a two-phase I/O strategy relies on a fast communication network by paying a relatively small cost on exchanging request data among processes to obtain a higher gain in file system access time. This trade-off works effectively as the speed of I/O systems is much slower than the communication systems. However, as the scale of parallel computers grows, soon into an exascale computing era, the number of processes running applications also increases. The communication of the two-phase I/O exhibits an all-to-many message exchange pattern, whose cost may exceed the I/O phase for large parallel jobs when the number of non-contiguous I/O requests is significant, due to the high possible communication congestion on the I/O aggregators [44, 45].

In this chapter, we present an improvement for MPI collective I/O, denoted as the two-layer aggregation method (TAM), which adds an intra-node request aggregation layer so that the communication in the two-phase strategy consists of two layers of request aggregations. In the intra-node aggregation layer, MPI processes running on the same compute nodes perform a request aggregation to a subset of processes, denoted as local aggregators. In contrast to local aggregators, we denote the I/O aggregators in the original two-phase I/O as the global aggregators. As communication takes place among processes on the same node, the cost is expected to be relatively small. This intra-node aggregation performs on all compute nodes independently and concurrently. Once receiving the requests from local processes, the local aggregators coalesce them into fewer contiguous requests. After coalescing non-contiguous I/O requests, the local and global aggregators across compute nodes enter the traditional two-phase I/O to complete collective read or write operations. In this chapter, we refer to the communication between local and global
aggregators as the inter-node aggregation. An advantage of TAM is the reduction of the number of inter-node communications at global aggregators since the number of local aggregators is much less than the number of processes. In the traditional two-phase I/O, each global aggregator may receive requests from all other MPI processes, potentially causing communication contention at the global aggregators for large-scale applications with a significant number of non-contiguous I/O requests. The intra-node aggregation alleviates such a problem by breaking the all-to-many communication into two layers so that the global aggregators receive requests only from the local aggregators.

We implement TAM in ROMIO, the implementation of the MPI-IO functions used most frequently in HPC and provided by vendors as part of their MPI implementation [46]. Our performance evaluations are conducted on Theta, Cray XC40 parallel computer with Intel KNL processors at the Argonne National Laboratory, and Cori, a Cray XC40 supercomputer with Intel Haswell processors at the National Energy Research Scientific Computing Center (NERSC). Comparisons of TAM against the traditional two-phase I/O from the latest implementation of ROMIO are presented using three I/O benchmarks: E3SM-IO [42], S3D-IO [47], and BTIO [48]. E3SM-IO and S3D-IO are I/O kernels of two large-scale production applications E3SM and S3D, respectively, while BTIO is a benchmark from NASA’s NAS Parallel Benchmarks. These benchmarks contain a vast number of non-contiguous I/O requests that result in inter-node communication congestion for two-phase I/O as the number of nodes scale. From the experimental results, we observe a significant time reduction in communication costs. The intra-node communication cost, which is the extra cost introduced by TAM, vanishes as the number of processes increase, which matches our expectation. We analyze how the choice of the number of TAM local
aggregators affects the performance from the inter-node communication network utilization and congestion point of view. The overall performance improvements are up to 29 and 6.7 times faster for collective write and read.

In Section 4.1, we discuss the two-phase I/O bottlenecks and existing solutions. In Section 4.2, we propose our TAM improvement for two-phase I/O in detail. Finally, we evaluate the TAM in Section 4.3.

4.1. Background

The implementation of two-phase I/O [49] in ROMIO selects a subset of MPI processes, denoted as I/O aggregators, as proxies to carry out the file access operations for the remaining processes. The aggregate access file region of a collective I/O call is divided among the aggregators into nonoverlapping regions, called file domains. Each aggregator is responsible for reading and writing for its file domain assigned to the file. When rearranging the requests from non-aggregators to aggregators, also known as the communication phase, aggregators gather I/O requests that intersect its file domain. In the I/O phase, aggregators coalesce the received requests and read/write data from/to the file system.

Many MPI libraries such as MPICH [6] and OpenMPI [8] adopt ROMIO as the implementation for MPI-IO functions. For parallel jobs running multiple MPI processes per compute nodes, ROMIO selects one aggregator per node in its default settings. The selection of I/O aggregators happens at file open time.
Due to the fact of I/O devices being the slowest hardware component of a parallel computer system, the I/O phase is often the bottleneck of collective I/O. Many strategies have been proposed in the past decades to reduce its cost. Ma improved MPI-IO output performance with active buffering and threads [50]. A strategy that aligns the file domains with the file locking protocol is presented in [51, 52]. Chaarawi and Gabriel proposed an algorithm that selects the number of aggregators automatically based on the file view and process topology [53]. LACIO is developed as a strategy to exploit the logical I/O access pattern among processes and physical layouts of file access to optimize I/O performance [54]. Two phases I/O pipelining that overlaps the communication with file access is proposed in [55, 56]. With the emerging of the solid-state driver (SSD), the cost of file access can be further reduced [57]. Burst buffering is proposed to take advantage of faster SSD devices to improve parallel I/O performance [58, 59]. Although SSD cannot always replace traditional hard disks, hybrid usage of both disks by placing requests with high I/O cost to a small number of SSDs can achieve reasonably good I/O performance [60, 61, 62].

Research has been conducted recently to reduce the timing cost of the communication phase. Tsujita et al. proposed a method for overlapping the I/O phase with the communication phase with the help of multi-threading [63]. Cha and Maeng applied a node reordering approach for reducing communication costs with non-exclusive scheduling [64]. MPICH-G2 presents a multi-level topology-aware strategy for MPI collective communication that can improve communication by considering reducing both intra-node and inter-node traffic [65]. TAPIOCA proposes a topology-aware two-phase I/O algorithm that takes advantage of double-buffering and one-sided communication to reduce the process
idle time during data aggregation [66, 67]. HierKNEM, a kernel-assisted topology-aware collective framework, improves communication performance on multi-core systems by using multiple layers of collective algorithms [68]. Chakraborty et al. further improves the kernel-assisted collective techniques by reducing communication contention [69]. Optimizations that consider the communication topology have demonstrated their potentials for enhancing the two-phase I/O performance.

The request rearrangement in the two-phase I/O exhibits an all-to-many communication pattern, where for collective write operations, all the MPI processes send their requests to the I/O aggregators. Our recent study for the I/O performance of E3SM [42] model has shown that the communication phase dominates the overall performance for writing cubed sphere variables whose I/O pattern consists of a long list of small and non-contiguous requests on every MPI process. This communication pattern can cause network contention as the number of processes scales.

4.2. Design

Our proposed new method consists of three steps for a collective I/O operation: intra-node aggregation, inter-node aggregation, and I/O phase. Focusing on communication among processes running on the same node, the intra-node aggregation gathers all requests into a subset of processes, denoted as local aggregators. During this step, there is no communication taking place across compute nodes. Once the requests are received, the local aggregators coalesce the requests into a potentially smaller number but larger contiguous requests. During the inter-node aggregation step, local aggregators send the coalesced requests to the global aggregators based on the assigned file domains. The
Table 4.1. This table summarizes the terms used in Chapter 4.

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>number of processes in collective I/O</td>
</tr>
<tr>
<td>$q$</td>
<td>number of processes per node in collective I/O</td>
</tr>
<tr>
<td>$c$</td>
<td>number of local aggregators per node</td>
</tr>
<tr>
<td>$k$</td>
<td>average number of I/O requests per process</td>
</tr>
<tr>
<td>$L$</td>
<td>set of local aggregators</td>
</tr>
<tr>
<td>$G$</td>
<td>set of global aggregators</td>
</tr>
<tr>
<td>$p_l$</td>
<td>number of local aggregators</td>
</tr>
<tr>
<td>$p_g$</td>
<td>number of global aggregators</td>
</tr>
<tr>
<td>$v$</td>
<td>I/O amount in bytes</td>
</tr>
</tbody>
</table>

I/O phase remains the same as the original two-phase I/O: The global aggregators fulfill the gathered I/O requests with the file system. Thus, the proposed techniques are independent of underlying file systems.

Both collective read and write distribute metadata of I/O requests from all processes to local aggregators, followed by local aggregators to global aggregators, and global aggregators to file systems. Data in collective write has the same flow as its metadata of I/O requests. Data in collective read, on the other, flows from file systems to global aggregators, followed by global aggregators to local aggregators, and local aggregators to all processes. Consequently, the data communication pattern of collective read is the reverse of the collective write data exchange pattern.

Let $p$ be the number of processes that participate in collective I/O operations with process rank IDs ranging from 0 to $p - 1$. For processes that are placed on the same node, we also use local rank IDs, the ascending order of process rank IDs on this node, to refer to them. For instance, a compute node with three process rank IDs $\{2, 5, 7\}$ placed on it has local rank IDs $\{0, 1, 2\}$. Let $k$ be the average number of I/O requests per process. Let
be a set of local I/O aggregators, and \( p_l \) be the number of local aggregators in total. Let \( G \) be the global aggregators selected in the original two-phase I/O, and \( p_g \) denote the number of global aggregators. We keep the number of global aggregators the same as the original two-phase I/O because the selection of global aggregators has been optimized for the I/O stage. However, our approach does not limit the choice of global aggregators. For the rest of this chapter, we refer to the term communication contention as the additional communication cost when there are a large number of concurrent receive operations at global aggregators. The defined terms are summarized in Table 4.1.

4.2.1. Intra-node Aggregation

The intra-node aggregation selects one or more local MPI processes as local aggregators. Each local process sends its requests to one of the local aggregators on the same node. We partition local processes evenly into \( p_l \) groups, so each local aggregator receives from approximately the same number of non-aggregators. A local aggregator gathers non-contiguous I/O requests for processes with rank IDs higher than its rank and smaller than the next local aggregator’s rank. Since each non-aggregator sends its request to only one local aggregator, the communication exhibits a many-to-one pattern.

Our design selects local aggregators based on a spread-out principle. The selection of local aggregators depends only on local rank IDs within a node, so the real rank IDs do not have to be contiguous within a node. Formally, let \( q \) be the number of MPI processes running on a compute node, \( c \) be the number of local aggregators on the node, and \( e = (q \mod c) \). We select processes with local rank IDs \( \left\lfloor \frac{q}{c} \right\rfloor \cdot i \) for \( i = 0, 1, \cdots, e - 1 \) and \( \left\lfloor \frac{q}{c} \right\rfloor e + \left\lfloor \frac{q}{c} \right\rfloor (i - e) \) for \( i = e, \cdots, c - 1 \) as local aggregators.
Sometimes a process rank can serve as both local and global aggregators. Figure 4.1 uses two examples to illustrate the selection policy for local and global aggregators: one (Figure 4.1 (a)) is when the number of compute nodes is equal to the global aggregators, and the other one (Figure 4.1 (b)) is when there are more compute nodes than the global aggregators. For both cases, \( c = 3 \) out of \( q = 8 \) MPI processes on each node are local aggregators, so \( e = (8 \mod 3) = 2 \). Hence local rank IDs \( \{0, 3, 6\} \) for every node are local aggregators. In the former case, there are three global aggregators. The local aggregator with the lowest rank ID per node also serves as a global aggregator. In the latter case, only three out of six nodes contain global aggregators, i.e., on nodes 0, 2, and 4, so rank IDs 0, 16, and 24 are both local and global aggregators.

We implement TAM in ROMIO. The current implementation of ROMIO stores the hostnames of all processes at open time, so local rank IDs that allow us to compute the rank IDs of local aggregators, are available without extra communication cost. The input of MPI collective I/O has an MPI Datatype that defines the file access regions from the file view of the process. At the start of collective I/O, the MPI Datatype is flattened into offset-length pairs that define the non-contiguous regions of file access. For collective write, every process first sends the number of its flattened non-contiguous I/O requests and the size of its write data to its corresponding local aggregator, followed by sending the write data. For collective read, processes send their non-contiguous I/O requests to local aggregators in the same way as the collective write. Data exchange from local aggregators to all processes, on the other hand, happens when the local aggregators have received the read data from global aggregators.
The proposed local aggregator selection policy takes advantage of the possibility that non-contiguous I/O requests from processes of adjacent ranks are likely contiguous, which allows coalescing by the same local aggregator. Furthermore, the input non-contiguous I/O requests to collective I/O are in monotonously non-decreasing order according to the MPI standard, so ROMIO implements two-phase I/O using this fact. If TAM is implemented on the top of existing ROMIO two-phase I/O, the local aggregators must sort all its requests before entering the inter-node and I/O phases of two-phase I/O because local aggregators are the providers of non-contiguous I/O requests from two-phase I/O perspective. Once local aggregators have gathered all non-contiguous I/O requests, they sort non-contiguous I/O requests into a monotonously non-decreasing order based on the file offsets. The offset-length pairs received from a non-aggregator are already sorted in monotonically non-decreasing order themselves, due to the requirement by the MPI-IO standard [19]. We apply the heap merge sort algorithm to merge and sort all non-contiguous I/O requests gathered at each local aggregator. Its time complexity is $O \left( \frac{p_k}{p_i} \log \left( \frac{p}{p_i} \right) \right)$. After sorting all non-contiguous I/O requests, the aggregators coalesce any two consecutive requests that are contiguous.

ROMIO implements the two-phase I/O communication kernel with point-to-point MPI asynchronous functions MPI_Isend and MPI_Irecv followed by MPI_Waitall, so we adopt the same communication functions for the intra-node and inter-node communications for TAM to make a direct comparison with the current ROMIO implementation in the experimental result section. Point-to-point communication functions can avoid the necessity for constructing new sub-communicators by reusing the input communicator. However,
collective communication functions, such as `MPI_Alltoallw`, may improve communication performance.

The aggregated data size at local aggregators can exceed the memory limit when the input file size is large. To resolve this problem, we combine the intra-node aggregation phase with the original two-phase I/O that finishes in multiple rounds. However, this implementation strategy sacrifices the advantage of intra-node I/O requests coalescing, resulting in a larger metadata size exchanged in the inter-node aggregation phase later.

![Figure 4.1. Examples of placement for local and global aggregators on distributed compute nodes.](image)

(a) The placement of 12 local and three global aggregators on three compute nodes while 8 MPI processes are running on each node. This configuration illustrates the case when the number of nodes equals the number of global aggregators. (b) The placement of 16 local and two global aggregators on six nodes in the case the number of nodes is more than the global aggregators. Blue circles represent global aggregators. Green circles represent local aggregators. A global aggregator can also serve as a local aggregator. The placement policy for local and global aggregators is to spread them out evenly among the available resource to prevent possible communication contention.
Figure 4.2. Illustrations of communication pattern in a collective write operation for two-phase I/O in (a) and TAM in (b). Each square box represents a compute node, and circles are MPI processes. The six green circles are local aggregators, and the three blue ones are global aggregators. Inter-node and intra-node communications are blue and green arrows, respectively. A one-to-one mapping connects the global aggregators and the file servers. In (a), there is no local aggregation phase. The all-to-many (24-to-3) communication pattern shows a potential contention at the global aggregators. In (b), the communication clearly shows a reduced contention (6-to-3) at the global aggregators with the help of local aggregation phase.
4.2.2. Inter-node Aggregation

Inter-node aggregation is essentially the communication phase of the original two-phase I/O, but with participation from only the local aggregators as the I/O requesters. In the traditional two-phase I/O, the communication is an all-to-many, between all \( p \) processes and \( p_g \) global aggregators. Figures 4.2 (a) and (b) illustrate the communication complexity of two-phase I/O and TAM, respectively. It is clear to observe the reduction of communication contention on the global aggregators when TAM is used. The two-phase I/O can be considered a special case of TAM when \( p_l \) is equal to \( p \). In this case, the intra-node aggregation is skipped.

If the number of global aggregators is higher than the number of compute nodes, there will be more than one global aggregator placed on the same compute node. If the number of global aggregators is less than the number of compute nodes, then a subset of compute nodes is selected, and one process on each of the selected compute nodes becomes a global aggregator. In this case, some compute nodes do not contain global aggregators. Our scheme spreads out global aggregators across all ranks on a node because the method may utilize hardware resources better when multiple CPUs are on the same node. For example, each Haswell node of the supercomputer Cori has two CPUs with a total of 32 cores. Processes with ranks from 0 to 15 are on the cores from the first CPU, and processes with ranks from 16 to 31 are placed on the cores from the second CPU by default. Spreading out aggregators on the same node maximizes the utilization of both CPUs on the same node. However, TAM can adapt to any existing approach for global aggregator placements. For example, Cray’s MPI-IO evenly distributes global aggregators across all nodes. Within a node, it assigns the lowest ranks as global aggregators by default.
At the end of intra-node aggregation, every local aggregator has \( \frac{p_k}{p_l} \) number of sorted offset-length pairs on average. Every local aggregator computes lists of its non-contiguous I/O requests to different global aggregators based on the file domains assigned to them. For collective write, local aggregators send their non-contiguous I/O requests to global aggregators, which presents a many-to-many inter-node communication. For applications with a high volume of non-contiguous access requests, it is frequent that a global aggregator has non-contiguous I/O requests from all processes evenly. Thus, a global aggregator receives \( \frac{p_k}{p_l} \) number of requests from every local aggregator. For collective read, the inter-node data communication pattern is reversed, but the metadata and data contents per communication remain the same. Local aggregators receive data from global aggregators in this stage. Therefore, the inter-node data communication pattern is \( p_g \) to \( p_l \) for collective read, instead of \( p_l \) to \( p_g \) in the case of collective write. Since \( p_g \) is not necessarily equal to \( p_l \), this asymmetry of communication patterns can cause a difference in collective read and write performance.

For two-phase I/O, each global aggregator receives \( \frac{p_k}{p_g} \) number of requests on average from every process during the inter-node aggregation. Sorting requests received from local aggregators with heap merge sort algorithm can improve performance since file requests can coalesce. This approach can significantly reduce the I/O cost by avoiding exchanging a large number of non-contiguous I/O requests with file servers. However, sorting is not necessary if the implementation chooses to apply data sieving, which is enabled by default for collective read in ROMIO. If data sieving is disabled, sorting a \( p_l \) number of sorted arrays of requests for the traditional two-phase I/O is \( O \left( \frac{p_k}{p_g} \log p \right) \). TAM has offset sorting operations at both intra-node aggregation and inter-node aggregation. The
total offset sorting complexity of intra-node and inter-node aggregation phases for TAM is $O\left(\frac{p_k}{p_l} \log\left(\frac{p}{p_l}\right) + \frac{p_k}{p_g} \log(p_l)\right)$. When $p_l \geq p_g$, TAM has a smaller time complexity of sorting than two-phase I/O for collective write. This assumption is valid for file systems that choose $p_g$ values based on available file servers that are much less than the number of compute nodes.

4.2.3. I/O Phase

In our design, the I/O phase remains the same as the original two-phase I/O implemented in ROMIO. Only global aggregators enter this phase. In addition, two-phase I/O and TAM have the same input and output at the I/O phase.

After inter-node aggregation, a global aggregator has gathered non-contiguous I/O requests within its file domain. When data sieving is enabled, a large chunk of data, with start and end offsets defined by the minimum and maximum byte of the I/O requests gathered at a global aggregator, is read from the file system. For collective read, data buffers for I/O requests are filled using memory copy from this large chunk of data. For collective write, a global aggregator updates the data sieving chunk according to the I/O requests. Later, the data sieving chunk is written back to file servers as a contiguous chunk. If data sieving is not enabled, the read and write for non-contiguous requests from file servers are independent I/O.

Although different file systems store files differently, the improvement of TAM over two-phase I/O is not limited to a specific file system. As long as all global aggregators have to receive high volumes of non-contiguous I/O requests from a large number of
processes, TAM can improve the performance since the inter-node communication cost can be reduced with the help of intra-node aggregation.

4.3. Experimental Results

Table 4.2. Datasets used in our evaluation in Chapter 4. For E3SM F and G benchmarks, the non-contiguous requests are collected from production runs using 21600 and 9600 MPI processes, respectively. As we present the strong-scaling results, the non-contiguous requests are partitioned among all $p$ processes used in our experiments. BTIO and S3D-IO have more significant numbers of non-contiguous requests, and the numbers increase as the number of processes $p$. For S3D-IO, $x$, $y$, and $z$ are the number of processes used to partition $X$, $Y$, and $Z$ dimensions. $x \cdot y \cdot z$ is the total number of processes.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># non-contiguous requests</th>
<th>I/O amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>E3SM G</td>
<td>$1.72 \times 10^8$ to $1.76 \times 10^8$</td>
<td>85GiB</td>
</tr>
<tr>
<td>E3SM F</td>
<td>$1.35 \times 10^9$ to $1.37 \times 10^9$</td>
<td>14GiB</td>
</tr>
<tr>
<td>BTIO</td>
<td>$40 \times 512^2 \sqrt{p}$</td>
<td>5GiB</td>
</tr>
<tr>
<td>S3D-IO</td>
<td>$16 \times 800^2 x$</td>
<td>61GiB</td>
</tr>
</tbody>
</table>

We conduct our experiments on Theta, a Cray XC40 parallel computer system with Intel KNL processors at Argonne National Laboratory and Cori, a Cray XC40 supercomputer with Intel Haswell processors at the National Energy Research Scientific Computing Center (NERSC). Each KNL node contains one CPU with 64 CPU cores. Each Haswell node contains two CPUs with a lower count of 16 CPU cores on each CPU. Moreover, the bisection bandwidths of the interconnect network of these two supercomputers are different. Running benchmarks on both systems enable us to study whether TAM adapts to different hardware systems and whether the communication contention of two-phase I/O can occur on different hardware architectures. We set the Lustre stripe size to 1 MiB and stripe count to 56, the total number of available OSTs on Theta. All performance
results are presented in a strong-scaling evaluation. For every parameter setting, we average timings from five independent executions. Using four I/O benchmarks, E3SM-IO (G and F cases), BTIO, and S3D-IO, we present a performance comparison between TAM and the two-phase I/O implementation in ROMIO.

The ROMIO library from the MPICH release version of 3.3 implements collective read and write differently for Lustre file systems. By default, the implementation of Lustre collective write driver differs from the implementation of collective read driver in ROMIO in terms of the file domain partitioning and collective buffer size. Collective read adopts a static-cyclic file domain partitioning strategy. Each global aggregator has a contiguous file domain that aligns with lock boundaries. File domains of global aggregators for collective write on Lustre file systems, on the other hand, are group-cyclic. With our Lustre setting, each global aggregator owns \( \frac{x}{56\text{MiB}} \) number of file domain chunks of size 1MiB for writing a file of size \( x \). Two-phase I/O finishes read and write in multiple rounds. \( p_g \) is equal to Lustre stripe size (56) by default for collective write to avoid lock contention. The maximum size of any file domain handled by ROMIO Lustre collective write driver per round is a contiguous Lustre stripe, which is 56MiB in our Lustre setting. For collective read, \( p_g \) is equal to the number of compute nodes, and the maximum data size handled per round by each global aggregator is a contiguous chunk of 16MiB by default for all file systems. Thus, the collective read driver reads at most \( p_g \) contiguous 16MiB file domains per cycle. The difference in the implementations for collective read and write creates a chance for us to evaluate the impact of \( p_g \) and collective buffer size on performance for TAM.
We customize the default ROMIO library into a version that performs at least as well as the Cray MPICH library. This customized version is used as the baseline of our evaluations. TAM is implemented on top of this customized version ROMIO. Firstly, we modify the default MPICH ROMIO to have the same global aggregator placement as Cray MPI for collective write. Using the environmental variable "MPICH_MPIIO_HINTS_DISPLAY," we observe that Cray MPI selects global aggregators from different compute nodes in a round-robin fashion on Lustre file systems. For example, to select four aggregators from 2 nodes, each running 64 MPI processes with contiguous ranks, Cray MPI picks processes with rank IDs in the order of \{0, 64, 1, 65\}. Cray MPI library is not open-source, so we use experiments to analyze its algorithms for collective read and write. According to our analysis on Cray MPI library (compiler craype/2.6.1 and toolkit cray-mpich/7.7.10) with Darshan DXT utility to dump file offsets and lengths information for all MPI processes, Cray MPI library selects the Lustre stripe size of global aggregators by default for collective read and write with the file domain partitioning and collective buffer size identical to the ROMIO collective write implementation. Such implementation of collective read has sub-optimal performance compared with the ROMIO collective read implementation given a large number of processes and nodes because collective read does not suffer from performance degradation of Lustre lock contention [52]. Therefore, we adopt the current ROMIO collective read implementation strategy to carry out the evaluations of two-phase I/O and TAM for collective read evaluation.

We replace all `MPI_Isend` with `MPI_Issend` during the aggregation. This change is critical for a large number of collective I/O requests, where the two-phase I/O must be carried out in multiple rounds. Posting asynchronous send requests using `MPI_Isend`
may be cached by the operating system if the message size is small. In this case, at the end of each round of two-phase, even though a call to `MPI_Waitall` is made, processes may continue into the next rounds and post more asynchronous send requests. Therefore, the send requests accumulate in the message queue. A high number of pending asynchronous send requests could seriously hurt the communication performance, due to the possible overwhelming in the message queue processing. Replacing `MPI_Isend` with `MPI_Issend` prevents non-aggregators from continuing into the next round of two-phase I/O, as `MPI_Issend` requires all pending send requests to be received before `MPI_Waitall` returns.

4.3.1. E3SM-IO Case Study

E3SM [42] is an exascale earth system modeling program for simulating atmosphere, land, and ocean behavior in high resolution. It is an I/O module developed to use PIO library [70], which is built on top of PnetCDF [71]. Writing data at each checkpoint is through posting nonblocking PnetCDF APIs that allow small requests to be aggregated and flushed together. PnetCDF library is a high-level parallel I/O library popular in the climate research community, which is built on top of MPI-IO. Flushing pending nonblocking requests are implemented by aggregating the request data and combining the MPI file views before making a single call to the MPI collective write function. In our experiments, the cost of posting the nonblocking APIs is negligible, and thus, we measure the timing of collective write calls inside the PnetCDF flush API.

We evaluate two data decompositions used in E3SM production runs, namely F and G cases [42]. The I/O kernel of E3SM has been extracted for I/O study [72]. F case
Figure 4.3. The I/O bandwidth comparisons of TAM and two-phase I/O in the strong-scaling evaluation from 256 to 16K processes on KNL and Haswell nodes. Haswell nodes have 32 physical cores, and KNL nodes have 64 physical cores. We set the number of processes per node equal to the number of available physical cores. Setting $p_l = p$ automatically degrades TAM to two-phase I/O. For our TAM in these comparisons, we set $p_l = 512$ (or $p_l = p$ if $P < 512$) based on empirical results. For I/O patterns with a large number of non-contiguous I/O requests, this strategy works well. We explain why such $p_l$ yields better performance later.

has atmosphere, land, and runoff models components. The data access pattern in the F case consists of 1.36 billion non-contiguous write requests and a total write amount of
Figure 4.4. The degree of contention for all datasets is denoted as the average number of receive operations for metadata exchange at global aggregators. TAM has 512 local aggregators in the comparisons.

14 GiB. The G case has active ocean and sea-ice components. G case decomposition file has an MPAS grid data structure that consists of pentagons and hexagons on top of a spherical surface. The I/O pattern of G case contains a shorter list of non-contiguous requests, 180 million in total, and a data size of 85 GiB. The data request file offsets and lengths of individual processes from the production runs are recorded. In both F and G cases, the numbers of non-contiguous requests are different among processes, but the difference is small. To present the performance in a strong-scaling, we assign requests of all processes from the production run evenly among the processes used in our experiments. The assignment is based on the units of processes. In other words, the entire offset-length pairs from a process in the production run is assigned to a process. The non-contiguous I/O requests in E3SM-IO do not have a subarray pattern. Adjacent file offsets do not have a regular interval. Thus, the file access patterns of E3SM-IO differ from BTIO and S3D-IO.

Figure 4.3 shows I/O bandwidth comparisons of TAM and two-phase I/O in the strong-scaling evaluation from 256 to 16K processes on KNL and Haswell nodes. We do not present results beyond 16K processes since the two-phase I/O performance results show performance slow down towards 16K processes already. Similar performance degradation
is expected as the number of processes scales further. We set 512 local aggregators for experiments on both KNL and Haswell nodes. This setting of $p_l$ for TAM is not necessarily the optimal value for all datasets. Nevertheless, we prove the point that TAM improves the performance of collective I/O without tuning parameters harshly. In general, the results on Cori Haswell have higher bandwidth than the results on Theta KNL. A study of communication and I/O bandwidth on Cori also presents similar results [73]. Haswell node has more sockets and fewer cores (processes) per node, so processes on the same Haswell node share fewer resources compared with KNL nodes. Thus, the contention is less significant on Haswell nodes. In addition, performance degradation of write results is more significant than the performance degradation of collective read results for two-phase I/O on both Haswell and KNL nodes, due to the differences in the implementations of collective read and write mentioned earlier. Nevertheless, two-phase I/O running on both systems drops its bandwidth as the number of processes increases from 4K to 16K. TAM, on the other hand, does not encounter such performance degradation, though two-phase I/O can outperform TAM for smaller jobs, due to the intra-node aggregation overheads of TAM. From Figure 4.3, we observe that TAM maintains a good write/read bandwidth when the number of processes increases.

We use the average number concurrent receive operations at global aggregators for metadata exchanges to illustrate communication contention for all datasets in Figures 4.4. This metric is equivalent to the average number of concurrent outgoing and incoming communication requests at global aggregators for collective read and write data exchange. From Figures 4.4 (a) and (b), we can observe that the degree of contention of E3SM F
and G cases using two-phase I/O scales with the number of processes. TAM, on the other hand, has a smaller degree of contention.

For intra-node aggregation, three components are contributing to the timing. The first component is the communication for gathering metadata to local aggregators. The communication pattern is many-to-one, so the total number of MPI send requests is \( p \) to be received by \( p_l \) local aggregators. Each of the \( \frac{p}{p_l} \) gathering operations within a node can thus run simultaneously. The second component is to merge-sort the request file offsets at every local aggregator. The third component is memory operation for moving the request data into a contiguous space based on the sorted offsets. All three components have timings proportional to request data amount and the number of offsets. Therefore, as we increase the number of local aggregators \( p_l \), the data amount and the number of offsets per local aggregator decrease, so the time of intra-node aggregation decreases proportionally. From Figures 4.5 and 4.6 we observe that the intra-node aggregation time decreases proportionally with increasing number of local aggregators. Consequently, the cost of the intra-node aggregation reduces and becomes negligible when running TAM on a large number of nodes.

In the strong-scaling study, the total read/write amount stays the same regardless of the number of processes. On the Lustre system, the number of global aggregators is fixed for collective write, so the cost of the I/O phase is constant. For collective read, \( p_g \) is equal to the number of compute nodes, so I/O phase cost vanishes as the number of compute nodes increases if global aggregators have similar file domain sizes. Thus, the bottleneck of the performance for TAM is the inter-node aggregation phase.
For inter-node aggregation, five components are contributing to its timing. The first two components are flattening the MPI file view into a list of offset-length pairs (calculating my requests) and calculating others’ requests to identify the global aggregators who are responsible for writing the request (calculating other requests). In TAM, only local aggregators make calls for calculating my requests. The time complexity is proportional to the number of locally aggregated non-contiguous I/O requests. Calculating other requests involves a many-to-many inter-node communication from local aggregators to global aggregators. Its timing depends on the number of non-contiguous requests and the number of MPI requests, $p_l \cdot p_g$. For collective write, the third component is to merge-sort the offsets of the locally aggregated non-contiguous I/O requests. The fourth component is the construction of MPI derived datatypes at every global aggregator for sending/receiving data from local aggregators. The time complexities of the third and fourth components are both proportional to the number of I/O requests at local aggregators. The last component is the inter-node communication between local aggregators and global aggregators for data. In short, the execution time of inter-node aggregation is a sum of computation cost that depends on the number of I/O requests at local aggregators and the inter-node communication for metadata and data. Based on our experimental observations, inter-node communication time is the dominant factor.

Inter-node communication performance depends on whether the communication channels are under or over-utilized. A typical trend is shown in Figures 4.5(a)-(d), the timing cost of E3SM F case KNL collective write. Initially, on four nodes, the inter-node aggregation cost decreases as the number of local aggregators increases. However, as the total number of nodes increases from 4 to 256, this trend changes. For example, Figure 4.5
(b) and (c) illustrate a convex curve of inter-node communication time concerning the number of local aggregators per node. Figure 4.5 (d) shows a monotonously increasing trend of inter-node communication as the number of local aggregators per node increases. The inter-node communication has a many-to-many communication pattern between local aggregators and global aggregators. If global aggregators communicate data with a small number of local aggregators, the communication channels are under-utilized at receiving nodes. Thus, the network channels of the nodes where the global aggregators are placed are not saturated up to the case when they communicate with all processes, which explains the observation from Figure 4.5 (a). On the other hand, communication contentions occur if global aggregators exchange data with an enormous number of local aggregators. In Figures 4.5 (d), communication contention starts at assigning more than four local aggregators per node. In the original two-phase I/O, every global aggregator must prepare to communicate metadata and data with all other processes when all-to-many communication occurs. Consequently, as the number of processes increases, the communication contention is expected to become worse at the global aggregators. We should choose a $p_l$ that does not cause over-utilization and under-utilization of communication channels. For example, in Figures 4.5 (b) and (c), $p_l = 256$ is a turning point that yields the lowest inter-node communication cost.

Collective read and write have different turning points and degrees of inter-node contention. For example, from Figures 4.5 (d) and (h), KNL read results for F case has turning point at $p_l = 1K$. The turning point of the KNL write result, on the other hand, occurs at $p_l = 256$. This difference is expected since collective read and write have reversed inter-node communication patterns. In general, we observe the performance degradation
for receiving data from a considerable number of processes is more severe than the conti-
tention for sending data to the same number of processes at global aggregators. We set \( p_l \)
to be a constant 512 by default. However, the optimal choice of \( p_l \) depends both on the
underlying hardware architecture and the degree of contention illustrated in Figures 4.4.
Determining the optimal \( p_l \) based on communication patterns and hardware systems can
be an interesting future study.

4.3.2. BTIO Benchmark

Developed by NASA Advanced Supercomputing Division, BTIO is part of the benchmark
suite NPB-MPI version 2.4 for evaluating the performance of parallel I/O \[48\]. BTIO
uses a block-tridiagonal partitioning I/O pattern over a three-dimensional array. BTIO
requires a square number of MPI processes to run, which divide the cross-sections of a
global 3D array evenly. We set the global array size to be \( 512 \times 512 \times 512 \). The total write
amount is of size \( 8 \times 512^3 \times 5B = 5\text{GiB} \) with five 8-byte double values per variable. Unlike
E3SM-IO, which has a near-constant size of non-contiguous requests, the total number of
non-contiguous requests for BTIO increases along with the number of processes, as shown
in Table 5.1. This property allows us to test TAM for data with a small I/O data size
but a large number of I/O requests.

Figure 4.7 presents the breakdown timings of BTIO with TAM. With 16K processes,
the size of non-contiguous requests is \( 512 \times 40 \times \sqrt[5]{16384} = 1.34 \times 10^9 \). Two-phase I/O
completes data exchange in multiple rounds. Each has a file domain size bounded by the
Lustre stripe. Although the overall data exchange pattern for BTIO is high, as shown
in Figures 4.4 not all processes exchange data with global aggregators per two-phase
Figure 4.5. Timing breakdown for E3SM F case with different number of local aggregators. The right-most bar in every subfigure is the two-phase I/O timing ($p = p_g$). (a)-(d): KNL write. (e)-(h): Haswell write. (i)-(l): KNL read. (m)-(p): Haswell read.

I/O round. Therefore, data exchange of BTIO in two-phase I/O with a large number of processes does not show the same degree of communication contention compared with the metadata exchange of BTIO. Consequently, the exchange of metadata is the bottleneck of two-phase I/O for BTIO due to a high degree of communication contention, though the number of bytes transferred in the metadata exchange is less than the data
Figure 4.6. Timing breakdown for E3SM G case with different number of local aggregators. The right-most bar in every subfigure is the two-phase I/O timing \((p = p_g)\). (a)-(d): KNL write. (e)-(h): Haswell write. (i)-(l): KNL read. (m)-(p): Haswell read.

exchange. Taking advantage of the spatial locality property of BTIO, TAM reduces the cost of metadata exchange at the inter-node aggregation stage by reducing the number of offset/length pairs transmitted with I/O request coalescing. The number of the coalesced requests is \( \left( \frac{1}{2} \right)^{k_l} \) of the original request size. Furthermore, TAM can reduce the degree
of contention for metadata exchange, since only local aggregators and global aggregators perform inter-node communication.

Figure 4.7. Timing breakdown for BTIO with different number of local aggregators. The right-most bar in every subfigure is the two-phase I/O timing (p = p_g). (a)-(d): KNL write. (e)-(h): Haswell write. (i)-(l): KNL read. (m)-(p): Haswell read.
4.3.3. S3D-IO Case Study

S3D-IO case study is the I/O kernel of a parallel turbulent combustion application, named S3D, developed at Sandia National Laboratories [47]. The I/O kernel is a checkpoint of three-dimensional arrays, corresponding to a 3D Cartesian mesh. Four variables are written at every checkpoint: mass, velocity, pressure, and temperature. Pressure and temperature are 3D arrays, while mass and velocity are 4D arrays with the fourth dimension sizes 11 and 3. Processes partition the first three dimensions of every variable in a block-block-block fashion. Thus, a process owns 16 non-contiguous 3D subarrays. We set the global 3D array size to $800 \times 800 \times 800$, the same as [52]. This setting results in a total write amount of size $8 \times (11 + 3 + 1 + 1) \times 800^3 B = 61$GiB with 8-byte double type per value.

The block-block-block partitioning of the global 3D array is expected to produce non-contiguous requests that mostly can be coalesced at the local aggregators, a similar effect observed in the BTIO benchmark. The number of non-contiguous I/O requests after coalescing at intra-node aggregation phase is $\left(\frac{1}{2}\right)^{\frac{n}{p_i}}$ of the original one. The total number of non-contiguous requests for S3D-IO is $800^2 x$. $x$ is the number of processes in the first dimension. Nevertheless, inter-node aggregation using two-phase I/O is still a bottleneck of S3D-IO as the number of processes increase.

As shown in Figure 4.8, TAM does not show a significant improvement for reading S3D-IO collectively. With our Lustre setting for collective write in ROMIO, a process has I/O requests that intersect with the file domains of all global aggregators. Therefore, the communication pattern is all processes to all global aggregators when two-phase I/O is used. Collective read, on the other hand, does not have such a communication pattern.
File domains of global aggregators in ROMIO for collective read are contiguous blocks that align with lock boundaries. Thus, a process only needs to communicate with a subset of global aggregators with file domains that overlap with the 3D subarrays owned by the process. Hence the communication pattern of two-phase I/O collective read does not cause a severe degree of contention compared with collective write. In Figure 4.4 (d), we can confirm that the degree of contention of two-phase I/O collective read is approximately 1000, which is ten times less than that of two-phase I/O collective write with 16K processes. As a result, TAM does not improve the performance of the two-phase I/O collective read in the same way as collective write for S3D-IO.

4.3.4. Limitation of TAM

The benchmarks and application kernels in our experiments, though differ in detailed I/O pattern, have high numbers of non-contiguous I/O requests. Thus, global aggregators receive data that originated from all processes at the inter-node aggregation phase. However, TAM is designed for I/O patterns with a large volume of non-contiguous requests that cause a high communication contention for global aggregators and all processes. If inter-node communication contention is not severe, the intra-node aggregation phase becomes redundant. In this case, MPI collective I/O implementation can adopt the traditional two-phase I/O strategy because the intra-node aggregation stage is redundant given such a scenario.
Figure 4.8. Timing breakdown for S3D-IO with different number of local aggregators. The right-most bar in every subfigure is the two-phase I/O timing \((p = p_g)\). (a)-(d): KNL write. (e)-(h): Haswell write. (i)-(l): KNL read. (m)-(p): Haswell read.

4.4. Conclusion

In this chapter, we have demonstrated the communication cost of the two-phase I/O strategy can become the performance bottleneck of the MPI collective I/O. Our proposed TAM is designed to tackle the problem from the angle of reducing communication contention at global aggregators. Adding an intra-node aggregation effectively reduces the
communication contention at the global I/O aggregators and thus allows collective I/O to scale up with more MPI processes. Experiments show that TAM works well for applications that make a large number of non-contiguous I/O requests from every process. As the HPC community is entering the exascale era, keeping MPI-IO implementation scalable to higher numbers of MPI processes has become increasingly important.

The design concept of TAM does not limit the first phase aggregation to be within a compute node. We have illustrated that the cost of intra-node aggregation stage in strong-scaling vanishes as the number of processes scales up. When the total number of nodes scales much more massive than our experimental settings, the first phase aggregation across multiple nodes is expected to have a small cost as well. Inter-node aggregation to global aggregators, on the other hand, suffers from communication contention even if there is one local aggregator per node, so the end-to-end time can be lower if we reduce the number of local aggregators by performing the first phase aggregation across a small number of compute nodes.

There are other opportunities to improve the collective I/O performance further. One possibility is to overlap the communication with the I/O as the pipelining approaches proposed in [55, 56]. Moreover, the intra-node aggregation in TAM is motivated by the scenario when the number of MPI processes is significant, resulting in a high number of messages sending to a small amount of I/O aggregators. However, when the number of MPI processes is not significant enough, such as assigning one communication process per node in the MPIOpenMP programming model, TAM is less effective compared with the applications that assign a large number of MPI processes per node. Nevertheless, as the number of nodes scales up in exascale computing, the first phase aggregation across
multiple nodes can still improve the communication performance. Future work includes the extension of a TAM to consider MPI processes allocated at compute nodes that are physically near with each other sharing the same communication hardware, such as routers in the identical cabins.
CHAPTER 5

Improving all-to-many personalized communication in
two-phase I/O

The Message Passing Interface (MPI) standard defines a set of programming interfaces for interprocess communication [19]. MPI-IO, a submodule of the MPI standard, provides interfaces for parallel shared-file access. I/O operations in scientific applications such as [39, 40, 41, 42, 43] are implemented with MPI-IO libraries. MPI processes can collectively open a file to perform I/O operations. Implementations of the collective I/O functions can coordinate processes’ operations to achieve better end-to-end performance compared with independent I/O. A well-known collective I/O design is the two-phase I/O strategy [5], which has become the implementation backbone for collective I/O in almost all MPI libraries.

Two-phase I/O consists of a communication phase and an I/O phase. I/O aggregators, a subset of processes, are assigned with file access regions, denoted as file domains. The aggregators gather I/O requests and data, based on their file domain, from the rest of the processes in the communication phase and carry out I/O operations with file systems in the I/O phase. In general, the communication between nonaggregators and I/O aggregators is many-to-many. ROMIO, the implementation of the MPI-IO functions used most frequently in high-performance computing (HPC) and provided by vendors as part of
their MPI implementation \cite{46}, implements two-phase I/O in multiple rounds of communication and I/O with asynchronous MPI functions. When the number of noncontiguous I/O requests is significant, the communication phase of two-phase I/O exhibits an all-to-many personalized communication pattern, where the “many” group refers to the I/O aggregators. The communication cost may exceed the I/O cost for large parallel jobs when the number of noncontiguous I/O requests is significant because of high communication contention and straggler effects \cite{44, 45} caused by multiple rounds of all-to-many communication, especially as the number of processes increases.

In this chapter, we explore all-to-many personalized communication algorithms: pairwise, spread-out, and the two-layer aggregator method (TAM) for improving two-phase I/O communication performance. We extend the spread-out algorithm adopted by the implementations of personalized all-to-all in major MPI production libraries to adapt to the all-to-many communication pattern by reducing the straggler effect resulting from an unbalanced communication workload. Throttling techniques are additionally applied to the asynchronous MPI point-to-point implementation to reduce communication contention. For evaluation, we replace the metadata and communication kernels of two-phase I/O in ROMIO with different all-to-many personalized communication algorithms. Experiments are conducted on two different supercomputing systems: Cori, a Cray XC40 supercomputer with Intel KNL processors and the Lustre file system, and Summit IBM Power System AC922 nodes equipped with IBM POWER9 CPUs and IBM GPFS. We use three benchmark programs: E3SM-IO \cite{42}, FLASH/IO \cite{47}, and BTIO \cite{48}. These three I/O benchmarks have different metadata and data communication patterns.
Applying communication throttling in a two-phase I/O communication phase with an all-to-many pattern can improve the communication performance by up to 10 times, compared with the current implementation of ROMIO two-phase I/O. Our balancing strategy for the spread-out algorithm improves communication performance by over 30%. The two-layer aggregation method [11], an implicit throttling approach that performs intranode and internode communications separately, works best for I/O patterns that require all-to-many communication because internode communication contentions are significantly reduced. However, we expect that applying the proposed spread-out reordering method, along with the throttling technique, to the communication kernels of existing communication designs such as TAM can further reduce internode communication contentions for the future exascale applications running on a considerable number of compute nodes. When applications do not exhibit significant many-to-many communication contentions, the current ROMIO implementation for all-to-many communications can outperform the rest of the communication kernels. Thus, the communication kernel of two-phase I/O should be dynamically chosen based on application I/O patterns.

5.1. Background

We define the terminology of personalized communication. Let $P$ be the array of all processes ranked from 0 to $p - 1$. A sorted array of sender ranks $S$ transfer their data with arbitrary length to a sorted array of receiver ranks $R$. Let $m_{i,j}$ be the message sent from rank $i$ to rank $j$. $m_{i,j}$ is empty if either $i \notin S$ or $j \notin R$. We refer to $m_{0,j}, ..., m_{p-1,j}$ as the local messages at rank $j$. We assume that point-to-point communication operations (send/receive) between any two arbitrary processes are available. An algorithm
schedules the send and receive operations to accomplish the data transfer. All-to-many communication has $S = P$. All processes are sending personalized messages to a subset of processes.

Our point-to-point communication model has two components: the startup term $t_s$ and the per-byte transfer (inverse bandwidth) time $t_w$. The startup and transfer terms are classical communication models, as mentioned in [25]. Sending $k$ bytes from a sender to a receiver, including self-send, has a communication cost of $t_s + kt_w$. For simplicity of analysis, we assume a textbook one-port communication constraint, which means the maximum number of either concurrent send or receive operations is 1. We can, however, extend our results to an $n$-port communication constraint by discussing send and receive operations in blocks of processes. We denote the term communication contention as the additional time cost resulting from excess concurrent point-to-point communication requests at a process.

5.1.1. Motivation

Our research is motivated by the communication pattern of two-phase I/O [5]. Two-phase I/O consists of communication and I/O phases. A few MPI processes called I/O aggregators are selected as I/O proxies for all processes. Only I/O aggregators exchange data with the file systems. Other processes communicate with I/O aggregators to complete their I/O requests. For collective write, all processes send their I/O requests and write data to I/O aggregators. I/O aggregators then exchange data with file servers according to the gathered I/O requests. Parallel I/O for large files in the exascale computing era can cause memory overflow at I/O aggregators, especially when the number of I/O aggregators
is much smaller than the number of processes in the computation. ROMIO [46], a widely used MPI-IO implementation adopted by major production libraries, avoids the memory overflow by implementing two-phase I/O in multiple rounds. The I/O driver processes file domains with limited sizes in each round. Thus, the communication of two-phase I/O consists of multiple rounds of many-to-many communication. With a large number of noncontiguous I/O requests, the communication pattern becomes all-to-many. Our recent study of the I/O performance of the E3SM [42] model has shown that such all-to-many communication dominates the overall performance. The performance degradation is more severe as the number of processes scales.

5.1.2. Related Work

Personalized communication is implemented by the MPI_Alltoallv and MPI_Alltoallw functions. Many works of literature have focused on all-to-all communication patterns \( S = R = P \). We summarize the mainstream algorithms that are implemented by production libraries such as MPICH [6] and OpenMPI [8].

Bruck’s algorithm [28] is an efficient algorithm for transmitting small messages in personalized communication. It has a performance advantage over the traditional recursive-doubling approach in practice [29]. This algorithm finishes in \( \log(p) \) steps for \( p \) that is a power of two. If \( p \) is not a power of two, some slight overhead occurs. At step \( i \), rank \( x \) sends nonlocal messages gathered from previous rounds to \( x + 2^{i-1} \mod p \). When \( S = R = P \) and all receivers have the same local data size \( k \), the overall performance is \( \log(p) t_s + \frac{k}{2} \left( \log(p) + 2^{\log(p)} \right) t_w \). This performance is desirable when \( k \) is small.
For parallel I/O problems, \( k \) is usually a large value, so the communication kernel should adopt algorithms that optimize the \( t_w \) term. This class of algorithms is designed based on the principle that processes are always busy receiving their personalized messages. The pairwise algorithm is designed for large data exchange when \( p \) is a power of two. The algorithm finishes in \( p - 1 \) steps. At step \( i \), rank \( j \) exchanges data with rank \( i \oplus j \). Another algorithm, called the spread-out algorithm, can also handle the case when \( p \) is not a power of two; rank \( j \) exchanges data with rank \( j - i + p \mod p \) at step \( i \). When \( S = R = P \) and all receivers have the same local data size \( k \), both the pairwise and spread-out algorithms have communication cost of \( (p - 1)t_s + kt_w \). These two methods have advantages over Bruck’s algorithm for a large \( k \).

Two-phase transmission for all-to-all exchange is an efficient strategy for reducing the total number of internode communications. Träff and Rougier have proposed a local node gathering strategy for MPI\_Alltoall data using communicator splitting [74]. SLOAV [75] is an improvement over the Bruck’s algorithm with two-phase message transmission that handles data with variable size for MPI\_Alltoallv and MPI\_Alltoallw. The two-layer aggregation method (TAM) applies the intranode gathering principle to MPI two-phase communication [11]. A subset of processes, denoted as local aggregators, gathers I/O requests from all processes. Then, local aggregators and I/O aggregators carry out two-phase I/O. With the help of intranode aggregation, internode communication contentions are significantly reduced because a smaller number of processes participate in data exchange among compute nodes.
5.1.3. MPI Asynchronous Communication

Modern supercomputers have multiple intranode and internode communication channels. Applications can overlap computation and communication for better performance. Therefore, implementing personalized communication patterns with asynchronous MPI functions may utilize hardware resources more efficiently. To implement algorithms for all-to-all with asynchronous MPI communication functions, the MPICH library posts all MPI_Isend and MPI_Irecv requests in the order defined by the algorithms. A subsequent MPI_Waitall for all requests is made in the end.

There are three major MPI asynchronous communication implementations. One method is a thread-based approach, which is adopted by the current branches of MPICH and MVAPICH [7]. MPI_Init creates a communication thread for every process that continuously handles point-to-point communication requests posted from the master thread. When a large number of asynchronous send requests are posted to a receiver, however, the mutex-based approach that most MPI implementations adopt today for thread safety can be a source of contention [76]. Another method is to assign dedicated “ghost” processes for processing asynchronously communication requests offloaded from user processes [77]. The “ghost” process approach offers a more flexible core usage compared with the thread-based method, but multiple processes can still race for shared resources. The third approach is to utilize hardware interrupts as in the Blue Gene/Q [78, 79, 80], and Cray systems.
5.2. Design

In this section, we propose an algorithm for all-to-many personalized communication that evenly spreads out the communication workload. The new algorithm is a generalization of the spread-out algorithm used for MPI_Alltoallv and MPI_Alltoallw in MPICH. We refer to this improvement as the balanced spread-out algorithm. We compare the spread-out and the balanced spread-out algorithm from the point of view of communication straggler effects. A throttling technique for both algorithms is also discussed for reducing communication contention.

5.2.1. The Balanced Spread-Out Algorithm

Algorithm 10: The spread-out algorithm.

1. rank ← local process rank ID
2. if rank ∈ R then
3.   for i ∈ [0, ... p − 1] do
4.     src ← (rank + i) mod p
5.     rank receives m_{src,rank} from src
6.   end
7. end

Algorithm 10 is the traditional spread-out algorithm adopted by major production libraries. The algorithm depicts the receiving order of local messages at an arbitrary receiver. Since R is not necessarily equal to P, non-receivers do not enter the condition at Line 2. When the communication pattern has a few receivers with lower ranks that receive from all processes, non-receivers can cause a potential contention at the beginning, as illustrated in Figure 5.1 (a). All processes with ranks larger than the largest receiver
Figure 5.1. This figure illustrates the communication workload of all processes in the spread-out and balanced spread-out algorithms in the first round. The red circles represent processes that are receivers. The blue circles represent the rest of the processes. An arrow between two circles represent a point-to-point communication. (a) The communication workload is imbalanced in the spread-out algorithm. Rank 3 receives data from 3–7 in the first round. (b) The communication workload is balanced in the balanced spread-out algorithm with virtual rank list \{0, 4, 1, 5, 2, 6, 3, 7\}.

**Algorithm 11:** The balanced spread-out algorithm.

```plaintext
1 rank ← local process rank ID
2 if rank ∈ R then
3     j ← local receiver index in R
4     rank_list ← Any ordering of process ranks
5     q ← p mod |R|
6     if j < q then
7         rindex ← ⌈p/|R|⌉j
8     else
9         rindex ← ⌈p/|R|⌉q + (j − q) ⌊p/|R|⌋
10    end
11 for i ∈ [0, ..., p − 1] do
12     src ← rank_list[(rindex + i) mod p]
13     rank receives m_{src,rank} from src
14 end
15 end
```
rank $r_{\text{max}} \in R : r \leq r_{\text{max}} \forall r \in R$ among receivers directly send their messages to $r_{\text{max}}$. For large $p - r_{\text{max}}$, contention occurs due to the straggler effect, which is explained later.

Algorithm 11 resolves the contention scenario of Algorithm 10 by providing a load-balancing solution. To generalize the solution, we construct a rank_list owned by all processes. The array rank_list can be a simple ordering of processes from 0 to $p - 1$. It can also be any ordering of $P$, such as round-robin, based on compute nodes to adapt to different types of rank assignments. The algorithm divides $p$ processes in the rank_list into $|R|$ groups. If $|R|$ does not divide $p$, $p \mod |R|$ groups have size of $\lceil \frac{p}{|R|} \rceil$, and the rest of the groups have size $\lfloor \frac{p}{|R|} \rfloor$. Receivers and the process groups form a bijection mapping. The first process in the group that corresponds to an arbitrary receiver is rank_list[rank_index]. The receiver posts its requests from senders in order of rank_list[rank_index],.., rank_list[rank_index + $p \mod p$]. Figure 5.1 (b) is an example of the initial communication step of the proposed algorithm. We use a rank_list setting \{0, 4, 1, 5, 2, 6, 3, 7\} as an example. A receiver posts receive operations from two senders at a time.

5.2.2. Straggler Effect in Communication

A communication straggler refers to the communication time difference between a process and the average. Straggler effects are the idle time that a process waits for other processes to finish because of dependency on available data or operation orders. The original spread-out algorithm suffers from straggler effects when applied to all-to-many case, as shown in Figure 5.1 (a). With a one-port communication constraint, process rank 2 starts to receive data from process rank 4 after process rank 4 has finished the sending operation
to rank 3. In general, any receive operation at ranks 0–2 has to wait for all the rest of the senders to finish their previous communications.

We assume that the latency of a point-to-point data transfer has a white noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$, where $\sigma^2$ is the variance of the noise term. Let $\mu$ be the mean communication time. For small data size, the ratio $\mu$ to $\sigma$ is large. The message size exchanged per round does not exceed the collective buffer size, so the noise term is not negligible even for very large files. Straggler effects can be formulated as the following. When a receiver attempts to receive from a sender, there are two cases. The first case is that the sender is ready to send, and the second case is that the sender has not finished its previous send operations. When a process finishes receiving its first message and becomes ready to receive its second message, both cases can occur with an equal chance. The first case does not cause delay, but the second case causes a delay in the receive operation at the receiver. The expectation of the straggler effect $X_1$ after the first step is shown Equation 5.2. We use $\phi$ and $\Phi$ to denote the probability and cumulative functions of standard normal distribution.

\begin{align}
E(X_1) &= \frac{1}{2} \times 0 + \int_{0}^{\infty} x\phi(x) dx \\
&= \sigma \sqrt{\frac{1}{2\pi}}
\end{align}

For the spread-out example, rank 0 has straggler effects, denoted by random variable $X_i$ for each of sender ranks $i \in \{2, ..., p-1\}$. Straggler effects from senders with higher ranks are dependent on those from lower ranks. Nevertheless, by treating them independently,
we can approximate the expectation of total delay in Equation 5.4.

\[ E \left( \sum_{i=1}^{p-1} X_i \right) \approx \sum_{i=1}^{p-1} \sigma \sqrt{\frac{1}{2\pi}} \]

\[ = (p - 1) \sigma \sqrt{\frac{1}{2\pi}} \] (5.4)

The total delay scales as the number of processes increases. On the other hand, for Algorithm 11, the expectation of total straggler effects at any receivers is close to zero as long as \(|R| \leq \frac{p}{2}\). For example, in Figure 5.1, receivers send to themselves at the beginning. Rank 0 starts to receive from rank 4 after it finishes its self-send. It is unlikely that rank 1 has not finished its self-send by the time when rank 0 finishes receiving data from rank 4. We use \(X'_1\) to denote the random variable for the straggler effect from the second communication in Algorithm 11. If we assume \(\sigma = \frac{1}{8}\mu\), a reasonably large variance, the probability of the first straggler effect happening \((X'_1 > 0)\) when \(|R| = \frac{p}{2}\) is Equation 5.5 for Algorithm 11 which is much less than \(\frac{1}{2}\) for Algorithm 10. In addition, regardless of the value of \(\sigma\), this probability is always less than \(\frac{1}{2}\).

\[ 1 - \Phi \left( \frac{2 - 1}{\sqrt{\frac{1+2}{8}}} \right) = 1 - \Phi \left( \frac{8}{\sqrt{3}} \right) \]

\[ < 2 \times 10^{-6} \] (5.6)

Moreover, this probability reduces as \(|R|\) or \(\sigma\) decrease. Therefore, the total straggler effects can be considered as negligible. When \(p > |R| > \frac{p}{2}, p \mod |R|\) out of \(|R|\) receivers have the same straggler effects as the spread-out algorithm. When \(|R| = p\), the proposed algorithm has identical straggler effects as the spread-out algorithm. To sum up, the
balanced spread-out algorithm is expected to have less performance degradation caused by straggler effects compared with the spread-out algorithm.

5.3. Implementation

The pairwise algorithm is a store-and-forward design, so adjacent steps have data dependency. Hence this algorithm is usually implemented with MPI blocking functions. Spread-out algorithms, on the other hand, can be implemented with asynchronous MPI functions because there is no data dependency for the personalized communication pattern. Since our proposed algorithm is a generalization of the spread-out algorithm, it can also be implemented by using MPI asynchronous functions. We present two implementation methods that can improve the performance of the spread-out and balanced spread-out algorithms.

5.3.1. Replacing Isend with Issend

Both MPI_Isend and MPI_Issend can be used for posting data transfer requests from senders to receivers. The difference between these two, according to the standard, is the behavior when MPI_Wait is called for their requests.

MPI_Wait may return the request from MPI_Isend when the send buffer is ready to be modified. The request from MPI_Issend cannot be returned from MPI_Wait until the MPI_Irecv at the remote side has started receiving data from the corresponding MPI_Issend. For a single point-to-point communication, the end-to-end latency difference between using MPI_Isend and MPI_Issend is not significant. The difference is critical
for the performance of all-to-many communication, however, especially when there are multiple rounds.

For small messages, some MPI implementations cache the data of `MPI_Isend` requests. `MPI_Wait` immediately returns after the message copy from the send buffer. If the all-to-many operation is executed many times in a loop with small message sizes, senders can accumulate a large number of data buffers for `MPI_Isend` requests because the program is not necessarily blocked by `MPI_Waitall` after data memory copy. Receivers, on the other hand, can escape from the `MPI_Waitall` blocking only after receiving all data in the `MPI_Irecv` requests in the current iteration. When a large number of requests accumulate at the senders, the communication performance at senders degrades dramatically, resulting in poor performance. We refer to this scenario as request contention at senders.

To avoid request contention at senders, we could call `MPI_BARRIER` at the end of each iteration. The barrier would force synchronization of senders and receivers, so senders could not post requests in the next iteration until all requests in the current iteration were finished. Performing a global barrier per round, however, can significantly degrade performance. Alternatively, replacing `MPI_Isend` with `MPI_Issend` can resolve this issue. `MPI_Issend` forces receivers to acknowledge requests by calling `MPI_Irecv` at the remote side. This approach allows us to force senders to block until receivers have at least entered the sender’s current round. We adopt this `MPI_Issend` strategy, avoiding the need to use a barrier to synchronize all processes and the associated delays.
5.3.2. Throttling for Asynchronous Communication

Oversubscription caused by a large number of MPI requests can be a performance bottleneck \cite{44}. Asynchronous implementation of all-to-many algorithms often requires receivers to handle multiple MPI\_Irecv requests simultaneously. When MPI\_Waitall is called, receivers test whether data for individual MPI\_Irecv have been received. The checking operations can be considered as a light-weight version of MPI\_Test. A long list of requests can result in longer checking time. Furthermore, the orders of acknowledgment for senders and messages received for receivers are not constrained. Hence the worst-case checking complexity is proportional to the number of MPI requests in the queue. In addition, when a large number of MPI requests are passed to lower-level communication libraries, repeated checking can also happen at those libraries. Therefore, posting an extensive array of MPI requests is not recommended.

Throttling at a collective communication algorithm level is a useful technique to avoid performance degradation caused by oversubscription. The current MPI\_Alltoallv and MPI\_Alltoallw implementation of Algorithm \cite{10} by MPICH has adopted this strategy. The throttling strategy separates MPI\_Irecv requests into small groups with a maximum size of comm\_size. For example, the MPICH master branch has a default size of 32, and Cray-mpich has comm\_size (MPICH\_ALLTOALLV\_THROTTLE) equal to 8 by default. MPI\_Waitall is called for every group of MPI\_Irecv requests sequentially at receivers. The total number of concurrent communication is bounded by comm\_size. Similarly, TAM \cite{11} implicitly throttles for internode communication since only local and I/O aggregators perform internode communication.
5.4. Experimental Results

Table 5.1. Datasets used in Chapter 5. The second column shows the total number of noncontiguous requests. For the E3SM F benchmark, the noncontiguous requests are collected from production runs using 21,600 processes. We present the strong-scaling results, so the noncontiguous requests partitioned among all $p$ processes are used in our experiments. BTIO has more significant numbers of noncontiguous requests, and the number increases as the number of processes $p$.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Noncontiguous Requests</th>
<th>Write Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>E3SM F</td>
<td>$1.36 \times 10^9$</td>
<td>14GiB</td>
</tr>
<tr>
<td>FLASH/IO</td>
<td>$24p$</td>
<td>$8 \times 80^4p$GiB</td>
</tr>
<tr>
<td>BTIO</td>
<td>$1024^2p$</td>
<td>40GiB</td>
</tr>
</tbody>
</table>

We conduct evaluations for MPI collective write in ROMIO by replacing its metadata and data communication kernels with five all-to-many communication algorithms.

Our implementations are based on the Lustre and GPFS drivers in the MPICH-3.3 release. The default ROMIO implements metadata and data communications by posting $\text{MPI}_{\text{Isend}}$ and $\text{MPI}_{\text{Irecv}}$ in ascending rank order followed by $\text{MPI}_{\text{Waitall}}$. We explore four communication algorithms for metadata and data exchange: (1) the current ROMIO implementation that posts all requests in ascending rank order with $\text{Issend}$, (2) the spread-out algorithm, (3) the balanced spread-out algorithm, (4) the pairwise algorithm, and (5) TAM. Figure 5.3 illustrates the communication patterns of these algorithms. For the balanced spread-out algorithm, we use the default rank list ordering $\{0, ..., p - 1\}$. For TAM, we use the default setting of 512 for the number of local aggregators. If the number of processes is less than 512, TAM is equivalent to the default ROMIO two-phase I/O. All algorithms except the pairwise algorithm are implemented with the MPI asynchronous
functions \texttt{MPI\_Issend} and \texttt{MPI\_Irecv}. The pairwise algorithm is implemented by using \texttt{MPI\_SendRecv}.

We perform all evaluations on two supercomputing systems. One is Cori, a Cray XC40 supercomputer with Intel KNL processors and Lustre file system at the National Energy Research Scientific Computing Center (NERSC). Each Cori KNL node contains one CPU with 68 CPU cores. Compute nodes on Cori are connected with a dragonfly topology. We set the Lustre stripe count to be 64 and stripe size to be 1 MiB. In our experiments, we allocate 64 processes per Cori KNL node. The other supercomputer is Summit, at the Oak Ridge Leadership Computing Facility (OLCF), with IBM Power System AC922 nodes equipped with IBM POWER9 CPUs and IBM GPFS. Each Summit node has two CPUs with 21 CPU cores per socket. The interconnect topology on Summit is a non-blocking fat-tree with a Mellanox EDR 100G InfiniBand connection. The default MPI compilers are Cray-mpich on Cori and IBM Spectrum on Summit. Both MPI implementations are not open-sourced. Nevertheless, their end-to-end timing results can serve as baselines.

The Lustre driver assigns the number of I/O aggregators equal to the stripe count in a round-robin fashion across nodes by default. We present the performance using default aggregator placement that assigns ranks $64i \forall 0 \leq i < 64$ as I/O aggregators. This setting allows us to evaluate the advantage of the balanced spread-out algorithm over the original spread-out algorithm for all-to-many communication patterns. When the environmental variable \texttt{AGGREGATOR\_PLACEMENT\_STRIDE} is set to -1 on Cori, the placements of I/O aggregators are fully spread out across nodes, so ranks $\left\lfloor \frac{n}{64} \right\rfloor \forall 0 \leq i < 64$ become I/O aggregators on $n$ nodes for $p > n$. Therefore, the balanced spread-out algorithm becomes identical to the original spread-out algorithm when this hint is set. Moreover,
the spread-out algorithm and the evenly spread-out algorithm are identical for experiments on Summit and 64-node experiments on Cori by default, so we do not list the performance of these two algorithms separately. On the other hand, GPFS drivers select the process with the lowest rank as an I/O aggregator for every node by default.

Figure 5.2. Two-phase I/O implementation for the Lustre and GPFS drivers. Different colors represent file domains owned by different I/O aggregators. We label the file domains handled by different rounds of two-phase I/O. (a) Lustre driver. (b) GPFS driver.

ROMIO implements Lustre and GPFS drivers in multiple two-phase I/O rounds with different access patterns. As illustrated in Figure 5.2 (a), the file views of an I/O aggregator in the Lustre driver are interleaved contiguous regions of stripe size. For each round of two-phase I/O, the Lustre driver processes one stripe, a contiguous block of data with a size equal to stripe size multiplied by stripe count. Our Lustre setting results at 1MiB (stripe size) collective buffer size.

The GPFS driver evenly divides the entire file domain into the number of I/O aggregators continuous regions. For each round of two-phase I/O, each I/O aggregator handles
Figure 5.3. These figures illustrate the differences in the implementations we used for evaluation. We use the blue circles to denote nonaggregators. The red circles are I/O aggregators, and the green circles are local aggregators in TAM language. In this example, we have 2 I/O aggregators, 4 local aggregators, and 8 processes. We label their ranks next to them. The arrows indicate point-to-point communications. (a) Default ROMIO implementation. All communication requests are posted simultaneously. (b) The balanced spread-out algorithm with throttling limit 4. The difference between the original and balanced spread-out algorithm has been illustrated in Figure 5.1. With throttling, an I/O aggregator receives data from a batch of 4 processes at a time. After receiving from a batch is finished, it receives from the next batch. This figure shows the first round of communication. (c) Pairwise algorithm. An I/O aggregator receives data from 1 process at a time. After receiving data from rank $x$, it receives data from $x + 1 \mod p$. This figure shows the first round of communication. (d) Two-layer aggregation method (TAM). An arbitrary process sends data to 1 local aggregator. Then all local aggregators exchange data with all I/O aggregators.

a contiguous region of I/O requests within its file domain, as illustrated in Figure 5.2 (b). Furthermore, the default collective buffer size, the maximum number of bytes processed by an I/O aggregator per round, is 16 MiB in the GPFS driver. Therefore, the number of two-phase I/O rounds performed by the Lustre driver is 16 times that of the two-phase I/O rounds executed by the GPFS driver. Consequently, the numbers of two-phase I/O rounds in Lustre and GPFS drivers are different.

Table 5.1 summarizes the datasets used in our experiments. The E3SM-IO F case has an all-to-many communication pattern for both metadata and data communication.
FLASH/IO has many-to-many instead of all-to-many communication patterns, for both metadata and data. We show that algorithms designed for all-to-many patterns may no longer have advantages over the traditional approach. For BTIO, the metadata communication is all-to-many, and data communication is many-to-many. We can apply different communication strategies that adapt to these two patterns. The conclusion of this case study is that communication algorithms should be selected based on the I/O pattern.

5.4.1. E3SM-IO Benchmark

Figure 5.4. (a)–(d) illustrate the communication time of the E3SM-IO F case using the balanced spread-out algorithm with different throttling threshold. The figure titles identify the dataset, number of nodes, and computing system. The data communication pattern in E3SM-IO is all-to-many. The x-axis depicts the upper bound of the number of concurrent MPI_Irecv in any MPI_Waitall call.

E3SM [42] is an exascale Earth system modeling program for simulating atmosphere, land, and ocean behavior in high resolution. Its I/O module is implemented with PIO library [70], which is built on top of PnetCDF [71]. Checkpointing is implemented by using nonblocking PnetCDF APIs. The PnetCDF library is a high-level parallel I/O library popular in the climate research community; the library is built on top of MPI-IO. Sending nonblocking requests are flushed by aggregating the request data and combining the MPI file views, followed by a single call to the MPI collective write function. The cost
of posting the nonblocking APIs is negligible, so the end-to-end performance is almost equivalent to the timing of the collective I/O calls inside the PnetCDF flush API.

We evaluated a particular decomposition used in E3SM production runs, namely, F case [42]. The I/O kernel of E3SM has been extracted for I/O study [72]. F case has atmosphere, land, and runoff model components. The data access pattern in the F case consists of 1.36 billion noncontiguous write requests and a total write amount of 14 GiB. For all experiments in our evaluation, the E3SM-IO F case creates an all-to-many communication pattern for both metadata and data transfer from all processes to I/O aggregators.

To present strong-scaling results, we distribute all I/O requests from production runs evenly across all processes. In other words, the entire offset-length pairs from a process in the production run are assigned to a process. The noncontiguous I/O requests in E3SM-IO do not have a subarray pattern, and adjacent file offsets do not have a regular interval. Thus, the file access patterns of E3SM-IO differ from BTIO.

The end-to-end timing consists of computation, communication, and I/O components. Computation cost is dominated by request offset computation and the heap merge sort for I/O requests at I/O aggregators. I/O cost, by its name, is the cost for writing data gathered at I/O aggregators to file servers. Our focus is to improve the communication cost, so we put the computation and I/O cost into the “other” category.

Figure 5.4 presents the timing results of the E3SM-IO F case using the balanced spread-out algorithm with different throttling thresholds. The purpose of these experiments is to demonstrate that throttling is important for large-scale all-to-many communications. Figures 5.4 (a) and (b) show results on Cori 64 and 256 KNL nodes, respectively.
As the throttling threshold increases, the communication cost increases sharply. Summit 128-node results in Figure 5.4 (c), on the other hand, do not have such observable performance degradation. Nevertheless, on 512 nodes, the contention is significant as shown in Figure 5.4 (d). Both computing systems suffer from oversubscriptions of communication resources caused by an excessive number of MPI requests as the number of processes scales. For the rest of the evaluations, we use a default throttling threshold 32, which is also adopted by MPICH MPI_Alltoallw, for both the original and balanced spread-out algorithms. Tuning this threshold can be important on certain systems, but it is not the focus of this chapter.

Figures 5.5 (a) and (b) illustrate the E3SM-IO F case performance results running on Cori KNL. Both the spread-out algorithms have less metadata and data communication time compared with the other algorithms, which benefits from the communication throttling. Communication costs can be further divided into two parts. The first part is the metadata transfer from all processes to I/O aggregators. The second part is the data transmission from all processes to I/O aggregators based on metadata information. To avoid memory overflow for large files, ROMIO implements two-phase I/O by splitting the communication and I/O operations into multiple rounds. I/O aggregators perform two-phase I/O for requests that lie in a file domain with limited size per round. Thus, there are many consecutive all-to-many communications in the ROMIO implementation. For the Lustre file systems, the file domain has a maximum size of Lustre stripe. For GPFS, the maximum file domain size is the collective buffer size (16 MiB by default) multiplied by the number of I/O aggregators. Furthermore, the proposed evenly spread-out algorithm outperforms the original spread-out algorithm used by MPI_Alltoallw, as shown
Figure 5.5. (a)–(l) are breakdown timings of different communication kernels on Cori and Summit. The figure titles indicate the dataset, number of nodes, and computing system. Cray-mpich and IBM Spectrum MPI libraries are not open-sourced, so we present their end-to-end time only. “ROMIO default” means that the current version of ROMIO is used. “Hybrid” means that we use the ROMIO default for metadata exchange and the balanced spread-out algorithm for data exchange. (a) E3SM-IO F case on 64 Cori KNL nodes. (b) E3SM-IO F case on 256 Cori KNL nodes. (c) E3SM-IO F case on 128 Summit nodes. (d) E3SM-IO F case on 512 Summit nodes. (e) FLASH/IO on 64 Cori KNL nodes. (f) FLASH/IO on 256 Cori KNL nodes. (g) FLASH/IO on 128 Summit nodes. (h) FLASH/IO on 512 Summit nodes. (i) BTIO on 64 Cori KNL nodes. (j) BTIO on 256 Cori KNL nodes. (k) BTIO on 128 Summit nodes. (l) BTIO on 512 Summit nodes.

in Figure 5.5 (b) for 256-node experiments. Thus, avoiding the straggler effect discussed previously can improve the communication performance.
Figures 5.5 (c) and (d) show E3SM-IO F case performance results on Summit. As the number of processes scales up, ROMIO Issend has significantly higher metadata and data communication time. The spread-out algorithm with throttling can effectively reduce this communication cost, which in turn improves the performance. Different from the Cori results, the communication time of the pairwise algorithm on Summit becomes faster as the number of processes increases because the number of two-phase I/O round is \( \lceil \frac{14.07 \text{GiB}}{16 \text{MiB} \times 512} \rceil = 2 \) on Summit 512 nodes. The number of rounds on Cori, on the other hand, is a constant 225 regardless of the number of processes used given stripe count 64 and size 1 MB. The pairwise algorithm is an effective design for all-to-all personalized communication. However, the straggler effect for many `MPI_Recv` slows performance significantly as the number of rounds increases for all-to-many communication. Asynchronous communication, on the other hand, suffers less from straggler effects because multiple communications are processed simultaneously.

The two-layer aggregation method (TAM) has a performance advantage with our settings compared with the rest of the algorithms. When TAM is used, the participants of the internode metadata and data communications are limited to local and I/O aggregators, so the communication scale is much smaller compared with explicit all-to-many communications. Instead of reducing contentions by throttling and communication reordering, TAM eliminates the contention implicitly by reducing the communication size. The reduction of communication contention by TAM is at the expense of casting extra intranode communication and memory footprints because the intranode aggregation phase requires local aggregators to receive extra data from other data within the same node. However, such extra costs are negligible compared with the reduction in communication.
cost, so TAM has a huge advantage over other explicit throttling approaches. The communication kernel of TAM is the same as the one adopted by default in ROMIO, which posts all send/receive requests in ascending rank order followed by a wait all operation. Unfortunately, we have not found any compelling evidence that replacing the communication kernels of two-layer aggregations with any all-to-many algorithms discussed in this chapter improves performance with our benchmarks that run on a few hundred compute nodes. Nevertheless, when the number of nodes scales from hundreds to tens of thousands in exascale computing, we expect that the overwhelming number of MPI_Irecv operations queued at receivers will become a bottleneck for the internode aggregation of TAM because of the reasons discussed in the implementation section. Theoretically, the internode aggregation stage of TAM can suffer from performance degradation caused by communication contentions, even if only one process per node participates in the internode aggregation. As a result, replacing its internode communication kernel with the balanced spread-out algorithm will help scale up TAM in the future for tens of thousands of compute nodes.

5.4.2. FLASH/IO Benchmark

The FLASH I/O benchmark suite is the I/O kernel of a block-structured adaptive mesh hydrodynamics code developed for the study of nuclear flashes on neutron stars and white dwarfs [40]. We use the checkpoint variables I/O, whose data pattern is an array of 24 variables that consist of 3D blocks of data, to evaluate communication kernels of two-phase I/O. We use a default block size of $8^3$. Each 3D block has a fourth dimension of size one, so the flatten offsets of any 3D block are contiguous. A variable distributes its
Figure 5.6. The partition of ROMIO Lustre collective write driver FLASH/IO data. There are 24 checkpoint variables, each with contiguous offsets represented in 3D blocks. The colors represent the file domains of I/O aggregators. An I/O aggregator receives metadata and data from ranks labeled with its color. Metadata exchange is in one round, and data exchange is in multiple rounds. Each round of data exchange spans three rows of file domains (blue, brown, and green) within a variable in this example. Both metadata and data communication patterns are not all-to-many. 3D blocks evenly to all processes. Each process has a fixed number of 80 to 82 3D blocks, so variable data sizes scale with the number of processes. As a result, FLASH/IO is a weak-scaling benchmark.

Figure 5.6 illustrates an example of I/O access orders of ROMIO two-phase I/O in multiple rounds on Lustre file systems with stripe size 3. Data gathered by I/O aggregators are illustrated in different colors. To simplify the illustration, we assume I/O data at all processes have the same size. In this example, each checkpoint variable spans two Lustre stripes, so there are two rounds of two-phase I/O per variable. Both metadata and data communication have many-to-many patterns instead of all-to-many. For instance, the
aggregator with light-blue file domain receives I/O requests from process ranks 0 and 3 only during metadata exchange. It receives data from process rank 0 or 3 exclusively per two-phase I/O round.

In general, the communication contention in FLASH I/O is less significant than contentsions in BTIO and E3SM F cases. With 80 3D blocks of \(8^3\) double-precision floating-point values, a process has \(80 \times 8^3 \times 8\) bytes of data. With our Lustre setting on Cori, an aggregator receives data from at most \(\left\lceil \frac{1 \text{MiB}}{327680\text{B}} \right\rceil = 4\) processes per two-phase I/O round. On the other hand, I/O aggregators on Summit receive data from \(\left\lceil \frac{16 \text{MiB}}{327680\text{B}} \right\rceil = 53\) processes per round. Furthermore, 1,260 processes send metadata to an aggregator on average for the GPFS driver running on 512 Summit nodes. The Lustre driver running on Cori KNL nodes, on the other hand, has only MPI 21 receive requests per I/O aggregator on average at the metadata exchange stage. Therefore, data exchange on Summit has a larger degree of contention compared with data exchange on Cori.

Figures 5.5 (e) to (h) summarize the performance results of writing FLASH I/O checkpoint in parallel. With the pairwise blocking kernel, the communication time for all experiments is slower compared with the rest of asynchronous communication implementations. For Cori results in Figures 5.5 (e) and (f), we present their timing values up to 250 seconds. However, the actual communication time is more than 20 minutes. The pairwise algorithm, implemented with MPI_SendRecv, constrains the order of data received by different aggregators with the blocking MPI functions. An aggregator may stay idle, waiting for communication of a process to finish, even though the aggregator does not receive any data from the process. Thus, applying the pairwise algorithm, designed for an all-to-all pattern, to a many-to-many communication pattern can cause a significant
straggler effect. On Cori, algorithms with throttling techniques do not show an advantage over the original version that posts all requests in ascending rank order. With throttling comm_size 32, each MPI_Waitall handles at most one send request at a sender, so such an over-throttling becomes an overhead. On GPFS, I/O aggregators receive data from a larger number of senders, so data communication benefits from throttling, as shown in Figures 5.5 (g) and (h).

5.4.3. BTIO Benchmark

Figure 5.7. 3D-layout of BTIO. The ROMIO Lustre collective write driver partitions BTIO data with three permutations (along the third dimension) represented by (a), (b), and (c). The colors represent the file domains of I/O aggregators. An I/O aggregator receives metadata and data from ranks labeled with its color. Metadata exchange is in one round, and data exchange is in multiple rounds. Each round of data exchange spans a file domain (blue, brown, and green) of three rows of the 2D array. The metadata communication pattern is all-to-many, but the data communication pattern is not.

BTIO, developed by NASA’s Advanced Supercomputing Division, is part of the benchmark suite NPB-MPI version 2.4 for evaluating the performance of parallel I/O [48]. I/O
requests in BTIO are partitioned in a block-tridiagonal pattern over a three-dimensional square array. Therefore, the BTIO benchmark must run with a square number of MPI processes, which divide the cross-sections of a global 3D array evenly. Thus, we increment the number of nodes on Summit to 168 and 672. Figure 5.7 illustrates an example of the data partition pattern of BTIO with 9 processes and 3 aggregators. The three-dimensional array is divided into 27 subarrays of size $3 \times 3 \times 3$. All I/O requests within a subarray are located on a single process. We can view the BTIO data pattern as three planes of 3D subarrays of size $3 \times 3 \times 3$. There are $\sqrt{p}$ permutations of subarrays owned by processes on the planes. The first permutation is a row-major ordering of subarrays. In our example, Figure 5.7 (a) illustrates the first 2D subarray in the 3D array with this permutation method. The next permutation is by decreasing the subarrays row index and increasing the column index, taking the modulus of $\sqrt{p}$. For example, the next subarray plane is illustrated by Figure 5.7 (b) transformed from the initial row-major permutation. Figure 5.7 (c) shows the last subarray plane.

With multiple rounds of two-phase I/O implementation in ROMIO on Lustre, an I/O aggregator handles only a limited range of file domains per round. One round of two-phase I/O handles a contiguous region of file domains, evenly divided among I/O aggregators. In our example, we use three colors to represent the file domains at three I/O aggregators. I/O requests marked with the same color are aggregated to the same aggregator. Two-phase I/O in a 2D array of I/O requests completes in two rounds. For example, the top three rows in Figure 5.7 (a) are completed in the first two-phase I/O run. The bottom three rows are completed in the second two-phase I/O run. Consequently, for any two-phase I/O round, an I/O aggregator gathers data from at most 3 processes, so
the data exchange pattern is many-to-many instead of all-to-many. Metadata exchange has a single round. I/O aggregators gather offset/length pairs for noncontiguous requests from all processes in our example, which exhibits an all-to-many communication pattern. Therefore, the metadata and data communication patterns can be different in the BTIO benchmark.

Figures 5.5 (i) and (j) illustrate the breakdown performance for running two-phase I/O on 64 and 256 Cori KNL nodes. The evenly spread-out algorithm has the best metadata exchange performance. However, the original ROMIO MPI_Issend implementation yields the best data exchange performance. As mentioned earlier, the data exchange pattern of BTIO is not all-to-many, so the original and balanced spread-out algorithms may underutilize the communication channels because of the throttling approach. The BTIO size has 1,024 40 MiB 2D data arrays stacked side by side. Each of the 2D data arrays is evenly divided among all processes in a different permutation of square blocks. Hence an aggregator gathers 1 MiB of data from approximately \( \sqrt{p} \) processes per two-phase I/O round on the Lustre file systems. Thus, the size of communication is much less than the all-to-many communication for metadata exchange. Similar to FLASH/IO, throttling can have a negative effect since the number of requests handled by MPI_Waitall is too small. To optimize communication performance, we use the original ROMIO communication kernel for data exchange and the evenly spread-out algorithm for the metadata communication. The results presented in the hybrid column are better than the results of all other columns. Thus, the proposed all-to-many algorithms are not universal for any type of many-to-many communication in multiple rounds. When the many-to-many
communication size is small, the I/O driver should fall back to the original two-phase I/O implementation without throttling to improve communication.

Figures 5.5 (k) and (l) illustrate the performance breakdown on Summit. Since the collective buffer size on GPFS is 16 MiB by default, almost half of the data in a 2D data array is collected by an aggregator. Thus, an I/O aggregator gathers data from approximately $\frac{1}{2}p$ of the processes. Although the communication pattern is not all-to-many, the number of senders is large enough to cause contention. Consequently, unlike the results on Cori, the evenly spread-out algorithm has an advantage over the traditional ROMIO Issend approach. The number of I/O aggregators scales up with the number of nodes on Summit, so there are fewer two-phase I/O rounds as the number of processes increases. Therefore, data communication with the pairwise algorithm is faster on 672 nodes than 168 nodes because straggler effects resulting from multiple rounds of data communications on Summit are smaller on a larger number of nodes. Our experiments on Cori, on the other hand, have a fixed number of I/O aggregators. Hence the pairwise algorithm slows down on a large number of nodes because the straggler effects also scale up with the increasing number of processes.

5.5. Conclusion

Metadata and data communication become performance bottlenecks of two-phase I/O when the communication pattern is all-to-many in multiple rounds. In this chapter, we replaced the two-phase I/O communication kernels In ROMIO that adapt to input I/O patterns, instead of using different all-to-many personalized communication algorithms.
Experimental results demonstrated the necessity of communication balancing and throttling for all-to-many communication patterns. Moreover, we expect that adjusting personalized all-to-many communication kernels can further reduce communication contentions for existing communication designs such as the two-layer aggregation, especially when the number of compute nodes scales to meet the demand of exascale computing in the future. However, when an input I/O pattern does not exhibit an all-to-many communication pattern, two-phase I/O should fall back to the implementation without communication throttling.

Several opportunities remain for future research. The communication kernels are chosen by users with MPI hints to adapt to their application I/O patterns. It is useful to let the I/O drivers parse the I/O pattern and dynamically select appropriate communication kernels. Furthermore, the communication throttling approach can take advantage of compute node topology information. Topology-aware throttling can reduce communication contention at network switches, which in turn further improves communication performance.
The increasing data size of scientific application workflows has created a strong demand for more powerful supercomputers. As the scale of supercomputer systems grows, system failures, which can negatively affect system performance [13], become critical. If knowledge about failures can be predicted via heuristics, one can deploy an autonomous system that can schedule resources and actions in order to maximize the overall system performance, as argued in [14]. For example, failure recovery measures such as checkpoint saving can be used to reduce the cost from system failures [15] [81].

In this chapter, we focus on failure predictions of IBM Blue Gene (BG) systems, but our method is not limited to these systems. Failure prediction for BG systems faces two significant challenges. First, the number of failure events is rare [1]. A classification problem with a small number of positive classes suffers from a high false-negative ratio. As a result, a model can have high accuracy but low precision and recall. Second, the prediction lead time, defined as the time difference between prediction and the failure, should be reasonably large; yet current lead times of only a few seconds do not give the system enough time to deploy failure prevention measures [82]. A more practical range of lead time is from minutes to hours, as argued in [1] and [82].
For the third-generation BG/Q, these two challenges have become more severe because of the growing reliability, availability, and serviceability (RAS) event log size. According to a case study of Mira logs from 2013 to 2017, the number of fatal events is much smaller than the number of nonfatal events. BG/P systems, on the other hand, did not have a similar ratio of nonfatal and fatal events, according to the case study of the Intrepid logs from 2013 to 2014. Moreover, Blue Gene/Q systems generate RAS events in real-time with high-volume RAS event data. For example, Mira generates 80 times more RAS event logs than did Intrepid from 2013 to 2014. Researchers have focused on failure predictions for the earlier Blue Gene systems using machine learning techniques, but the new challenges brought by the BG/Q system demand new methods that give better prediction accuracy and modeling of lead time.

To meet this demand, we propose a new formulation of features based on spatiotemporal locality assumptions mentioned in [83] and algorithms for failure prediction. We define the relation between system error and events. Then, we apply a temporal clustering algorithm for fatal event records. A cluster represents a system error that should be detected. Later, we propose a feature-matching method for linking nonfatal event clusters and fatal event clusters that are likely caused by the same system error. We construct two types of features from every nonfatal event cluster. A neural network is proposed to model the correlation between the two types of features and the occurrence of system error. We also propose a real-time failure detection algorithm using the trained model.

We present a case study using our proposed method with six years of reliability, availability, and serviceability event logs recorded by Mira, a Blue Gene/Q supercomputer at Argonne National Laboratory. We use the 2013-2017 Mira RAS event data as training
data for prediction models. Then, we emulate the real-time prediction of failures for the period of data in 2018 using the trained models. We use accuracy, precision, recall, and the $F_1$ score as metrics and present results based on a variety of model parameters. In addition to predicting the occurrence of failures in general, we present the use cases of our proposed algorithm for critical types of failures: coolant and power failures.

The rest of this chapter is arranged as the following. In section 2, we briefly introduce the BG/Q system and discuss existing literature related to the Blue Gene system anomaly detection. In section 3, we propose our feature extraction techniques, training model, and real-time anomaly detection algorithm. In section 4, we present a case study of five years of Mira RAS event logs using our proposed methods.

The rest of this chapter is arranged as the following. In Section 6.1 we briefly introduce the BG/Q system and discuss existing literature related to the Blue Gene system anomaly detection. In Section 6.2 we propose our feature extraction techniques, training model, and real-time anomaly detection algorithm. In Section 6.3 we present a case study of five years of Mira RAS event logs using our proposed methods.

6.1. Background and Related Work

BG/Q is the third generation of the IBM Blue Gene series of supercomputers that can be scaled up to 20 PFs [84]. Details about network and message units are described in [85].

BG/Q systems have two major types of nodes: compute nodes and I/O nodes. Compute nodes are used for running applications, and I/O nodes are used for storing files shipped from computing nodes. The compute nodes are interconnected in a five-dimensional
torus topology. Every node has ten bidirectional ports with 2 GB/s bandwidth. In addition to computing and I/O nodes, service nodes are used to connect to every compute node, and every I/O node via JTAG interfaces with 1 Gb Ethernet for monitoring the system \[86\]. These service nodes report runtime noninvasive RAS events, which are the objectives of this study.

The Blue Gene/Q Reliability, availability, and serviceability (BG/Q RAS) Events Book \[87\] describes the details about RAS events of the BG/Q system. Every RAS event has many attributes, such as component, category, severity, and time stamp, location.

The followings attributes are of particular interest in this chapter.

(1) **Component**: the software component detecting and reporting the event, for example, CNK (compute node kernel), MC (machine controller), and MUDM (memory unit)

(2) **Category**: the entity that encountered an error, for example, software error, BQC (chip error), Coolant, AC/DC power, and DDR (memory controller)

(3) **Severity**: the various levels of severity: INFO – message that highlights the progress of system software; WARN – a message that indicates potential harmful situations, such as a software error threshold or failure of a redundant component; and FATAL – the message that indicates severe system errors, which can lead to application fail or abort

(4) **Event time**: timestamp of the event in seconds

(5) **Location**: the rack, midplane, node board, and node that an event is reported from
Predicting BG system failures is a concern of the high-performance computing (HPC) community for handling resilience on exascale supercomputers. Failures of high-end supercomputers can disrupt the running of applications. A commonly used strategy is to save checkpoints regularly. Later, if a node fails, the application can restart at another node with the data status at the saved checkpoints. If future failures can be accurately predicted based on historical data, checkpoints can be saved accordingly, instead of regularly. Thus, the efficiency of handling resilience at the exascale can be improved with the knowledge of future failure predictions.

Liang et al. introduced the concept of clustering and compression for RAS event data. Their methods can successfully remove more than 99% of raw RAS events without losing the accuracy for portraying the failures. Subsequently, Liang et al. formulated the prediction of failures in BG/L systems as a problem of nonfatal and fatal RAS event correlation. They also showed the temporal and spatial localities of the events in the system. This assumption has also been validated in another study. More recently, Liang et al. proposed a feature extraction method for predicting failures in BG/L systems. This method, known as the period-based method, is widely used for fatal event prediction. A period-based model is a prediction model that uses a fixed temporal window size for feature extractions. The model used in divides time into fixed-size intervals; features extracted from the intervals are used to predict failures in future intervals.

Following Liang and his colleagues’ works, researchers proposed new techniques for predicting failures in BG systems. Gujrati et al. proposed a meta-learning failure predictor for BG/L systems. Zheng et al. proposed preprocessing methods for system RAS events.
The concept of lead time was introduced in [92]. Lead time refers to the time difference of failure and the alarm for that failure raised by a prediction model. Thompson et al. [93] have tested their intention to maximize the lead time for their model. Zheng [82] proposed an approach that can estimate the lead time using the arithmetic mean of time differences between nonfatal and fatal events.

Event-driven approaches associate a fixed number of adjacent events reported before the occurrence of failures. Yu et al. [94] evaluated the impact of lead time and window size for both period-based and event-driven prediction models using a Bayesian network as a prediction model. Di et al. studied the correlation among fatal events [95] [96]. Later, they presented a similarity-based event filtering analysis for BG/Q system [97]. The event-driven approach differs from the period-based approach in the way that the time window for RAS events features are not fixed ranges. Instead, the number of events is fixed.

Similar to most machine learning problems, the modeling of failure occurrences for the BG/Q system can be divided into two phases: feature extraction and model fitting. In the feature extraction process, the input is a sequence of nonfatal events, and the output is if fatal events will occur in the future. The matching of the nonfatal event features to failure prediction labels is a challenging task [82]. If the nonfatal event features are not associated with the right fatal events, the prediction accuracy can be undesirable, since they do not necessarily correlate with each other. Zheng [82] used a randomized algorithm to correlate the pairs with a high Pearson correlation coefficient. However, their approach is applicable to post analysis of historical log instead of real-time prediction. Furthermore, recent studies have shown that lead time is critical. If the lead time is too
small, the system does not have response time for checkpoint saving. On the other hand, if the lead time is too large, it is difficult to predict with high accuracy the timestamp interval that a potential failure can happen.

The most widely used real-time anomaly detection algorithm for BG systems is the period-based approach proposed by [1]. Although not discussed in their chapter, the period-based method has the clear advantage of bounding the lead time. Moreover, their method can be readily deployed to control the nodes of real-time systems. Yu et al. [94] concluded that the event-driven feature extraction method constructs better features because the matching of nonfatal events and fatal events can be more accurate. However, real-time event-driven methods have not received sufficient attention because of the difficulty of feature construction. In the post-analysis of historical logs, we can associate fatal events with the last few nonfatal events according to their timestamp, but this binding method cannot be used in real-time systems since the goal is to predict whether those fatal events will occur or not. In this chapter, we address the problem of event-driven real-time anomaly detection by proposing a new algorithm.

6.2. Design

In a supercomputer system, let $E = \{e_i : i \in \mathbb{N}\}$ be independent temporal system errors. $E$ is a hidden sequence that a system-monitoring program aims to predict in advance. Every RAS event is associated with one of these system errors so that we can infer the system errors from the RAS events. Moreover, some system errors are severe errors that can prevent a block from booting. For those severe errors, they report at least one RAS event with FATAL severity.
Programs at control nodes can use RAS events that have been observed to predict whether any fatal RAS events will occur in the future. To be more specific, let \( X = \{ x_{t+i} : i \in [0, k] \} \) be a sequence of observed RAS events. Let \( Y \) be labels that indicate the occurrence of a system error at a certain time. An association function \( f \) can be used to make predictions of future failure occurrences.

The design has three objectives. First, \( X \) and \( Y \) should refer to the same \( e_i \in E \). We use a spatiotemporal locality assumption to achieve this goal. Second, we want to apply the lead time constraint, which means that the time difference between the last timestamp of \( X \) and the occurrence of \( Y \) should be reasonable. For example, predicting some FATAL event that will happen in the next 60 hours is not useful, since the time range is too large. Similarly, predicting the occurrence of a fatal event that will happen in the next second is also not helpful since the system does not have enough response time. We achieve our second objective by using a time window. All predictions of failures should be within the specified time window. Third, the association function \( f \) must support real-time systems. To achieve this objective, we propose an anomaly detection algorithm that can utilize trained prediction models.

6.2.1. Identifying Independent Failures

The first question to be answered is the definition of system anomaly sequence \( E \). We apply a temporal clustering algorithm for this purpose. The underlying assumption is that the fatal RAS event indicating the same system error has spatiotemporal locality. This assumption has been argued in \[98\], \[99\] and \[83\].
For arbitrary RAS event records $x_a$ and $x_b$, let $\text{Dist}_t : X \times X \rightarrow \mathbb{N}$ be the temporal distance function. The function is computed as the difference between the timestamps of $x_a$ and $x_b$. Let $h_t$ be the temporal distance threshold. For INFO/WARN RAS events and FATAL RAS events separately, we apply the clustering algorithm proposed in [88] to the RAS events with the temporal distance function $\text{Dist}_t$ and threshold $h_t$ to obtain temporal clusters $T_F$. The algorithm can be summarized as follows. For all events sorted in ascending order of their timestamps, if two adjacent events have a temporal distance less than $h_t$, they are joined into the same cluster. By the spatiotemporal locality assumption, RAS events in the same spatiotemporal cluster are associated with the same system errors.

### 6.2.2. Feature Construction

![Feature Construction Diagram]

Figure 6.1. This figure illustrates an example for feature construction as mentioned in Algorithm [13]. We set $n_{\text{event}} = 4$. The first feature $x_1$ consists of the first 4 RAS events. The second feature $x_2$ consists of 5 RAS events. The condition at Line 7 is triggered for building $x_2$. After constructing $x_2$, Line 12 is triggered to jump the starting index of next RAS event to the beginning of $x_3$.

We build input feature vectors from raw RAS events for predicting occurrence of failures. There are two types of variable construction, as mentioned in the background.
Algorithm 12: Feature Construction

Data: Ras event sequence $r_1, \ldots, r_n$, minimum number of events for prediction $n_{\text{event}}$, temporal merge threshold $t_{\text{max}}$

Result: An array of feature vectors $F$.

1. $F \leftarrow \emptyset$
2. start $\leftarrow 1$
3. $s \leftarrow 0$
4. while $i \leq n$ do
5.   if $s = n_{\text{event}}$ then
6.     end $\leftarrow i$
7.     while end $\leq n$ or $r_{\text{end}.\text{time}} - r_i.\text{time} < t_{\text{max}}$ do
8.       end $\leftarrow$ end + 1
9.     end
10.    F add feature vector $\{r_{\text{start}}, \ldots, r_{\text{end}-1}\}$
11.   $i \leftarrow$ start
12.   while $i \leq \text{end}$ and $r_{\text{start}.\text{time}} - r_i.\text{time} < t_{\text{max}}$ do
13.     $i \leftarrow i + 1$
14.   end
15.   start $\leftarrow i$
16.   $s \leftarrow 0$
17. else
18.   $i \leftarrow i + 1$
19.   $s \leftarrow s + 1$
20. end

section. One approach is the period-based approach, and the other method is the event-driven approach.

Suppose we have RAS events sequence $x_0, x_1, x_2, \ldots$ ordered by timestamps. For event-driven approach proposed in [82], there is a parameter $n_{\text{event}}$ that indicates a threshold for number of adjacent events for forming a variable. For instance, the $j^{\text{th}}$ variable selected is $\{x_i : j \leq i < n_{\text{event}} + j\}$. Therefore, the total number of variables created is equal to the total number of events subtracted by $n_{\text{event}} - 1$. This method works well for BG/L and BG/P systems. However, for BG/Q systems, the number of RAS events is significantly
larger than BG/P systems. For example, Mira dataset [100] has one RAS event per 3 seconds on average. Hence the control nodes have to make predictions every 3 seconds. Since adjacent variables only differ by 1 event, adjacent features have small differences given large \( n_{\text{event}} \). Moreover, a lot of RAS events in BG/Q systems have precisely the same timestamp, so the variables created depend on the order of RAS events with the same timestamp, which is not well-defined for input variable construction.

We propose an event-driven approach for selecting input variables. Instead of selecting a fixed \( n_{\text{event}} \) number of RAS events per variable, the new approach sets \( n_{\text{event}} \) as a minimum threshold. Adjacent events with timestamp differences less than a threshold \( t_{\max} \) are merged into the same feature. Therefore, a feature contains at least \( n_{\text{event}} \) number of RAS events. Algorithm 12 formally describes the proposed feature construction approach. The algorithm iterates through all events with the "while" loop at Line 4. If the number of accumulated events reaches \( n_{\text{event}} \) at Line 5, the algorithm constructs an input feature. In addition to events with index from variable start to \( i \), Lines 6 to 9 absorb events with timestamps difference from \( i \) less than \( t_{\max} \). Lines 11 to 14 shift the starting index of the next feature, jumping events with timestamps too close to \( x_{\text{start}} \). Figure 6.1 illustrates an example for how RAS events are merged. \( t_{\max} \) is equal to 4 in the example. Feature \( x_1 \) and \( x_2 \) show the case that more than \( t_{\max} \) number of events are absorbed into the same input feature by Lines 6 to 9 of Algorithm 12. \( x_3 \) shows that the case that the starting indices can have a gap for adjacent input features, which is handled by Lines 11 to 14 of Algorithm 12.
Figure 6.2. Illustration of convolutional neural network architecture used for training. The 2D convolutional layer has kernel size $2 \times 15$ and stride size $2 \times 1$. The pooling layer followed by the 2D convolutional layer has kernel size $1 \times 15$. The 1D convolutional layers have kernel size 3. The pooling layers followed by the 1D convolutional layers have kernel size 2. Padding is applied to all convolutional+pooling layers. Each of the fully connected layers has 2,048 hidden nodes.

Algorithm 13: Anomaly Detection Algorithm

Data: Maximum lead time threshold $t_{\text{lead}}$, minimum number of events triggering prediction $n_{\text{event}}$, prediction model $\hat{f}$

Result: Continuously report if a failure will happen within the lead time range.

1. $s \leftarrow 0$
2. $F \leftarrow \text{Empty Array}$
3. while True do
4.   $r \leftarrow \text{NextEvent}$
5.   $s \leftarrow s + 1$
6.   if $r.$time $- F.$last.$time > $t_{\text{lead}}$ $\lor$ $s \leq n_{\text{event}}$ then
7.     Add $r$ to $F$
8.   else
9.     Report $\hat{f}(F)$
10.    Clear $F$
11.   Add $r$ to $F$
12. end
13. end

6.2.3. Realtime Anomaly Detection

The proposed feature construction method in Section 6.2.2 supports real-time anomaly detection.
Figure 6.3. (a) Log scale (based 10) number of events for 2013–2018. (b) Number of RAS events with INFO severity across all five years. (c) Number of RAS events with WARN severity across all five years. (d) Number of RAS events with FATAL severity across all five years.

For every input feature vector, we match it with the nearest independent failure with a larger starting timestamp. The difference between the starting timestamp of $X$ and the starting timestamp of its matched independent failure is defined as the lead time. We can define a lead time threshold $t_{lead}$. If an input feature vector has lead time greater than $t_{lead}$, we label it with STATUS_SAFE. Otherwise, we label the input feature with STATUS_FATAL. This threshold is also denoted as maximum lead time.
Having constructed input features vectors for $X$ and labels $Y$, we propose a deep learning model for emulating $f$. Figure 6.2 illustrates the proposed architecture of the deep learning model. There are two kinds of feature vectors. The first type summarizes the statistical distribution for the event attributes. The second type captures the spatiotemporal features for event occurrence. Therefore, the proposed model consists of two parts. The first part is a convolutional neural network that has type 2 features as input. After two steps of convolution and pooling, the dimension is reduced to $1 \times \frac{1}{30} w$. In the second part, the output of the convolutional neural network is joined with type 1 features. The joined vector is fed into a fully connected neural network for softmax classification.

For the first type of feature vector, we reuse two features proposed in [1], namely, the mean of time intervals between adjacent events in $X$ and the time elapsed since the occurrence of the last fatal event. In addition, we propose the following features. Every event in $X$ has component and category attributes in terms of strings. We compute the joint probability of the component and category pair in $X$. The joint probability distribution models the type of system error. Moreover, we count the number of distinct spatial locations for events in $X$. There are four distinct spatial levels: rack, midplane, node board, and node. This feature indicates whether $X$ contains events from a wide range of locations or not.

For the second type of feature vector, we record the spatiotemporal difference between adjacent pairs of events in $X$. We record a vector of size $2 \times |X|$: one dimension is for temporal differences, and the other dimension is for spatial differences. The temporal difference is defined as the difference in timestamps of two events. The spatial difference is defined as the difference in spatial level. If two RAS events are on different racks, they
have spatial difference 4. Otherwise, if they are on different midplanes, they have spatial
difference 3. Otherwise, if they are on different node boards, they have spatial difference
2. If they are on the same node board, but different node, they have spatial difference
1. If the two events are on the same node, they have spatial difference 0. This type of
feature vector explains spatiotemporal variations within $X$.

Algorithm 13 illustrates our proposed anomaly detection algorithm. A while loop
at Line 3 keeps receiving RAS events reported from the control node. The algorithm
dynamically constructs features in realtime using the same feature construction strategy
described in Algorithm 12. After a feature vector is constructed, the algorithm report if
there are any anomalies using regression function $\hat{f}$.

6.3. Experimental Results

![Figure 6.4. The frequency of all fatal events in logarithm (10) scale.](image)
We use the Mira dataset [100] to evaluate the proposed algorithms. This dataset contains RAS event logs from 2013 to 2018. Figure 6.3a summarizes the number of events by severity. Most events have WARN-level severity. Events with INFO severity in Figures 6.3b, 6.3c and 6.3d indicate the number of events across all years by severity. We can observe that there are far more RAS WARN events than events with INFO and FATAL severity. Almost every 3 seconds, a WARN event will be reported.

Our case study is the type of application that adopts a prediction model for real-time anomaly detection along with a single time series. Supercomputing systems are constantly improved. Past errors may not occur in the future anymore. Results for using future data to predict past data do not necessarily reflect the model’s efficacy. Thus, we should not split the training and testing data in the way that some training data has larger timestamps than testing data since models trained with such settings are not useful for real-world applications. Consequently, the training part of the data is composed of RAS events from 2013 to 2017. RAS events with timestamps in 2018 are used for evaluation.

Table 6.1. Number of events used for training and testing. Events for training part have timestamps from 2013 to 2018. Events for testing part have timestamps in 2018.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Training</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>INFO</td>
<td>16,725,291</td>
<td>4,800,661</td>
</tr>
<tr>
<td>WARN</td>
<td>46,127,365</td>
<td>13,762,599</td>
</tr>
<tr>
<td>FATAL</td>
<td>2,580,811</td>
<td>239,719</td>
</tr>
<tr>
<td>Total</td>
<td>65,433,467</td>
<td>18,802,979</td>
</tr>
</tbody>
</table>

Table 6.1 summarizes the distribution of RAS events in training and testing data according to severity. While the number of events in 2018 for testing is approximately 20% of the total number of events from 2013 to 2018, we can observe that the distribution
of fatal events is not uniform in training and testing data. In 2018, the number of fatal
events was only 1.3% of the number of all events in this time range. On the other hand,
from 2013 to 2017, the number of fatal events is 3.9% of all events in this time range.
Furthermore, if we apply the method proposed in Section 6.2.1 for clustering all fatal
events with a temporal threshold of one minute. The total numbers of fatal clusters are
3076 from 2013 to 2017 and 119 in 2018. Therefore, the testing set has much less sum of
the positive class than the training set. Since the dataset is highly imbalanced, a random
guesser on the testing set cannot achieve an expectation of $F_1$ score of more than 0.05.
Thus, the failure detection task is challenging.

These distributions of fatal events create a challenge for prediction models since mod-
els can overfit the training dataset based on the fatal event number. We address the
imbalanced data problem by multiplying larger weights to the positive class in the cost
function during gradient descent. Thus, the false-negative prediction has a higher cost
in the training process. Therefore, this method prevents the model from overfitting ac-
ccuracy by reducing the false-negative predictions. We also applied the receiver operating
characteristic (ROC) curve for determining the threshold of classification.

6.3.1. General Failure Detection

We present a case study for detecting general occurrence of system errors for the Mira
dataset. Failure prediction models are built offline. Offline models do not require frequent
updates given abundant historical data. However, frequent updates of the prediction
models can incorporate failure types that are unobserved in the past. Nonetheless, in
our case study, we assume the model is updated once a year. Algorithm 13 utilizes the
Table 6.2. Case study for the Mira dataset using proposed methods. A prediction model $\hat{f}$ is trained with data from 2013 to 2018 by using features extracted by Algorithm 12. We apply Algorithm 13 with the trained model to predict failures in 2018.

<table>
<thead>
<tr>
<th>$n_{\text{event}}$</th>
<th>Max lead time</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>$F_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>1800</td>
<td>99.4%</td>
<td>37.3%</td>
<td>65.9%</td>
<td>47.6%</td>
</tr>
<tr>
<td>300</td>
<td>3600</td>
<td>99.3%</td>
<td>43.8%</td>
<td>67.5%</td>
<td>53.8%</td>
</tr>
<tr>
<td>300</td>
<td>7200</td>
<td>99.0%</td>
<td>39.4%</td>
<td>67.5%</td>
<td>51.4%</td>
</tr>
<tr>
<td>600</td>
<td>1800</td>
<td>99.6%</td>
<td>41.3%</td>
<td>47.1%</td>
<td>44.0%</td>
</tr>
<tr>
<td>600</td>
<td>3600</td>
<td>99.5%</td>
<td>40.8%</td>
<td>64.7%</td>
<td>50.0%</td>
</tr>
<tr>
<td>600</td>
<td>7200</td>
<td>99.6%</td>
<td>46.4%</td>
<td>63.9%</td>
<td>53.8%</td>
</tr>
<tr>
<td>900</td>
<td>1800</td>
<td>99.6%</td>
<td>67.4%</td>
<td>39.5%</td>
<td>49.8%</td>
</tr>
<tr>
<td>900</td>
<td>3600</td>
<td>99.4%</td>
<td>50.5%</td>
<td>47.6%</td>
<td>49.0%</td>
</tr>
<tr>
<td>900</td>
<td>7200</td>
<td>98.7%</td>
<td>28.7%</td>
<td>51.6%</td>
<td>36.9%</td>
</tr>
<tr>
<td>1200</td>
<td>1800</td>
<td>99.6%</td>
<td>56.5%</td>
<td>43.1%</td>
<td>48.9%</td>
</tr>
<tr>
<td>1200</td>
<td>3600</td>
<td>99.4%</td>
<td>46.7%</td>
<td>45.7%</td>
<td>46.2%</td>
</tr>
<tr>
<td>1200</td>
<td>7200</td>
<td>99.2%</td>
<td>71.1%</td>
<td>45.7%</td>
<td>55.6%</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Period</th>
<th>Observation Window</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
<th>$F_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1800</td>
<td>5</td>
<td>95.9%</td>
<td>21.3%</td>
<td>35.8%</td>
<td>26.7%</td>
</tr>
<tr>
<td>1800</td>
<td>10</td>
<td>95.5%</td>
<td>13.4%</td>
<td>24.6%</td>
<td>17.4%</td>
</tr>
<tr>
<td>3600</td>
<td>5</td>
<td>91.8%</td>
<td>28.0%</td>
<td>36.5%</td>
<td>31.7%</td>
</tr>
<tr>
<td>3600</td>
<td>10</td>
<td>91.9%</td>
<td>35.7%</td>
<td>39.2%</td>
<td>37.4%</td>
</tr>
<tr>
<td>7200</td>
<td>5</td>
<td>87.9%</td>
<td>37.3%</td>
<td>54.5%</td>
<td>45.4%</td>
</tr>
<tr>
<td>7200</td>
<td>10</td>
<td>85.5%</td>
<td>42.1%</td>
<td>43.7%</td>
<td>42.9%</td>
</tr>
</tbody>
</table>

prediction model for real-time prediction of unseen data from the future. We present simulation results.

Figure 6.4 illustrates the frequency of all types of fatal events. We can observe that ‘MC-BQC’ and ‘MMCS-AC_TO_DC_PWR’ have more than one million events. Most failures are related to these two. MC refers to the machine controller running on the
service node, and MMCS refers to the control system running on the service node. BQC refers to compute chip error. This failure causes the termination of computing jobs. AC_TO_DC_PWR refers to circuit power issues. It implies that the entire node board is in error. All jobs must be terminated on the node board. Thus it is a more severe type of error.

Firstly, we build models for predicting any types of failures in order to make a direct comparison with the previous study. Later, we build prediction models for 'MMCS-AC_TO_DC_PWR' and 'MMCS-Coolant,' which are detrimental failures that can be in the interests of system administrators.

On the testing dataset, let the true positive $tp$ be the number of STATUS_FATAL results that $\hat{f}$ predicts correctly. Let the false-negative $fn$ be the number of STATUS_FATAL results that $\hat{f}$ fails to predict. Let the false-positive $fp$ be the number of false-alarm STATUS_FATAL results that $\hat{f}$ reports incorrectly. Let the true negative $tn$ be the number of STATUS_SAFE results that $\hat{f}$ predicts correctly. We use accuracy, precision, recall, and the $F_1$ score as the evaluation metrics. Accuracy is the percentage of predictions that $\hat{f}$ is correct on the testing dataset, defined as $\frac{tp + tn}{tp + tn + fp + fn}$. Precision is the percentage of STATUS_FATAL results that $\hat{f}$ can retrieve from the testing dataset, defined as $\frac{tp}{tp + fp}$. Recall that the percentage of predictions with output STATUS_FATAL that $\hat{f}$ made are correct, $\frac{tp}{tp + fn}$. The $F_1$ score is defined as $2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$.

We present the results detection of all system failures in Table 6.2 with different choices of parameters. In general, the accuracy of prediction on the testing dataset is high: more than 98%. High accuracy alone is not sufficient since the dataset is highly imbalanced. Positive class is approximately 5% in the constructed features. Thus, a classifier that
always predicts negative class can also achieve 95% accuracy. Therefore, we must present precision and recall results.

One observation is that as we increase $n_{\text{event}}$, the recalls have a decreasing trend, regardless of the maximum lead time for a fatal prediction. Precision, on the other hand, is less stable due to the imbalanced nature of testing data. The fatal clusters that define the binary labels of the classification contain too many different types of fatal events. More than 2,500,000 of fatal events are divided into approximately 3,000 clusters. Thus, it is possible that one cluster contains many independent system errors over an extensive time range. Different types of system error can have very different prediction lead time, which is justified in the next section. However, the occurrence of a fatal cluster is determined by the first independent system error. Therefore, if $n_{\text{event}}$ is not large enough, the prediction model may not be able to cover the features for the first system error. Thus, increasing $n_{\text{event}}$ improves recalls of the model. However, setting $n_{\text{event}}$ to be too large could cause the curse of dimensionality problem. Lots of noise are introduced into the input feature.

We implemented the period-based fatal prediction approach proposed in [1]. This approach is the most widely used real-time anomaly detection algorithm for BG system, so we make a direct comparison of our method to it. Using the same training and testing dataset, we summarize the results in Table 6.3 with different parameters suggested in the chapter. The Period column refers to the observation window size in seconds. It defines the maximum lead time of the prediction model. The Observation Windows column refers to the number of observation windows used for building the input features. 5 and 10 are reasonable parameters suggested in the original chapter.
With the same maximum lead time, our proposed anomaly detection algorithm achieves higher accuracy, precision, and recall. The proposed algorithm adopts a new event-based feature extraction algorithm. It avoids the association of unrelated input features and prediction according to the spatiotemporal locality assumption. Furthermore, with the help of two different types of features, the model can achieve better results.

6.3.2. Critical Failure Detection

As mentioned earlier, "MMCS AC TO DC PWR" and 'MMCS Coolant' (control system circuit and coolant failures) are the fatal events that report the failure of a large partition of supercomputing systems.

We filter out the fatal events with attributes 'MMCS AC TO DC PWR' and 'MMCS Coolant.' Similar to the previous experiments, we use 2013-2017 RAS event data as training data and 2018 RAS event data as testing data. Depending on the different maximum lead time, the distributions of positive and negative classes vary. In our case study, the maximum lead time is from 900 to 7200 seconds. For coolant failures, the number of the positive class in training data is from 2067 to 2148 in training data and from 939 to 947 in testing data. The number of negative class in training data is from 107349 to 107430 in training data and from 18235 to 18243 in testing data. For AC/DC power failures, the number of the positive class in training data is from 1466 to 2388 in training data and from 575 to 650 in testing data. The number of negative class in training data is from 79266 to 80188 in training data and from 20994 to 21069 in testing data. Features for both failures are highly imbalanced. The AC/DC power failure has a
smaller proportion of the positive class, so failure detection for it can be more challenging than coolant failure.

Table 6.4 illustrates the precision, recall, and $F_1$ score of "MMCS-AC TO DC PWR" and "MMCS Coolant" failures detected with different parameters. We do not present the accuracy results since all of them are extremely high, ranging from 99.8% to 99.9%.

For Coolant failure results presented in Table 6.4, we can observe the following trends. Setting the maximum lead time 1800 seconds gives the best results for any $n_{event}$. Given the same lead time 1800 seconds, $n_{event} = 60$ yields best precision and recall, which is up to 97% $F_1$ score. For the AC to DC power failure results, $n_{event} = 240$ and maximum lead time of 3600 seconds yield the best precision and recall.

The best parameter settings for coolant and AC to DC power suggested that the number of events and lead time we should choose for different types of failures can be different. If the number of events we use for making a prediction is too large for coolant failure (240 events), we could introduce too much noise to the input feature, which could cause overfitting problems during the training, given the fact that the number of coolant events is very small. AC to DC power failure, on the other hand, has 100 times more events than the coolant failure. Thus, it is preferable to use a larger size of input feature. The best choice of lead time depends on the system characteristics. Broad lead time can increase the false-positive rate because the number of the positive class is forced to be reduced. Choosing a narrow lead time range can increase the false-negative rate since the scope of the negative class is enlarged.

In this section, we have shown that it is possible to build models for specific types of failures. Compared with results in Table 6.2, the best results in Table 6.4 are much
better. As mentioned earlier, the traditional assumption for the spatiotemporal locality of a system error may not be applicable to BG/Q data. Multiple independent system errors can occur in a single time range since the total number of components in the whole system is enormous. Thus, the detection occurrence of specific types of failure can be a good future direction.

Table 6.4. Case study for the Mira dataset using proposed methods for coolant and power related events. A prediction model $\hat{f}$ is trained with data from 2013 to 2018 by using features extracted by Algorithm 12. We apply Algorithm 13 with the trained model to predict failures in 2018.

<table>
<thead>
<tr>
<th>Type</th>
<th>$n_{\text{event}}$</th>
<th>Max lead time</th>
<th>Precision</th>
<th>Recall</th>
<th>$F_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coolant</td>
<td>15</td>
<td>900</td>
<td>69.7%</td>
<td>88.5%</td>
<td>78.0%</td>
</tr>
<tr>
<td>Coolant</td>
<td>15</td>
<td>1800</td>
<td>69.7%</td>
<td>88.5%</td>
<td>78.0%</td>
</tr>
<tr>
<td>Coolant</td>
<td>15</td>
<td>3600</td>
<td>69.7%</td>
<td>85.2%</td>
<td>76.7%</td>
</tr>
<tr>
<td>Coolant</td>
<td>15</td>
<td>7200</td>
<td>69.4%</td>
<td>80.6%</td>
<td>74.6%</td>
</tr>
<tr>
<td>Coolant</td>
<td>60</td>
<td>900</td>
<td>82.4%</td>
<td>100%</td>
<td>90.3%</td>
</tr>
<tr>
<td>Coolant</td>
<td>60</td>
<td>1800</td>
<td>94.1%</td>
<td>100%</td>
<td>97.0%</td>
</tr>
<tr>
<td>Coolant</td>
<td>60</td>
<td>3600</td>
<td>94.1%</td>
<td>91.4%</td>
<td>92.8%</td>
</tr>
<tr>
<td>Coolant</td>
<td>60</td>
<td>7200</td>
<td>94.1%</td>
<td>78.0%</td>
<td>85.3%</td>
</tr>
<tr>
<td>Coolant</td>
<td>240</td>
<td>900</td>
<td>59.1%</td>
<td>100%</td>
<td>74.3%</td>
</tr>
<tr>
<td>Coolant</td>
<td>240</td>
<td>1800</td>
<td>68.2%</td>
<td>88.2%</td>
<td>76.9%</td>
</tr>
<tr>
<td>Coolant</td>
<td>240</td>
<td>3600</td>
<td>68.2%</td>
<td>75.0%</td>
<td>71.4%</td>
</tr>
<tr>
<td>Coolant</td>
<td>240</td>
<td>7200</td>
<td>68.2%</td>
<td>65.2%</td>
<td>66.7%</td>
</tr>
<tr>
<td>AC/DC</td>
<td>15</td>
<td>900</td>
<td>35.2%</td>
<td>62.7%</td>
<td>45.1%</td>
</tr>
<tr>
<td>AC/DC</td>
<td>15</td>
<td>1800</td>
<td>42.3%</td>
<td>55.6%</td>
<td>48.1%</td>
</tr>
<tr>
<td>AC/DC</td>
<td>15</td>
<td>3600</td>
<td>54.6%</td>
<td>51.7%</td>
<td>53.1%</td>
</tr>
<tr>
<td>AC/DC</td>
<td>15</td>
<td>7200</td>
<td>43.5%</td>
<td>46.3%</td>
<td>44.9%</td>
</tr>
<tr>
<td>AC/DC</td>
<td>60</td>
<td>900</td>
<td>49.2%</td>
<td>48.9%</td>
<td>49.1%</td>
</tr>
<tr>
<td>AC/DC</td>
<td>60</td>
<td>1800</td>
<td>52.6%</td>
<td>50.0%</td>
<td>51.2%</td>
</tr>
<tr>
<td>AC/DC</td>
<td>60</td>
<td>3600</td>
<td>71.5%</td>
<td>61.8%</td>
<td>66.3%</td>
</tr>
<tr>
<td>AC/DC</td>
<td>60</td>
<td>7200</td>
<td>51.5%</td>
<td>58.2%</td>
<td>54.7%</td>
</tr>
<tr>
<td>AC/DC</td>
<td>240</td>
<td>900</td>
<td>65.2%</td>
<td>51.0%</td>
<td>57.3%</td>
</tr>
<tr>
<td>AC/DC</td>
<td>240</td>
<td>1800</td>
<td>28.1%</td>
<td>53.3%</td>
<td>36.8%</td>
</tr>
<tr>
<td>AC/DC</td>
<td>240</td>
<td>3600</td>
<td>97.5%</td>
<td>71.8%</td>
<td>86.3%</td>
</tr>
<tr>
<td>AC/DC</td>
<td>240</td>
<td>7200</td>
<td>97.5%</td>
<td>71.8%</td>
<td>82.7%</td>
</tr>
</tbody>
</table>
6.4. Summary

In this chapter, we have proposed an algorithm for detecting system anomalies in the BG/Q system. The problem is challenging because system failures are rare. The algorithm can construct features based on a spatiotemporal locality assumption and make a prediction using our customized convolutional neural network in real-time. Experimental results have shown that the proposed approach has better prediction accuracy compared with the traditional period-based method in a previous study with controllable lead time. We have found that our proposed approach is applicable for detecting specific critical types of failures.

There are opportunities to improve the failure prediction model better. The current prediction model is a binary classifier. It can be generalized to a multi-class classifier. In other words, instead of the predicting occurrence of a failure within a fixed lead time, the model can output probabilities of failures within a different length of lead time ranges in real-time.
CHAPTER 7

Predicting Resource Requirement in Intermediate Palomar

Transient Factory Workflow

A transient astronomical event, or simply a transient, is an astronomical object or event with a relatively short lifespan, instead of having a fixed location in the sky, like a star, or a periodic orbit, like a planet \[101\]. Studying transients such as the thermonuclear or core-collapse of a supernova helps researchers to understand the life-cycle of stars and their interaction in the formation of galaxies and the evolution of our universe. Thus there are several large synoptic surveys searching for these transients, for example, SLOAN \[102\], CATALINA \[103\], ATLAS \[104\], the Palomar Transient Factory (PTF) \[105\], and the Zwicky Transient Facility (ZTF) \[106\]. Since the objects of interest in these projects are short-lived, it is critical to be able to process the images taken quickly, so that detailed follow-up observations can be made on the transients just identified \[107, 108, 109, 110\]. For this reason, it is important to anticipate the resources required for processing the images from these survey telescopes. Our focus is on the Intermediate Palomar Transient Factory (iPTF) \[111\], a successor of the PTF and a predecessor of the ZTF. We show that many spatial and temporal features, common to all similar workflows \[112, 113\], are most useful for predicting the resource requirements. Therefore, the insight developed in this work is relevant to many astronomical data analysis projects.
Advancements of the optics and the image capturing systems in the synoptic surveys have dramatically increased the volume of data captured and created an unprecedented demand for analyzing these images. For example, the iPTF project uses a large field camera with 11 active 2048x4096 CCDs that capture 7.8 square degrees on the sky every two minutes producing 70GB per night [114]. The ZTF uses the same telescope as iPTF but has an imager with a wider field-of-view and shorter exposure times yielding an order of magnitude more data in their search for young supernovae and fast-evolving transients. Due to the computing resources required, the data processing tasks are often completed in a computer center. In the case of iPTF, the images are transferred to the National Energy Research Scientific Computing Center (NERSC) and the transient discovery workflow is executed on the Cori supercomputer [113]. The iPTF workflow on Cori consists of three main parts: I/O, image subtraction (computation), and database matching (database queries). The execution time for these tasks within the analysis workflow varies with a number of characteristics of the image and system states.

In this chapter, we propose a machine learning prediction strategy for predicting the performance of the iPTF workflow workload. Our conclusions are not limited to the iPTF workflow. Instead, they are readily useable for the ZTF workflow since the transient discovery tasks, and image attributes of iPTF and ZTF are similar [112]. In addition, we have taken care to build the prediction models based on features in image analysis workflows. In particular, we design several features and select two prediction targets that indicate the workload of the workflow pipeline. The features and prediction targets allow us to predict workflow performance in real-time. The critical contribution of this work is a Bayesian network that represents the conditional correlation among the features.
and prediction targets. The Bayesian network allows us to filter out redundant features, which in turn makes the model more actionable and avoids overfitting problems. We use experimental results to support the correctness of the Bayesian network. The testing data performance of our prediction models trained with the selected features achieves strong correlations between prediction and actual values and NMSEs as low as 0.1.

The rest of this chapter is arranged as the following. In Section 7.1, we describe the related work for transient discovery. Later, we present our preprocessing, feature selection, and model design methods in Section 7.2. Finally, we present a case study of iPTF workflow data with our proposed approaches in Section 7.3.

7.1. Background

Gravitational wave (GW) detections, neutrino detections, and electromagnetic spectrum observations can reveal the death throes of stars \[107, 108, 115\]. However, the detailed mechanism of such stellar explosions is still an open question. Thus, many of the recent astronomical observations are focused on gathering data about transients to study the chaotic moments after stars explode \[116, 117, 118, 119, 120, 121, 122, 123, 124, 125\].

To study these transients, one set of telescopes perform synoptic surveys by repeatedly looking at a large fraction of the sky for any changes, while another set of telescopes, typically, many larger ones, will be directed to make more detailed observations about the newly discovered transients. In this process, the images gathered by the synoptic surveys need to be processed quickly so that the most promising candidates can be identified in near real-time for follow-on observations. Since there are only a limited number of large
telescopes that could complete the follow-up observations, the workflow for processing the synoptic survey images must identify the most interesting transients accurately and reliably [112]. Additionally, the astronomers are increasingly focusing on short-lived transients, such as gamma-ray bursts or GW counterparts that might last only hours, which requires image processing workflows to be completed in minutes. Thus, such data processing workflows are often quite complex and require a considerable amount of computing resources, which is why iPTF uses Cori at NERSC.

The iPTF workflow pipeline processes a high volume of images in parallel. It consists of I/O operations, computation for filtering out relevant objects in images with machine learning techniques, and database operations. Scheduling jobs for such a workflow on the supercomputer is challenging since the execution time of different stages depends on the input data properties. A comprehensive study of workloads on supercomputers has shown that large-scale applications on supercomputers have a similar workload within a short period [126]. However, they conclude that the workload of a pipeline is likely to vary across a long-time period. Merzky et al. argue for the same conclusion in their RADICAL-Pilot paper [127]. Existing methods for scheduling jobs on supercomputers need an estimation of job workloads. In the case of iPTF, we use the number of objects to be extracted from an image as an estimation of workflow. Armstrong et al. [128] study the problem of scheduling multiple tasks with different workloads. Tang et al. propose methods for scheduling supercomputer jobs with conflict goals: fairness, wait time, and system utilization [129] [130]. However, predicting workload is a challenging task. Users cannot accurately predict the workload of their programs, so scheduling algorithms may be given wrong workload information. A follow up of Tang’s work points out that the workload
estimation is usually inaccurate on supercomputers [131]. They propose a method for adjusting the expected execution time of user jobs for better performance. Nevertheless, predicting workload based on the known software task and data can give more accurate estimation because domain knowledge is taken into account. In this chapter, we take advantage of the domain-specific knowledge of iPTF.

The estimation of processing time based on workload is a challenging task since the execution time depends on both hardware conditions and software input data. The hardware condition is often unknown to users of supercomputers. However, we can infer it based on the execution time of recent jobs in the workflow pipeline. The software input data we are dealing with are iPTF images. Based on the available features of the images, we can predict the processing time of the bottleneck stage. We define the features and prediction model in the next section.

7.2. Design

The iPTF workflow pipeline has 40 checkpoints, labeled from 0 to 39, see an illustration in Fig. 7.5. We name the task between checkpoint $i$ and $i + 1$ to be the processing stage $i$, and, therefore the execution time of stage $i$ is the time difference between the time reported at checkpoint $i$ and that at checkpoint $i + 1$. Stage 0 of this workflow is to transfer the image captured by iPTF telescope to NERSC. The remaining stages of this workflow are executed on Cori [113, 112].
Figure 7.1. (a) Pearson correlation coefficients for pair-wise processing time of stages before removing I/O anomalies. (b) Pearson correlation coefficients for pair-wise processing time of stages after removing I/O anomalies. (c) Spearman correlation coefficients for pair-wise processing time of stages before removing I/O anomalies. (d) Spearman correlation coefficients for pair-wise processing time of stages after removing I/O anomalies.
7.2.1. Removing I/O anomalies

We discover that the processing time of stage 5, 10, 14, and 19 contain anomalies. These are all I/O stages of the pipeline. Their mean processing time is higher than their 90% quantile. Moreover, the pair-wise Pearson correlation coefficients are all greater than 0.9. Figure 7.1 (a) illustrates the pair-wise correlation coefficients of raw dataset. Spearman correlation coefficients, on the other hand, are not showing such a strong correlation in Figure 7.1 (c). Thus, we conclude that the strong linear correlation of these I/O stages results from a few large anomalous numerical values. The anomaly values of the I/O stages are up to hours, which is much larger than the typical I/O processing time of the program (less than 2 minutes). Therefore, we use the means of the I/O stage processing time as breakpoints. After we split the original dataset based on the mean of these four stages, approximately 10% of the dataset are classified as anomalous. The anomalous part shows a pair-wise linear correlation among the I/O stages close to 1, while the rest 90% of the regular dataset does not show a moderate pair-wise correlation. Figures 7.1 (b) and (d) illustrates the pair-wise correlations between different stages after removing I/O anomalies. This observation implies that the long execution time of these I/O stages occur together, from which it is reasonable to conclude that the I/O system is undergoing a severe degradation when these images passed through the pipeline. Consequently, we can confidently filter out these anomalies. These anomalies have timestamps in 2015, which was the first year that Cori was in service and are likely the product of various system instabilities. Our focus should be on the regular part of the dataset since such anomalies are not expected moving forward.
7.2.2. Prediction Target

![Stage processing time of all stages after removing all I/O anomalies.](image)

Figure 7.2. Stage processing time of all stages after removing all I/O anomalies.

Figure 7.2 illustrates the stage processing time distribution of all stages from 1 to 38. Stage 36 (between checkpoints 36 and 37) is the dominant stage. This stage is a database-intensive operation for associating candidates with local galaxies, a complex spatial query in Postgres using q3c [132]. It is a critical piece of information in prioritizing follow-up observations. Predicting the processing time of this stage allows the schedule of the number of compute nodes ahead of time. For example, if the predicted time is relatively large, a larger number of compute nodes can be reserved. The rest of the stages...
with relatively long processing time are 25, 29, and 31. Stage 25 is an image subtraction. Stages 29-31 are parts of the real-bogus (RB) machine learning classifier. The RB classifier removes the false-positive detections caused by nonlinearity of the detectors, astrometric misalignment, imperfect convolution kernels, Poisson noise of bright objects, cosmic rays, and many other factors. These three stages are computationally intensive operations.

There are two prediction targets for the workload of an image: One is the dominant stage 36 processing time, and the other is the number of celestial objects extracted. Stage 36 is the most time-consuming step of the entire pipeline. Dominant stage processing time is a direct measurement of workload. However, it is subjected to the variance of the supercomputer’s performance. The number of objects extracted determines the workload of specific operations. However, it is not available until the start of database stages, which is the checkpoint 33. Many stages, such as stage 25 and 27, which runs hotpants for image subtraction, have workloads dependent on the number of objects extracted. Moreover, the workloads of database stages are also proportional to the number of objects extracted. Thus, predicting the number of objects is equivalent to predicting the workload of multiple stages, which allows proper resource scheduling. Our goal is to predict the number of the object extracted at checkpoint 1, which is the start of the pipeline on Cori.

The proposed prediction targets are weakly correlated. Figure 7.3 (a) and (b) depict the boxplots of the processing time of stage 36 and the number of objects extracted in a 24-hour interval. From 9:00 AM to 7:00 PM, no iPTF image is taken because iPTF only operates at night. Different hours have a similar variance of these two prediction targets. Furthermore, we can observe that means of the number of objects extracted and the processing time of checkpoint 36 have correlations over these periods. Nevertheless,
these two prediction targets are not interchangeable because they only have 0.62 Pearson and 0.58 Spearman correlations at image granularity.

7.2.3. Feature selection

To predict either stage 36 processing time or objects extracted, we define attributes for making such prediction. There are two constraints for the prediction model to make a useful inference. Firstly, the input attributes of the prediction model should be ready before the output of prediction occur. For example, the moment when a model makes a prediction for stage 36, processing time must happen before the timestamp of checkpoint 36. Otherwise, the prediction result is useless for scheduling a computing resource since the program for running stage 36 is already started. Another constraint is that the prediction model should be light-weighted. Since the goal of this research is to schedule computational resources smartly, a heavy machine learning model that consumes larger resources is unacceptable.

Stage 36 is nearly at the end of the pipeline (39 stages in total), so it is possible to apply timings of stages from 0 to 35 before checkpoint 36 as prediction attributes. We denote this class of features as type 1. However, models for predicting the number of objects extracted does not have the access to type 1 features since this prediction is expected at checkpoint 1 in order to have a raw estimate of the entire workload. In addition, some attributes of the image are known in advance of stage 0, so we can directly apply them as model features for both of the prediction targets. This class of features is denoted as type 2. Table 7.1 illustrates all the image attributes and the checkpoints that they are available on the pipeline. For the image attributes, we denote the number of objects
Figure 7.3. (a) Average stage 36 processing time in 24 hours period. (b) Average objects extracted in 24 hours period. (c) Average stage 36 processing time in spatial on right ascension (horizontal) and declination (vertical) scale. (d) Average objects extracted in spatial on right ascension (horizontal) and declination (vertical) scale. Note the processing time roughly correlates with the position on the sky following the overdensity of stars in our own galaxy.

extracted as type 2a and the rest as type 2b. Type 2b does not include seeing and the
Table 7.1. This table shows the earliest checkpoint that an image attribute is available.

<table>
<thead>
<tr>
<th>Feature Name</th>
<th>Checkpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right Ascension</td>
<td>2</td>
</tr>
<tr>
<td>Declination</td>
<td>2</td>
</tr>
<tr>
<td>UTC time of start of observation</td>
<td>2</td>
</tr>
<tr>
<td>Julian Date for above</td>
<td>2</td>
</tr>
<tr>
<td>Airmass</td>
<td>2</td>
</tr>
<tr>
<td>PTF ID</td>
<td>2</td>
</tr>
<tr>
<td>Chip ID</td>
<td>2</td>
</tr>
<tr>
<td>Filter</td>
<td>2</td>
</tr>
<tr>
<td>Seeing</td>
<td>14</td>
</tr>
<tr>
<td>Limiting magnitude</td>
<td>14</td>
</tr>
<tr>
<td><strong># of objects extracted</strong></td>
<td><strong>33</strong></td>
</tr>
<tr>
<td># of objects saved</td>
<td>33</td>
</tr>
<tr>
<td>Ecliptic Longitude</td>
<td>2</td>
</tr>
<tr>
<td>Ecliptic Lattitude</td>
<td>2</td>
</tr>
<tr>
<td>Galactic Longitude</td>
<td>2</td>
</tr>
<tr>
<td>Galactic Lattitude</td>
<td>2</td>
</tr>
<tr>
<td>Extinction on sky</td>
<td>2</td>
</tr>
</tbody>
</table>

number of objects saved for predicting the number of objects extracted by our previous constraints.

In addition to attributes limited to a single image, we can infer the workload of the current images from the workload of other images that have been processed in the recent past. From a temporal point of view, we expect the nearest $k$ number of images that have been processed by the pipeline should inform the current workload. We notice that the cameras of iPTF rotate in the sky in a continuous fashion. However, on some rare occasions, they jump from one region to another. Thus, the nearest temporal neighbors may contain some spatial gaps. We plot the average processing time of stage 36 and the average objects extracted in Figure 7.3 (c) and (d) on $9 \times 9$ on right ascension and
Figures 7.4. iPTF prediction model feature summary. Type 1 features are the stage processing time of current images prior to checkpoint 36. Type 2 features are the attributes of current image. Category 2a is the number of objects extracted and category 2b contains other attributes of the images shown in Table 7.1. Type 3 features are the features provided by images in the spatiotemporal nearest neighborhood.

decelination grids. The heap maps have continuous colors in most of the regions, which indicate the similarity of the means of these attributes within a spatial neighborhood.
Therefore, we decided to construct spatiotemporal features. For example, the processing time of stage 36 of images taken from nearby coordinates within a short period can inform the prediction of stage 36 processing time of the current image. The temporal proximity implies a similar condition of a computing system, and the spatial proximity implies a similar number of objects and workloads. This type of feature is denoted as type 3. Due to the first constraint, all images in the most adjacent neighborhood must have reached checkpoint 37 before the start of stage 36 of the current image for predicting the stage 36 processing time. For the case of predicting the number of objects, the nearest spatiotemporal neighborhood must have passed checkpoint 33, according to Table 7.1.

![Figure 7.5. iPTF online prediction illustration. The circles indicate checkpoints of the workflow pipeline. The stages are denoted as arrows. The prediction of the number of objects is at checkpoint 1. The prediction of the stage 36 processing time is at checkpoint 36.](image)

By the first constraint, all attributes selected from adjacent images must be ready at the checkpoint 36 (for stage 36 processing time) and checkpoint 1 (for the number of objects extracted) of the current image. Figure 7.5 illustrates these two predictions in the workflow.

Figure 7.4 illustrates the features selected for predicting the number of objects and stage 36 processing time. Applying all the available features for making predictions is not an optimal strategy since some features can be independent of the prediction target in the presence of other features. In these scenarios, overfitting problems can occur. Although a
Figure 7.6. iPTF prediction model feature Bayesian network. Features that are directly connected to the red prediction targets are the one that should be used by the prediction models.
model yields low error after training, the testing performance is reduced, because random errors of the features and the prediction target that happen in the training data only are correlated. However, the correlations of errors may only exist in the training data. Therefore, the model produces false inference on testing features.

We propose a Bayesian network of features and prediction targets in Figure 7.6. A Bayesian network can show the conditional independence of variables. Features that are directly linked to a prediction target should be used by prediction models. The network is created by iterating through combinations of features and compare their testing errors of trained models. Although it is possible to construct a Bayesian network from a greedy algorithm, such as [133], we rely on domain knowledge of iPTF for the creation of the Bays network to avoid overfitting. For instance, the part in the dashed rectangle denotes the Spatiotemporal neighborhood of the current image. The rest of the figures are features related to the current image. These two parts should have identical internal topology since they are both image entities. Therefore, the search range of Bays network topology is reduced significantly with the incorporation of domain constraints. In the Section 7.3, we justify the structure of the Bayesian network by analyzing the prediction accuracy on testing data with different combinations of features.

We keep the directions of the arrows that satisfy the constraints of prediction mentioned in Section 7.2.3. In other words, if A points an arrow to B, the value of A is available before B. Thus, this network is a trimmed version designed for feature selections of our objectives. Our prediction targets are labeled in red. Features that are directly connected to the red targets are those that can build a model with the best prediction results. If there is a long path between a feature and the target, the correlation between the feature
and the target is expected to be weak, especially when other features are present on its path to the target.

### 7.2.4. Prediction model

We apply a multi-layer perceptron, a typical neural network architecture, also denoted as a fully connected network, for predicting target attributes from input attributes.

A fully connected network consists of layers. The first layer is the input layer, and the last layer is the output layer. There are some hidden layers that in-between the input and the output layer. Let $A$ be a $m \times n$ matrix describing the connections between layer $l_1$ of size $n$ and layer $l_2$ of size $m$ for a fully connected network. The $j^{th}$ row and $i^{th}$ column of $A$, $a_{j,i}$, is the weight from the $j^{th}$ neuron of $l_1$ to the $i^{th}$ neuron of $l_2$. Let $b$ be a bias vector of dimension $n$. The $i^{th}$ element of $b$ is the bias for the $i^{th}$ neuron of $l_2$. The output feature vector $v$ from $l_1$ has size $n$. Then $f(Av + b)$ is the output vector for layer $l_2$. Our prediction network has one input layer, three hidden layers, and one output layer. There are 1024 neurons in the first hidden layer and 256 neurons in the second and third hidden layers. The final layer has one neuron that outputs the regression result. The activation function is Relu $f(x) = \max(0, x)$.

An advantage of this neural network is that it is capable of simulating any functions [134]. For the problem size we are dealing with, the proposed network size is sufficient. Moreover, this parameter settings of the multi-layer perceptron satisfy the requirement of light-weighted machine learning model, since a prediction for one feature can finish within a few seconds on a regular desktop computer.
7.3. Experimental Results

We conduct experiments on a single Linux machine installed with a GTX 1080 and CUDA version 10.1.

For a real-world case study, we split the iPTF pipeline data\footnote{https://portal.nersc.gov/project/astro250/xswap/iptf/} into training data and testing data. The training data contains 90\% of the data with lower timestamps, and the testing data includes the other 10\%. With such a splitting strategy, predictions on the testing data are equivalent to real-time simulations. This splitting method is a simulation of a production run since the goal is to train historical data for predicting future results. In addition, we also present 10-fold cross-validation results for evaluation.

We train the neural network discussed in Section 7.2.4 with Stochastic Gradient Descent (SGD) for 200 epochs with mini-batch size 256. The learning rate is from 0.88 to 1, with a 0.1 momentum factor. This training strategy guarantees the convergence of all our experiments. We set the drop-out ratio per layer to be 0.33 to avoid the overfitting of the model\footnote{https://portal.nersc.gov/project/astro250/xswap/iptf/}. Training with an adaptive learning rate such as ADAM\footnote{https://portal.nersc.gov/project/astro250/xswap/iptf/} does not converge to a reasonable regression accuracy the same way as SGD. Besides, the final losses of the models are unstable with different initial parameters. The models trained with SGD, on the other hand, are stable with better accuracy results.

More complicated model architectures may yield better training results after convergence. However, the main goal of this chapter is to evaluate the proposed iPTF workflow features. Thus, we control the neural network model and training methods, treating the model as a black-box for regression, for comparing different choices of input features.
Root mean square error (RMSE), Normalized mean square error (NMSE), and correlation coefficients (Pearson and Spearman) are the evaluation metrics in this section. MSE evaluates the regression error with formula \( \frac{1}{N} \sum_{i=1}^{N} (\hat{f}(x_i) - y_i)^2 \) for prediction model \( \hat{f} \), input feature \( x \), and prediction output \( y \). RMSE is the square root of MSE. NMSE scales down the units of RMSE with formula \( \frac{1}{\bar{y}} \text{MSE} \). A good model should have small RMSE and NMSE. Correlation coefficients measure if the predictions have the same trend as the actual data in the parity graph. The metrics are between 0 and 1. A good model has large correlation coefficients. Pearson correlation takes the magnitude of difference into account. The Spearman correlation examines if the trend of two variables is the same or not. Although our training objective is to reduce regression error, the problem can be reduced to a classification problem. For example, we can classify the workload to be discrete results, such as small, medium, and large, for different ranges of the outputs. In this scenario, the absolute errors are less important due to the change of the error function. If the correlations between prediction and real measurements are large, the classification result should be mostly correct, since the prediction is likely to lie in the right region.

7.3.1. Baselines

We can set up the baselines that evaluate the performance of our model via RMSE. Testing RMSE should fall between a lower bound and an upper bound.

An upper bound of the prediction targets for RMSE is the standard deviation. If a prediction model always predicts the mean of the prediction target in the dataset, its RMSE is equivalent to the standard deviation. The estimated standard deviations from
the populations are 60.4 for the stage 36 processing time and 653 for the number of objects extracted.

A lower bound of the prediction targets for RMSE is the estimation of the prediction targets’ standard deviation within a batch. iPTF cameras take images in batches. Images within a batch are processed in parallel with the same checkpoint 0 start time. These images have very similar spatiotemporal attributes. A regression model based on our proposed features without overfitting should give the same results. However, there are still some non-negligible variances of workload within a single batch. These variances are unpredictable by the information given in the features for regression since the attributes are identical. Thus, the variance within batches can serve as a lower bound of prediction error. We estimate the average standard deviation within a batch for the stage 36 processing time as 20.4 and the number of objects extracted as 210 from all data.

7.3.2. Predicting stage 36 processing time

Discussed in Section 7.2.3, several different types of features can be applied to predict stage 36 processing time, as summarized in Figure 7.4. The input features have three categories. The first type is historical processing time from stage 0 to 35. The second category is the image attributes, including spatial coordinates. The final type (type 3a) is the stage 36 processing time of spatiotemporally nearby images in the pipeline.

The number of the nearest neighbors is a tunable parameter. We use 1 for simplicity. In this section, we demonstrate the individual effectiveness of these three types of features.

Table 7.2 illustrate the results of predicting stage 36 processing time with a different combination of features for ten-fold cross-validation. CV refers to the average of 10-fold
cross-validation results, and LF refers to the last fold (as testing data) of the cross-validation results. The values in the brackets are prediction evaluations on testing data. The values that are not in the brackets are prediction evaluations on training data.

From the first three rows, we can observe relatively large mean square errors and low correlation coefficients. Thus, using any of the three features alone, the regression models do not have reasonably good results on testing data.

Table 7.2. Accuracy results of iPTF prediction models for predicting checkpoint 36 timing.

<table>
<thead>
<tr>
<th>Row ID</th>
<th>Features</th>
<th>CV RMSE (CV NMSE)</th>
<th>CV RMSE (CV NMSE)</th>
<th>CV Pearson (CV Spearman)</th>
<th>LF RMSE (LF NMSE)</th>
<th>LF Pearson (LF Spearman)</th>
<th>LF RMSE (LF NMSE)</th>
<th>LF Pearson (LF Spearman)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>38.7 (45.1)</td>
<td>0.212 (0.303)</td>
<td>0.79 (0.66)</td>
<td>36.9 (51.0)</td>
<td>0.207 (0.230)</td>
<td>0.80 (0.65)</td>
<td>0.81 (0.69)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>29.2 (31.8)</td>
<td>0.135 (0.160)</td>
<td>0.88 (0.82)</td>
<td>28.1 (44.0)</td>
<td>0.129 (0.216)</td>
<td>0.88 (0.73)</td>
<td>0.93 (0.80)</td>
</tr>
<tr>
<td>3</td>
<td>3d</td>
<td>43.4 (42.4)</td>
<td>0.287 (0.280)</td>
<td>0.70 (0.63)</td>
<td>41.8 (55.1)</td>
<td>0.281 (0.317)</td>
<td>0.72 (0.50)</td>
<td>0.76 (0.56)</td>
</tr>
<tr>
<td>4</td>
<td>3a,3b,3c</td>
<td>37.4 (41.7)</td>
<td>0.221 (0.278)</td>
<td>0.79 (0.65)</td>
<td>36.2 (54.3)</td>
<td>0.223 (0.331)</td>
<td>0.80 (0.53)</td>
<td>0.83 (0.59)</td>
</tr>
<tr>
<td>5</td>
<td>1,2</td>
<td>24.0 (28.2)</td>
<td>0.088 (0.125)</td>
<td>0.92 (0.87)</td>
<td>22.6 (38.6)</td>
<td>0.084 (0.158)</td>
<td>0.93 (0.79)</td>
<td>0.96 (0.86)</td>
</tr>
<tr>
<td>6</td>
<td>1,3d</td>
<td>34.1 (36.8)</td>
<td>0.179 (0.213)</td>
<td>0.83 (0.75)</td>
<td>32.6 (47.5)</td>
<td>0.170 (0.201)</td>
<td>0.84 (0.71)</td>
<td>0.86 (0.75)</td>
</tr>
<tr>
<td>7</td>
<td>2,3d</td>
<td>28.0 (30.2)</td>
<td>0.124 (0.145)</td>
<td>0.89 (0.84)</td>
<td>26.7 (42.1)</td>
<td>0.118 (0.200)</td>
<td>0.90 (0.76)</td>
<td>0.94 (0.83)</td>
</tr>
<tr>
<td>8</td>
<td>1,2,3d</td>
<td>23.6 (27.0)</td>
<td>0.085 (0.114)</td>
<td>0.93 (0.88)</td>
<td>22.5 (36.0)</td>
<td>0.084 (0.137)</td>
<td>0.93 (0.82)</td>
<td>0.96 (0.88)</td>
</tr>
<tr>
<td>9</td>
<td>1,2,3a,3b,3c</td>
<td>23.9 (28.5)</td>
<td>0.087 (0.127)</td>
<td>0.92 (0.87)</td>
<td>22.6 (38.9)</td>
<td>0.084 (0.160)</td>
<td>0.93 (0.79)</td>
<td>0.96 (0.86)</td>
</tr>
<tr>
<td>10</td>
<td>1,2,3d,3a,3c</td>
<td>23.7 (27.8)</td>
<td>0.086 (0.122)</td>
<td>0.93 (0.88)</td>
<td>22.1 (36.3)</td>
<td>0.080 (0.139)</td>
<td>0.93 (0.82)</td>
<td>0.96 (0.88)</td>
</tr>
</tbody>
</table>
Table 7.3. Accuracy results of iPTF prediction models for predicting the number of objects extracted.

<table>
<thead>
<tr>
<th>Row ID</th>
<th>Features</th>
<th>CV RMSE</th>
<th>CV NMSE</th>
<th>CV Pearson</th>
<th>CV Spearman</th>
<th>LF RMSE</th>
<th>LF NMSE</th>
<th>LF Pearson</th>
<th>LF Spearman</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>2b</td>
<td>309.3</td>
<td>0.122</td>
<td>0.83</td>
<td>0.83</td>
<td>311.9</td>
<td>0.125</td>
<td>0.83</td>
<td>0.83</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(352.5)</td>
<td>(0.168)</td>
<td>(0.72)</td>
<td>(0.68)</td>
<td>(319.7)</td>
<td>(0.113)</td>
<td>(0.75)</td>
<td>(0.75)</td>
</tr>
<tr>
<td>12</td>
<td>3a</td>
<td>321.9</td>
<td>0.135</td>
<td>0.81</td>
<td>0.82</td>
<td>319.2</td>
<td>0.134</td>
<td>0.82</td>
<td>0.83</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(320.0)</td>
<td>(0.138)</td>
<td>(0.76)</td>
<td>(0.76)</td>
<td>(335.8)</td>
<td>(0.128)</td>
<td>(0.71)</td>
<td>(0.72)</td>
</tr>
<tr>
<td>13</td>
<td>3b</td>
<td>270.7</td>
<td>0.098</td>
<td>0.87</td>
<td>0.87</td>
<td>267.3</td>
<td>0.098</td>
<td>0.87</td>
<td>0.87</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(303.2)</td>
<td>(0.127)</td>
<td>(0.78)</td>
<td>(0.77)</td>
<td>(318.5)</td>
<td>(0.124)</td>
<td>(0.75)</td>
<td>(0.75)</td>
</tr>
<tr>
<td>14</td>
<td>3c,3d</td>
<td>351.2</td>
<td>0.176</td>
<td>0.77</td>
<td>0.78</td>
<td>344.8</td>
<td>0.166</td>
<td>0.78</td>
<td>0.78</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(370.5)</td>
<td>(0.200)</td>
<td>(0.67)</td>
<td>(0.65)</td>
<td>(391.2)</td>
<td>(0.164)</td>
<td>(0.63)</td>
<td>(0.63)</td>
</tr>
<tr>
<td>15</td>
<td>2b,3a</td>
<td>262.9</td>
<td>0.090</td>
<td>0.88</td>
<td>0.88</td>
<td>260.3</td>
<td>0.091</td>
<td>0.88</td>
<td>0.88</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(288.9)</td>
<td>(0.113)</td>
<td>(0.81)</td>
<td>(0.79)</td>
<td>(287.6)</td>
<td>(0.095)</td>
<td>(0.80)</td>
<td>(0.80)</td>
</tr>
<tr>
<td>16</td>
<td>2b,3b</td>
<td>225.6</td>
<td>0.067</td>
<td>0.91</td>
<td>0.91</td>
<td>224.6</td>
<td>0.070</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(265.1)</td>
<td>(0.097)</td>
<td>(0.84)</td>
<td>(0.82)</td>
<td>(263.4)</td>
<td>(0.084)</td>
<td>(0.83)</td>
<td>(0.83)</td>
</tr>
<tr>
<td>17</td>
<td>3a,3b</td>
<td>270.9</td>
<td>0.098</td>
<td>0.87</td>
<td>0.87</td>
<td>267.9</td>
<td>0.098</td>
<td>0.87</td>
<td>0.87</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(301.8)</td>
<td>(0.126)</td>
<td>(0.78)</td>
<td>(0.77)</td>
<td>(316.8)</td>
<td>(0.123)</td>
<td>(0.75)</td>
<td>(0.75)</td>
</tr>
<tr>
<td>18</td>
<td>2b,3a, 3b</td>
<td>226.3</td>
<td>0.067</td>
<td>0.91</td>
<td>0.91</td>
<td>224.5</td>
<td>0.068</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(266.9)</td>
<td>(0.098)</td>
<td>(0.84)</td>
<td>(0.82)</td>
<td>(265.0)</td>
<td>(0.083)</td>
<td>(0.83)</td>
<td>(0.83)</td>
</tr>
<tr>
<td>19</td>
<td>2b,3a, 3c,3d</td>
<td>250.5</td>
<td>0.084</td>
<td>0.89</td>
<td>0.89</td>
<td>251.2</td>
<td>0.087</td>
<td>0.89</td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(284.4)</td>
<td>(0.114)</td>
<td>(0.81)</td>
<td>(0.79)</td>
<td>(284.7)</td>
<td>(0.098)</td>
<td>(0.79)</td>
<td>(0.79)</td>
</tr>
<tr>
<td>20</td>
<td>2b,3a, 3b,3c,3d</td>
<td>229.6</td>
<td>0.071</td>
<td>0.91</td>
<td>0.90</td>
<td>226.7</td>
<td>0.070</td>
<td>0.91</td>
<td>0.90</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(269.2)</td>
<td>(0.102)</td>
<td>(0.83)</td>
<td>(0.82)</td>
<td>(269.8)</td>
<td>(0.089)</td>
<td>(0.82)</td>
<td>(0.82)</td>
</tr>
</tbody>
</table>

The fourth to the sixth rows contain results with pair-wise combinations of three features. The historical processing time from stage 0 to 35 of the current image (type 1), combined with the image attributes (type 2), gives better testing results than replacing type 2 features with spatiotemporally nearby image processing time (type 3d). The combination of type 2 and 3d features yields similar results as the combination of type 1 and 2 features. Thus, we observe that image attributes can improve the overall performance significantly when added to either type 1 and 3d features. Type 1 and 3d images
combined (row 6), on the other hand, demonstrate improvement compared with models trained with either of them. This observation is expected because both features are timing features. They are likely correlated with each other. Nevertheless, joining them together can moderately improve testing accuracy compared with the results predicted by each of them separately. Finally, if all these three features are combined, the overall regression performance on testing data is the best (row 7). Therefore, the direct connections to stage 36 processing time in Figure 7.6 is verified.

Rows 9 and 10 of figure Table 7.2 illustrate the reasons why the attributes (types 3a, 3b) and stage processing time 0-35 of the spatiotemporally nearest neighborhood (type 3c) are not useful for predicting the stage processing time of the current image given the stage 36 processing of the neighborhood (type 3d). We can observe that the testing metrics in rows 9 and 10 are worse than those of row 8. Moreover, if we compare the results of rows 8 and 9, it is clear that the correlation between the prediction feature is stronger to the nearest neighbors’ stage 36 processing time than to the rest of type 3 attributes. Consequently, type 3d should be selected as a feature for prediction, instead of the rest of the features.

7.3.3. Predicting the number of objects extracted

Details of selecting the features of the number of objects extracted are presented in Figure 7.4. Prediction of the number of objects extracted takes place at the beginning of the pipeline. Thus, the type 1 features adopted for predicting stage 36 processing time cannot be used in this case, since the checkpoints for computing stage processing time have not
reached. All image attributes excluding the seeing, limiting magnitude, and objects extracted/saved, on the other hand, are available at the start of the pipeline. Furthermore, attributes from spatiotemporally adjacent images are available at checkpoint 2 if we carefully filter the neighborhood $N$ with constraint $\{n.\text{checkpoint}_{39} < x.\text{checkpoint}_1\} \forall n \in N$ for current image $x$. Similar to predicting stage 36 processing time, the attribute to be predicted (the number of objects extracted) is selected from the neighborhood. Furthermore, we also employ the stage processing time and other image attributes of its spatiotemporally nearest neighborhood. These features are type 3 features. From the Bayesian network in Figure 7.6, we expect types 2b, 3a, and 3b to have a strong correlation with the prediction objective.

Table 7.3 illustrate the results of predicting the number of objects extracted by our real-time prediction algorithm on testing data. Rows 11-13 illustrate the prediction results of using the attributes of the current image (type 2b), the number of objects extracted (type 3a), and other attributes (type 3b) of the spatiotemporally nearest neighborhood of the current image independently. Row 14 demonstrates the prediction accuracy using the stage processing time of spatiotemporally nearest neighborhood (type 3c,3d) that are not directly connected to the prediction target in Figure 7.6. Results in row 14 are worse than rows 11, 12, and 13, which is expected from the Bayesian network since the correlations of features in row 14 to the prediction target is weaker. Rows 15-17 demonstrate the results of using different combinations of pair-wise features in rows 11-13. It is apparent that the results of pair-wise combinations of features are better than results using each independent feature. Row 18 illustrates the prediction results of features that are directly connected
to the prediction targets in Figure 7.6. We can observe further accuracy improvements in both testing and training data compared with rows 15-17.

Rows 19 and 20 of Table 7.3 prove the point that the stage processing time of spatiotemporally nearest neighborhood of the current image is independent of the number of objects extracted of current image given types 3b, 3a, and 3b. Furthermore, type 3b has a stronger correlation with the prediction target compared with types 3c and 3d. Therefore, type 3b should be directly connected to the prediction target, as shown in the Bays network in Figure 7.6.

7.4. Conclusion

We study the features for predicting the workload of the iPTF workflow pipeline. I/O anomalies of the iPTF workflow pipeline are filtered out. The proposed features and prediction targets of workloads are summarized into a Bayesian network that shows the conditional dependency of features. The selected features from the Bayesian network achieve prediction accuracy close to the lower bound defined by the randomness within data. Our spatiotemporal feature construction approach is not limited to the iPTF survey.

In the future, we plan to apply our predictions of workloads to schedule resources for the ZTF survey. A customized scheduling approach for the ZTF workflow pipeline can further improve the throughput of the system.
CHAPTER 8

A Filtering-based Clustering Algorithm for Improving Spatiotemporal Kriging Interpolation Accuracy

Geostatistical interpolation is the process that uses existing data and statistical models as inputs to predict data in unobserved spatiotemporal contexts as output. Kriging \[137\] is a well-known geostatistical interpolation method that minimizes the mean square prediction error. Everyday weather forecasting, environmental hazard prediction, and mineral mining are all its application domains. For example, Holdaway \[138\] has modeled a variogram for predicting monthly U.S temperature. Noel \[139\] has modeled Piezometric-Head data in the Wolfcamp Aquifer for predicting heavy metal pollution levels.

Kriging interpolation algorithm requires statistical assumptions about data. Firstly, Kriging interpolation assumes intrinsic stationarity in data. It means that a semivariogram, a function of covariance between two points, depends on the displacement vector between them rather than their absolute coordinates. If data is not intrinsic stationary, Kriging interpolation will have poor accuracy \[140\]. Another commonly used assumption for Kriging is isotropy. Isotropic data implies that covariance functions with respect to the distance between data points in all directions are the same, which in turn simplifies the computation of Kriging interpolation. Unfortunately, real-world datasets usually do not obey these two assumptions, so Kriging interpolation accuracy is poor if these two
properties are blindly assumed. A strategy that improves Kriging interpolation accuracy is to apply a non-parametric analysis of data using randomized subsampling \cite{141} when a dataset does not hold these two properties. However, this approach requires human interaction. Moreover, because non-parametric analyses with subsampling use only a subset of data, there are still concerns for final Kriging interpolation accuracy.

In this chapter, we present a filtering-based data clustering algorithm that is designed to improve Kriging interpolation accuracy when data does not obey essential statistical properties. Input data points are divided into clusters based on the similarity function of Kriging interpolation error with respect to semivariogram models specified by users. Because only data within the same cluster are used to interpolate results, Kriging interpolation accuracy does not suffer from invalid statistical assumptions for the overall dataset. Our clustering algorithm will find the number of clusters without user interaction. The algorithm has two phases: filtering and reinforcement. Clusters are formed by minimizing clustering-based Kriging interpolation accuracy while maximizing the number of points in the individual cluster. Maximizing individual cluster size not only prevents the algorithm from producing trivial results but also ensures that clustering-based Kriging uses as much data as possible for interpolation.

We use two real-world datasets to evaluate our algorithm. Comparisons to traditional Kriging are made using two error criteria: normalized mean square error (NMSE) and $\chi^2$ test statistics for normalized deviation measurement. Our clustering-based Kriging has reduced NMSE by more than 50% for both datasets compared to NMSE produced Kriging without clustering. Moreover, only our clustering-based Kriging has $\chi^2$ test statistics less
than 1% significance level while Kriging without clustering has the $\chi^2$ test statistics at least greater than 10% significance level.

The rest of this chapter is arranged as the following. Firstly, we present the ordinary Kriging interpolation algorithm in Section 8.1. Secondly, we illustrate the design of our proposed filter-based clustering algorithm in Section 8.2. Thirdly, we discuss the implementation of the proposed algorithm using both caching and heuristics for improving computational performance in Section 8.3. Finally, we demonstrate evaluations of the proposed algorithm with two real-world datasets using two error measurements in Section 8.4.

8.1. Kriging Algorithm

There are three variants of Kriging: Simple Kriging, ordinary Kriging, and universal Kriging [142]. The type of Kriging used in this chapter is ordinary Kriging. In this section, we describe ordinary Kriging presented in [140] based on variogram function.

Given $n$ input data points consist of $k$ dimensional coordinates $S = \{s_1, ..., s_n\}$, their associated physical attributes $\{z_1, ..., z_n\}$, a variogram model $\gamma$ for covariance between data points, Kriging algorithm is a mapping $Z : \mathbb{R}^k \rightarrow \mathbb{R}$ that predicts the physical attribute $Z(s_0)$ by $\sum_{i=1}^{n} w_i Z(s_i)$ at a new location $s_0$, whereas $Z(s_i) = z_i \forall s_i \in S$ by input definition. Weights $W = \{w_1, ..., w_n\}$ are assigned such that the mean square error (MSE) of $\hat{Z}(s_0)$ is minimized subject to constraint $\sum_{i=1}^{n} w_i = 1$. With intrinsic stationarity assumption, semivariogram function has the property $\gamma(h) = \frac{1}{2} E(Z(s+h) - Z(s)) \forall h \in \mathbb{R}^k$. As a result, we can formulate a dual optimization problem with the following Lagrange
function.

\[ L(S, W, \lambda) = MSE(\hat{Z}(s_0)) - \lambda(\sum_{i=1}^{n} w_i - 1) \]

\[ = 2\sum_{i=1}^{n} w_i \gamma(s_0 - s_i) - \lambda(\sum_{i=1}^{n} w_i - 1) - \sum_{i=1}^{n} \sum_{j=1}^{n} w_i w_j \gamma(s_i - s_j) \]

Solving the Lagrange function with the first order condition, we can have the following linear system.

\[ \frac{dL}{dw_i} = 2w_i \gamma(s_0 - s_i) - 2\sum_{j=1}^{n} w_i \gamma(s_j - s_i) - \lambda \]

\[ \frac{dL}{d\lambda} = 1 - \sum_{i=1}^{n} w_i \]

Setting the derivatives equal to zero, we can obtain a linear system of \( n + 1 \) equations.

Let \( \Gamma \) be an \((n + 1) \times (n + 1)\) dimension matrix, where \( \Gamma[i, j] = \gamma(s_i - s_j) \forall i, j \in [n] \), \( \Gamma[n + 1, n + 1] = 0 \), and the rest of entries are 1. Let \( m \) be an \( n \times 1 \) dimensional column vector with \( m[i] = \gamma(s_0 - s_i) \forall i \in [n] \) and \( m[n + 1] = 1 \). Let \( w = (w_1, ..., w_n, \frac{\lambda}{2})^T \).

Sherman [140] formulates the linear system as \( \Gamma w = m \), so assigning Kriging interpolation weights is reduced to solving the linear system and retrieving weights \( w \). The variance of Kriging is \( w^T m \).

### 8.1.1. Spatio-temporal Kriging

Although Kriging was originally formulated to predict physical attributes of spatial data, spatiotemporal data interpolation such as weather forecast can also apply Kriging. For example, in [138], a spatiotemporal variogram model is formulated to interpolate future
temperatures in St. Paul metropolitan area using recent temperatures of all cities in Minnesota State by Kriging. The inputs are spatiotemporal coordinates of cities in Minnesota State \( \{s_i = (x_i, y_i, t_i)\} \) for location \((x_i, y_i)\) and time stamp \(t_i\), next day’s time stamp, and the spatial coordinates of St. Paul metropolitan area. Moreover, each \(s_i\) is associated with an attribute \(z_i\), representing temperature at location \((x_i, y_i)\) and time stamp \(t_i\). The output is the estimated next day’s temperature in St. Paul metropolitan area. An efficient implementation of real-time spatiotemporal Kriging for solving this type of problem is in [143].

### 8.1.2. Clustering-based Kriging

Abedini, Nasseri, and Ansari exploit data preprocessing using the K-means clustering algorithm for Kriging interpolation [144]. Firstly, the algorithm separates data points into clusters by using K-means clustering. Then, the prediction for physical attribute \(\hat{Z}(s_i)\) of a data point \(s_i\) is interpolated via other members in the same cluster. Finally, the accuracy of the algorithm is evaluated by using normalized mean square error (NMSE), which is defined as the following equation.

\[
\frac{1}{s^2 n} \sum_{i=1}^{n} (\hat{Z}(s_i) - Z(s_i))^2.
\]

\(s^2\) is the sample variance. NMSE not only measures the actual square error but also takes the sample variance into account.

K-means clustering-based Kriging has two challenges to be addressed. Firstly, the algorithm requires the number of clusters to be determined in advance. For large spatiotemporal datasets, trying a large number of parameter \(k\) is expensive. Human effort is
also required to compare different types of error measurements for choosing an optimal k. Secondly, the algorithm only considers the distance between pair-wise data points without taking the covariance of physical attributes into account. Thus, the algorithm may not necessarily minimize the Kriging interpolation error.

8.2. Design

We propose a top-down based clustering algorithm that improves Kriging interpolation accuracy within each output cluster. A cluster that does not violate an error constraint specified by users is called a consistent cluster, which is formally defined in definition 3.1. The algorithm is an optimization problem that maximizes the size of individual clusters without violating consistent constraints for clusters. Maximizing the size of clusters avoids trivial output, which means that one data point corresponds to one cluster. Moreover, it ensures that clustering-based Kriging uses as much consistent data as possible for interpolation, which in turn maintains precision for Kriging interpolation.

Kriging requires a fitted variogram model to be specified in advance. In [145], it has been shown that weighted least square (WLS) is feasible for most common models such as the exponential model and spherical model. The algorithm assumes that a predefined variogram model remains invariant.

Initially, all data points are inserted into a single cluster. Starting from line 3, the algorithm iteratively separates data points into clusters with two phases: filtering phase and reinforcement phase. Points that are inconsistent with the rest of the points in the same cluster are filtered out at the filtering phase. A new cluster will be initialized to hold the filtered points. Because the algorithm checks consistency by using all remaining
Algorithm 14: Filter-based clustering algorithm

Data: $s_1, ..., s_n, Z(s_1), ..., Z(s_n)$, a threshold, and a variogram model $M$

Result: A queue of clusters $Q$, each point $s_i$ belongs to a unique cluster.

1. $C \leftarrow \{s_1, ..., s_n\}$;
2. $Q \leftarrow \{C\}$;
3. while $(C \leftarrow \text{next}_\text{element}(Q)) \neq \emptyset$ do
4.     $C' \leftarrow \text{new cluster};$
5.     converge $\leftarrow$ false;
6.     while converge == False do
7.         converge $\leftarrow$ true;
8.         for $s \in C$ do
9.             converge $\leftarrow$ converge $\land$ Filter($s, C, C', \text{threshold}, M$);
10.        end
11.     end
12.     converge $\leftarrow$ false;
13.     while converge == False do
14.         converge $\leftarrow$ true;
15.         for $s \in C$ do
16.             converge $\leftarrow$ converge $\land$ Reinforce($s, C, C', \text{threshold}, M$);
17.        end
18.     end
19. Remove all elements in $C'$ from $C$;
20. if !is_empty($C'$) then
21.     Insert($Q, C'$);
22. end
23. end

data points in the original cluster, it may remove points that are consistent with the original cluster at the end of the filtering phase. Therefore, it has to reinforce the cluster by adding falsely removed points back at the reinforcement phase. Figure 8.1 illustrates the two phases.
Algorithm 15: Filtering

Data: point \( s \), cluster \( C \), where \( s \in C, C' \), a threshold, and a variogram model \( M \)

Result: if the point is consistent to the cluster

1. Remove \((s, C)\);
2. \( t \leftarrow \text{normalized_kriging_error}(s, C, M) \);
3. if \( |t| > \text{threshold} \) then
   4. Add \((s, C')\);
   5. Return False;
   6. else
      7. Add \((s, C)\);
      8. Return True;
   9. end

Algorithm 16: Reinforcement

Data: point \( s \), cluster \( C \), where \( s \in C, C' \), a threshold, and a variogram model \( M \)

Result: if the cluster is consistent

1. Add \((s, C)\);
2. if \( \text{consistent}(C, \text{threshold}, M) \) then
   3. Remove \((s, C')\);
   4. Return False;
   5. else
      6. Remove \((s, C)\);
      7. Return True;
  8. end

8.2.1. Filtering Phase

The first inner while loop from line 6 to line 11 of algorithm 1 iteratively filter out any point that has a large discrepancy between its Kriging interpolated physical attributes based on other points in the same cluster and its actual physical attribute. Points are removed from the cluster in a Gauss-Seidel style, which iterates through the cluster and removes points without taking previously removed points into account.
Algorithm 2 (filter function) returns a Boolean. It calls the function normalized kriging error. This function uses other elements in $C$ to predict the physical attribute of $s$ by the Kriging interpolation described in section 2. Then the difference between the predicted value and the actual value of $Z(s)$ is normalized by dividing Kriging variance. The result is referred to as the normalized Kriging error. Consequently, the threshold can be chosen from a significant value in a standard normal distribution. Selecting a threshold from a standard distribution has the advantage of controlling the degree of confidence that a user requires for clustering-based Kriging interpolation accuracy. If the threshold is exceeded, the algorithm will return false and the convergence variable is set to be false. In this case, another round of filtering for cluster $C$ will be triggered because the cluster has not converged yet.

8.2.2. Reinforcement Phase

In the second inner while loop from line 13 to line 18 of algorithm 1, the algorithm iteratively attempts to add removed points back to the original cluster if the consistency constraint is not violated. Hence the size of the cluster is maximized subject to the Kriging consistency constraint. Kriging cluster consistency, defined in definition 3.1, states that the normalized Kriging error should be less than the threshold for every point in all clusters explicitly.

Algorithm 3 (reinforce function) returns a Boolean. It inserts a point $s$ back to a cluster $C$ and tests the Kriging consistency of $C$. This process computes the normalized kriging error for all points in $C$. If any of the points has normalized Kriging error exceeding
the threshold, the consistency test fails and the point is rejected to be inserted back to
the cluster $C$ for the current round of reinforcement.

### 8.2.3. Convergence Analysis

Clustering is defined to be a partition of objects into groups such that objects within
every group are as correlated as possible and objects between groups are as less correlated
as possible. The term correlation defined for our clustering algorithm is based on Krig-
ing interpolation accuracy: Any point within a cluster should be estimated by Kriging
interpolation using other points in the same cluster with low normalized Kriging error.

**Definition 3.** A cluster $C$ is consistent if and only if the Kriging interpolation of any
element in $C$ using the rest of elements in $C$ does not have a normalized Kriging error
greater than the threshold.

**Definition 4.** A converged set of clusters $Q = \{C_1, ..., C_k\}$ satisfies: For any $2 \leq j \leq
k$ and $C_i \in Q$ such that $j > i$, $C_i$ is not consistent if any of points in $C_j$ is inserted into
$C_i$.

**Proposition 1.** The output of algorithm 1 is a converged set of clusters.

**Proof.** It is possible to prove this statement by induction on the number of iterations
for the loop at line 3.

*Base case:* At the end of the first iteration, $C$ has been filtered and reinforced. $\forall s \in C'$,
because the reinforcement phase is finished, it is impossible to put $s$ back to $C$, which is
the only element in $Q$. 
Figure 8.1. An illustration of filter algorithm. The left side contains clusters that are filtered and reinforced. The right side shows how a cluster is filtered and reinforced. The remaining points at the end of the process enter the next round.
**Inductive assumption:** Suppose that at the end of $k^{th}$ step, $\forall s \in C'$, $\not\exists A \in Q$ such that $A$ is consistent after $s$ is inserted into it.

**Inductive step:** Consider the $(k+1)^{th}$ step, $C$ is the cluster created in the last iteration. Because we cannot filter more elements than the number of elements in $C$, $C'$ at the end of this iteration strictly contained in the input $C$. By inductive assumption, $s \in C'$ is inconsistent with the rest of the clusters in $Q$. However, because of the way how the algorithm reinforces the cluster $C$, any element in $C'$ must not be consistent with $C$ in the end. Otherwise, they would have been put back to $C$. As a result, the inductive assumption is true at $(k + 1)^{th}$ step. $\square$

Although we have shown the final result is a stable solution for maximizing cluster size subject to threshold constraint, a question remains is how the order of data affects the rate of convergence. The strategy used in our implementation is to shuffle data in advance. However, in the future, it is interesting to find out how to align data so that filtering and reinforcement are faster.

**8.2.4. Complexity Analysis**

Kriging interpolation algorithm is the core of filtering and reinforcement phases. In section 2, we have shown that the Kriging interpolation algorithm can be reduced to solving a linear system of equations. In our implementation, we use a lower upper permutation (LUP) approach to solve the linear system. The solution is exact, but the complexity is bounded by $O(n^3)$. However, other solvers can speed up performance. For example, in \cite{146} and \cite{147}, there are Kriging solvers that solve the linear system reduced from
Kriging interpolation algorithm efficiently. In the following analysis, we treat this computation as Kernel function and focus on analyzing the complexity of Algorithm 1 at a high level, namely the number of calls to the kernel functions \texttt{normalized\_kriging\_error} and \texttt{consistent}.

A round of filtering and reinforcement at least reduces the size of data to be filtered by 1. Otherwise the outer loop at line 3 breaks, so the outer loop is executed at most by \( n - 1 \) times. Consider the filtering phase, the worst case is that we filter 1 point per round from \( C \) to \( C' \), which costs \( O(n^2) \). For the reinforcement phase, it is also possible that points are reinforced back to the \( C \) 1 point per round. which costs \( O(n^2) \). Therefore, the overall complexity is \( O(n^3) \).

### 8.3. Implementation

We designed two approaches for improving computational performance: Caching and Heuristics. These two approaches do not change the result of experiments. Hence there is no accuracy loss. The entire algorithm was implemented in C language.

#### 8.3.1. Caching for Kriging Interpolation

When the filtering phase is converging, only a few points in the cluster are filtered out per round. Therefore, many points in the cluster still have the same neighbors as before. Thus, there is no need to compute normalized Kriging error again during the next round of filtering for those points if we can store the result of \texttt{normalized\_kriging\_error} for each data point. If no neighbors have been filtered out in the previous round by
checking the number of neighbors, the algorithm can simply read out the result of \texttt{normalized_kriging_error} without computing it again.

Similarly for reinforcement, when a point is put back into $C$, which is the cluster it has been filtered out from, the algorithm can check the consistency of the cluster using cached results for points that have the same neighbors as before. In addition, if reinforcement succeeds for a point, the cached result can be used for the next round of reinforcement. Otherwise, the caches of points that belong to the neighbor of the reinforced point should be poisoned for correctness.

### 8.3.2. Heuristics for Reinforcement

During reinforcement, the algorithm checks if $C$ is consistent after adding a point from $C'$ to $C$. Checking consistency requires computing normalized Kriging error for each point in the cluster, which is equivalent to a Jacobi style filtering phase. A trick can be used to report inconsistency faster. We can check the consistency of points that are more likely to violate the threshold constraint first. If they violated the constraint, there is no need to check the rest of the points.

Firstly, a point that violates the threshold constraint is likely to be the point that is reinforced because it has been filtered out in the filtering phase at the current iteration. We can treat this point as a special case and revise its consistency to the cluster before all other points. Secondly, we can wisely use the caching information in the previous round. For those points that already have high errors, their errors are likely to exceed the threshold than points with low errors when a new point is inserted into the cluster that they belong to. This estimation is a heuristic because the cached result may not be correct
due to cache poisoning. However, it provides a raw approximation for normalized Kriging errors cheaply given the cluster size is large. We can sort the order of reinforcement in descending order based on cached normalized Kriging error and check the consistency of points in this order. The extra cost is $O(n \log n)$ on average for quicksort. However, it may save many $O(n^3)$ operations for solving linear systems.

8.4. Experimental Results

We used two datasets in different applications to evaluate our algorithm. The first SOCR dataset is a classical dataset for testing Kriging interpolation accuracy. We will compare our filtering-based Kriging to K-means clustering-based Kriging and Kriging without clustering in terms of accuracy. The second IGRA dataset is a spatiotemporal dataset with a regular time interval. We will fit a variogram model step by step and justify that our clustering-based Kriging produces lower errors.

8.4.1. Quality Measurements

There are two measurements for evaluating the quality of spatiotemporal clusters in the following experiments. The first measurement is the NMSE value, which is defined in section 2.2. It is a ratio between the sum of squared leave-one-out cross-validation error and sample variance. The second measurement is $\chi^2$ test statistics. We compute the normalized Kriging error $\frac{(\hat{s}_i - s_i)}{\sigma}$ using leave-one-out cross-validation for all points in every cluster. The sum of squares of these errors is the $\chi^2$ test statistics. For both measurements, the smaller the values, the better the quality of clusters.
8.4.2. SOCR Data

SOCR dataset contains 85 data points, distributed near the south border of United States. Each data point has a 2 dimensional coordinate and a physical attribute that describes the water pollution level at the location. Because the dataset contains strong anisotropy, it is a challenging spatial dataset for Kriging. The dataset also contains the nugget effect, which indicates that the covariance of two points that are very close to each other is a constant greater than 0.

Noel fitted the parameters for NE-SW direction and NW-SE direction with geometrical anisotropy model derived by Journel and Huijbregts as the following.

\[
\gamma(h, \phi) = 14000 + 38h^2 \cos^2 \left( \frac{\pi}{4} - \phi \right) + 15h^2 \cos^2 \left( \frac{\pi}{4} + \phi \right)
\]

The output of algorithm 1 successfully divided the data points into 5 clusters. The last two clusters contain only three points, so they are treated as noise. The experimental results are shown in Table 8.1. The t-test p-value is a paired sample t-test for absolute Kriging errors between each of Kriging interpolation methods and Kriging interpolation without clustering at every data point. We reproduced K-means clustering-based Kriging with k=6 in, which is the best result they claimed, using K-means implementation in. Our filter algorithm reduced NMSE more than the K-means algorithm presented in. Furthermore, although K-means clustering-based Kriging reduced NMSE, there is no significant improvement for both absolute errors and normalized errors by observing the t-test p-value column and \(\chi^2\) column respectively. In addition, the 1% lower tile of \(\chi^2\) distribution with degree of freedom 85 is 57.63. As a result, only our filtering-based

\(^1http://wiki.stat.ucla.edu/socr/index.php/SOCR_061708_NC_Data_Aquifer\)
clustering algorithm has reduced errors that pass both $\chi^2$ test and paired sample t-test with a high confidence level.

<table>
<thead>
<tr>
<th>Method</th>
<th>cluster size</th>
<th>NMSE</th>
<th>$\chi^2$</th>
<th>t-test p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unclustered</td>
<td>1</td>
<td>0.24</td>
<td>197.75</td>
<td>0.5</td>
</tr>
<tr>
<td>K-means</td>
<td>6</td>
<td>0.11</td>
<td>161.88</td>
<td>0.3252</td>
</tr>
<tr>
<td>Filtering</td>
<td>5</td>
<td>0.048</td>
<td>22.53</td>
<td>$9.541E^{-9}$</td>
</tr>
</tbody>
</table>

8.4.3. IGRA Data

IGRA dataset\(^2\) contains sensor data collected by 67 NOAA stations across the United States in real-time. Every data point contains two-dimensional spatial coordinates for longitude and latitude, a physical attribute for atmosphere temperature, and a timestamp. The update interval for atmosphere temperature between two timestamps is 12 hours for all stations. This dataset is challenging because of anisotropy. An illustration of anisotropy with maximum Kriging range equals 30 is shown in Figure 8.2a. The variogram in the East direction appears to be different from other directions.

Similar to [138], we consider the monthly temperature of all stations. The semi-variance plot for the average of all stations in January 2014 with 4620 data points is shown in Figure 8.2b.

In Figure 8.2b, when $h > 30$, the variogram oscillates as $h$ increases. This oscillation indicates that the atmosphere temperatures between two locations that are far from each other are weakly correlated. Similar to [150], which uses nearest neighbors for Kriging interpolation, we apply local Kriging based on spatial range. The limit for maximum

\(^2\)http://www.ncdc.noaa.gov/data-access/weather-balloon-data
distance is set to be 30. There is also a sign of nugget effect for semivariance, starting from approximately 50. Using geoR \cite{151}, we fitted an exponential variogram model illustrated in Figure 8.2c.

$$\gamma(h) = 54.84102 + 153.71639 \exp(-0.05462576 ||h||)$$

We set the threshold to be 0.6, which is the threshold that gives reasonable size and number of clusters by several trials, we apply our clustering algorithm to IGRA atmosphere data at 67 stations in January 2014. The result is compared with Kriging without clustering. At every timestamp, the number of clusters is at most 6, with the last one or two clusters containing noise filtered in the end. Therefore, the results are reasonable because the algorithm does not reduce errors by dividing stations into trivial partitions.

In Figure 8.2d, the $\chi^2$ test statistics for both filtering-based clustering and Kriging without clustering are illustrated. For any given timestamp, Kriging with clustering has $\chi^2$ test statistics much smaller than Kriging without clustering. Moreover, the NMSEs span from 0.08 to 3.8. However, the high NMSEs are due to the noisy clusters filtered in the end. If we do not consider clusters with size less than 5, the NMSE is on average 0.27, with some outstanding time stamps at 1 or 2 for small clusters. Kriging without clustering has NMSE spanning from 6 to 18. These results indicate that Kriging interpolation accuracy based on clusters produced by our algorithm is better than Kriging interpolation accuracy without clustering.
Figure 8.2. (a) IGRA data variogram in four directions. The vertical axis shows the averaged covariance between points at a particular distance. Different lines show the variograms in different directions. (b) January IGRA data variogram. The vertical axis shows the averaged covariance between points at a particular distance. (c) Fitted model of the variogram curve bounded by distance 30. (d) The horizontal axis represents time in hours. The vertical axis shows the $\chi^2$ IGRA Kriging error for Kriging with clustering and Kriging without clustering. The $\chi^2$ statistics of clustering-based Kriging is lower than Kriging without clustering across all time stamps.
8.5. Summary and Future Work

In this chapter, we propose a filtering-based clustering algorithm that improves Kriging interpolation accuracy. Theoretical proofs have shown that the final clusters have consistency and convergence characteristics. We have proposed two implementation techniques, caching and heuristics, that help to improve the performance of the proposed algorithm. Furthermore, experimental results have shown that the filtering-based clustering algorithm has significantly improved Kriging interpolation accuracy compared to the traditional Kriging approach and K-means clustering-based Kriging.

Future work consists of at least two directions. The first direction is to derive theorems that can bound the rate of convergence of algorithm 1 given ground truth about data. The second direction is to speed up the performance of the filter algorithm by using parallel programming.
CHAPTER 9

A Hybrid Training Algorithm for Recurrent Neural Network Using Particle Swarm Optimization-based Preprocessing and Temporal Error Aggregation

Elman recurrent neural network [152] has been widely used as an auto-regressive (AR) model in many applications. For example, Hajdarevic uses a recurrent neural network to detect anomalies in thermal plant [153]. Xiao uses a recurrent neural network for impedance identification [154]. A recurrent neural network transits its internal state every time it processes input. This property makes it suitable for modeling complex patterns in time series.

Training recurrent neural network has two important performance considerations, namely final error and convergence speed. Increasing convergence speed allows the building of models on larger datasets with large numbers of parameters. Reducing errors improves model prediction accuracy, thereby making the recurrent neural network more actionable. Backpropagation is a dynamic programming design of first order gradient descent for training neural network [155]. Most recurrent neural networks are non-convex functions, so the final solution of gradient descent can be trapped at local minima or saddle points [156]. The current solution to obviate local minima is to train multiple networks with randomized initial weights and select the one with the lowest errors, but the
execution time can become prohibitively expensive. Thus, techniques that can improve both the convergence speed and accuracy are needed.

To achieve the goal of faster convergence in backpropagation, we propose an enhanced backpropagation algorithm that aggregates averaged errors from future timestamps. A dual equation system is used to delay error averaging in order to solve a numerical overflow problem without losing error size. To reduce the final error, we present a new training method that is based on the concept of particle swarm optimization (PSO) \cite{157} for finding the best-fit initial parameters. Such initial parameters enable the proposed backpropagation algorithm to produce results with much lower errors. The proposed hybrid algorithm consists of two phases: exploration and exploitation. Firstly, PSO is used to explore the search space of parameters based on the activation score. During exploration, an activation list is maintained to store the candidate parameters that produces lowest errors and highest activation scores. Activation score is computed using the principles of network weights initialization method as proposed in \cite{158}. A network with high activation score implies low convergence error after training with backpropagation. At exploitation stage, all parameter settings for the network in the activation list are trained with enhanced backpropagation algorithm. The network that yields the lowest final error is returned as the final output.

We evaluate our proposed methods using four real-world time series datasets: the QT ECG dataset \cite{159} containing ECG curves, MGH/MF dataset \cite{160} recording three-dimensional heart pressures, and two NASA space shuttle datasets \cite{161} collecting activities of space shuttle’s Marotta Valve. Enhanced backpropagation, applied to both regularized and adaptive momentum backpropagation, increases convergence speed by
10% to 20% and reduces testing MSE at convergence by 5% to 30%. Using PSO and activation list in exploration phase, our hybrid training algorithm reduces testing MSEs by more than 30% at convergence compared with traditional back propagation.

9.1. Background and Related Works

Elman neural network, referred as recurrent neural network, uses context layer to adjust its state at different timestamps. Figure 9.1 illustrates Elman neural network. The white arrows denote full connections between layers. A recurrent neural network is equivalent to a Turing machine [162], so it is capable of encoding complex patterns. There are many variants of recurrent neural networks for addressing specific problems, such as echo state network [163], Jordan network [164], and LSTMs [165].

\begin{align}
  h_t &= f(Aa_t + Cc_{t-1}) \\
  c_t &= h_t \\
  o_t &= g(Bh_t)
\end{align}

We formulate the notations used in [166] into compact matrix-vector representations. Vectors $a_t$, $h_t$, $c_t$, and $o_t$ are outputs of input layer, hidden layer, context layer and output layer respectively at time stamp $t$. We denote their size by $|a|$, $|h|$, $|c|$, and $|o|$, where $|c| = |h|$. Let $A$ be a $|h| \times |a|$ matrix such that $A[j,i]$ represents weight from input node $i$ to hidden node $j$. Let $B$ be a $|o| \times |h|$ matrix such that $B[j,i]$ represents weight from hidden node $i$ to output node $j$. Let $C$ be a $|h| \times |h|$ matrix such that $C[j,i]$ represents the weight from context node $i$ to hidden node $j$. Let $f : \mathbb{R}^{|h|} \rightarrow \mathbb{R}^{|h|}$ be the activation function at hidden layer and $g : \mathbb{R}^{|o|} \rightarrow \mathbb{R}^{|o|}$ be the activation function at output layer. $c_t$
is defined to be a zero vector when $t < 0$. For $t \geq 0$, $h_t$ is defined by Equation 9.1. $c_t$ is defined by Equation 9.2. $o_t$ is defined by Equation 9.3.

9.1.1. Back Propagation

Elman neural network can be trained with back propagation algorithm iteratively in four phases. Firstly, the network is unfolded across all time stamps. Figure 9.2 illustrates an Elman neural network unfolded across two timestamps. The second phase is computing
errors. Let $\Delta_t$ be a vector of size $|o|$ that represents the output layer errors at time $t$. Let $\delta_t$ be a vector of size $|h|$ that represents the hidden layer errors at time $t$. Let $e_t$ be the error vector defined by the difference between $o_t$ and actual output of the network at
time \( t \). Let \( f' \) and \( g' \) be the first order derivatives of activation functions \( f \) and \( g \). \( \Delta_t \) and \( \delta_t \) are computed based on Equations 9.4 and 9.5. In the third phase, the gradients for all parameters represented by matrices \( A \), \( B \), and \( C \) are computed using Equations 9.6, 9.7, and 9.8. In the final phase, gradients are averaged across all timestamps for gradient descent parameter updates.

\[
\Delta_t = g'(Bh_t) \odot e_t
\]  
\[
\delta_t = (B^T \Delta_t) \odot (f'(Aa_t + Cc_{t-1}))
\]  
\[
\frac{de_t}{dB[j,i]} = h_t[i] \Delta_t[j]
\]  
\[
\frac{de_t}{dA[j,i]} = a_t[i] \delta_t[j]
\]  
\[
\frac{de_t}{dC[j,i]} = h_{t-1}[i] \delta_t[j]
\]

The calculation of first order gradient descent of neural network, described above, brings up two challenges. The first challenge is to avoid the local optimal solutions caused by non-convexity. The gradient descent process fails to reduce error if a local optimum is reached. A possible solution is to randomly initialize multiple weights parameters and to apply gradient descent to all of them. However, this method has scaling problem because the number of random initialization is proportional to parameter size. Adding a momentum factor during parameter updates is another solution. For example, the ADAM algorithm \[136\] is designed for varying learning rate at current iteration based on momentums. Recent research has also shown that saddle points can slow down gradient
descent convergence \[156\]. \(l_2\) regularization, which adds penalty for large weights, can relieve the risk of saddle points.

9.1.2. Particle Swarm Optimization

Particle swarm optimization (PSO) \[157\] is generic optimization algorithm for arbitrary objective functions. A particle represents one parameter assignment. In case of recurrent neural network, the parameter assignment includes matrices \(A\), \(B\), and \(C\). PSO allows particles to adjust their locations based on both global and local best locations. The rule of thumb for setting the number of particles is \(p = 3N + 1\), where \(N\) is the number of parameters \[167\]. All particles share the same global best location \(g_{\text{best}}\), the location with the smallest MSE. Every particle also has a local best vector \(l_{\text{best}}\), the location with smallest MSE the particle traveled. The velocity vector, determined by the displacements from current location to the locations of both local best particle and global best particle stochastically, is used to update every particle.

The idea of combining particle swarm optimization and back propagation has been discussed in \[168\] and \[154\]. However, they ignore exploration for a wide range of intermediate results. In \[168\], there is no guidance for selecting networks from PSO phase other than MSE, so the output of PSO phase may already be local minimum solution. In \[154\], the authors use gradient as a metric to select outputs from PSO phase. However, large gradient does not necessarily imply lower convergence error after gradient descent. Our proposed hybrid algorithm is different from \[154\]. Instead of selecting networks from
final states of particles, we apply the activation score and the activation list that can thoroughly explore parameter space in the PSO phase. Moreover, we fundamentally improve back propagation algorithm to achieve faster convergence rates with lower errors.

9.2. Design

Algorithm 17: Hybrid Algorithm

```
Data: Time series data
Result: Trained Elman network weights

1. $E \leftarrow$ Synchronous PSO
2. $\forall e \in E$ Apply Enhanced Back Propagation to $e$
3. return $e \in E$ with smallest $MSE(data)$
```

The proposed hybrid algorithm consists of two main phases: exploration and exploitation. Exploration phase uses synchronous PSO to update a global activation list and exploitation uses an enhanced back propagation to train output from the exploration phase. Algorithm 17 describes the proposed hybrid algorithm. Line 1 is the exploration phase and line 2 is the exploitation phase. The parameter assignment that yields the smallest MSE is returned at line 3.

9.2.1. Exploration Phase

**Definition 5.** The activation score of a recurrent neural network is defined by Equation 9.9:

\[
1^T_{|h|} I_1(A)1_{|a|} + 1^T_{|h|} I_2(B)1_{|h|} + 1^T_{|h|} I_3(C)1_{|h|}
\]

(9.9)

$1_c$ is a vector if size $c$ filled with 1 for $c \in \mathbb{Z}^+$. $I_1 : \mathbb{R}^{|h| \times |a|} \to \{0, 1\}^{|h| \times |a|}$, $I_2 : \mathbb{R}^{|a| \times |h|} \to \{0, 1\}^{|a| \times |h|}$, and $I_3 : \mathbb{R}^{|h| \times |h|} \to \{0, 1\}^{|h| \times |h|}$ are matrix indicator functions.
Indicator functions are transformations such that function \( w_i : \mathbb{R} \to \{0, 1\} \), where \( i \in \{1, 2, 3\} \), is applied to every entry of input matrix independently and identically. We use Equation 9.10 to define \( w_i \).

\[
(9.10) \quad w_i(x) = \begin{cases} 
1 & \text{if } |x| < \hat{x}_i \\
0 & \text{otherwise}
\end{cases}
\]

(9.11)

\[
\hat{x}_i = \frac{8.72 \sqrt{\frac{3}{n}}}{\sum_{j=1}^{N} (\max(x_{i,j}) - \min(x_{i,j}))^2}
\]

Let \( n \) be the number of inputs to a single neuron. Let \( \max(x_{i,j}) \) and \( \min(x_{i,j}) \) be the upper bound and the lower bound to the input to neuron \( j \) given input data. \( \hat{x}_i \) is determined as Equation 9.11 using ideas from [158]. If a neural network has a high activation score, the parameter assignments are likely to converge to solutions with low error after gradient descent.

Algorithm 18 is a modified version of PSO algorithm that finds the global best particle in each epoch. Because particles are pulled towards the same global best particle, we call this modification as PSO synchronous update. Algorithm 19 for updating activation list \( AL \) is called at line 13 of Algorithm 18. \( AL \) is a priority queue that stores weight assignments with high activation scores and low MSEs. In Algorithm 19 the function \( AS \) computes the activation score of a particle using Equation 9.9. \( q \) particles with smallest MSEs are inserted into \( AL \). If \( AL \) is full, it pops the particle with the largest error if the particle has activation score less than the particle to be inserted. However, a particle with large MSE and large activation score can block a full list. Thus, lines 11-14 of
Algorithm 18: Synchronous PSO

<table>
<thead>
<tr>
<th>Data: Number of particles $p$, Maximum epochs $n_1$, constant $c_1, c_2$, Activation list size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result: A list of weight assignments $E$</td>
</tr>
</tbody>
</table>

1. $L \leftarrow$ Array of Random Particles($p$)
2. $AL \leftarrow$ New Activation List
3. for $i \in (1, \ldots, n_1)$ do
   4. for $j \in (1, \ldots, p)$ do
      5. $\phi_1, \phi_2 \leftarrow$ random_numbers
      6. $L[j].velocity \leftarrow L[j].velocity + \phi_1 c_1 (L[j].l_{\text{best}} - L[j].loc) + \phi_2 c_2 (g_{\text{best}} - L[j].loc)$
      7. $L[j].loc \leftarrow L[j].loc + L[j].velocity$
      8. if $\text{MSE}(L[j].loc) < \text{MSE}(L[j].l_{\text{best}})$ then
          9. $L[i].l_{\text{best}} \leftarrow L[i].loc$
      end
   end
11. $g_{\text{best}} \leftarrow \min_{q \in L}(\text{MSE}(q))$
12. Call Update Activation List(AL)
13. end
15. $E \leftarrow AL \cup \{g_{\text{best}}\}$
16. Return $E$

Algorithm 19 is a mutation stage that allows insertion of a particle with lower activation score and smaller MSE.

9.2.2. Exploitation Phase

When back propagation algorithm is applied to an unfolded recurrent neural network, hidden layer errors $\delta_t$ are computed using $\Delta_t$ at the same time stamp. It is possible to aggregate error from all future timestamps via links between context layer and hidden layer. This part of error is $C^T \delta_{t+1}$ according to dynamic programming. Let $t_{\text{max}}$ be the total number of timestamps. Let $\gamma_t$ be a vector of size $|h|$ representing the accumulated error for hidden layer. $\gamma_{t_{\text{max}}}$ is set to be equal to $\delta_{t_{\text{max}}}$. Equation 9.13 is proposed to
compute hidden layer error instead of using Equation 9.5 when $t < t_{\text{max}}$. The error back propagated from next layer, namely $\delta_{t+1}$, is averaged by the number of hidden nodes using Equation 9.12. The following theorem shows the new objective errors to be minimized.

(9.12) \[ \gamma_t = (B^T \Delta_t + \frac{1}{|h|} C^T \gamma_{t+1}) \odot f'(Aa_t + Cc_{t-1}) \]

(9.13) \[ \delta_t = (B^T \Delta_t + C^T \gamma_{t+1}) \odot f'(Aa_t + Cc_{t-1}) \]

**Theorem 7.** The objective errors of hidden layer to be minimized at timestamp $t$ in unfolded Elman neural network defined by $\delta_t$ are $\Delta_t + \sum_{i=t+1}^{t_{\text{max}}} \frac{\Delta_i}{|h|^{i-t-1}}$.
Proof. We proceed by induction from $t = t_{\text{max}}$ to $t = 0$. Objective errors can be back propagated through weight matrices by chain rule.

Base case: When $t = t_{\text{max}}$, proof is trivial by Equation 9.5.

Inductive step: Assume $\forall t \geq \tau$, the objective errors of hidden layer to be minimized at timestamp $t$ defined by $\delta_t = \Delta_t + \sum_{i=t+1}^{t_{\text{max}}} \frac{\Delta_i}{|\mathcal{h}|_{i-t-1}}$. We can decompose $\delta_{\tau-1} = x_{\tau-1} + y_{\tau-1}$, where $x_{\tau-1} = B^T \Delta_{\tau-1} \circ f'(Aa_{\tau-1} + Cc_{\tau-1})$ and $y_{\tau-1} = C^T \gamma_{\tau} \circ f'(Aa_{\tau-1} + Cc_{\tau-1})$. It follows that $x_{\tau-1}$ is exactly the error back propagated from $\Delta_{\tau-1}$. It suffices to show that the objective error of $y_{\tau-1}$ is $\sum_{i=\tau}^{t_{\text{max}}} \frac{\Delta_i}{|\mathcal{h}|_{i-\tau}}$. We can decompose $\gamma_{\tau} = x'_{\tau} + y'_{\tau}$ into two parts: $x'_{\tau} = B^T \Delta_t \circ f'(Aa_{\tau} + Cc_{\tau})$ and $y'_{\tau} = \frac{1}{|\mathcal{h}|} C^T \gamma_{\tau+1} \circ f'(Aa_{\tau} + Cc_{\tau})$. Similarly for $\delta_{\tau} = x_{\tau} + y_{\tau}$, where $x_{\tau} = B^T \Delta_{\tau} \circ f'(Aa_{\tau} + Cc_{\tau})$ and $y_{\tau} = C^T \gamma_{\tau+1} \circ f'(Aa_{\tau} + Cc_{\tau})$. $x_{\tau}$ and $x'_{\tau}$ are identical. They represent the part of objective errors propagated from $\Delta_{\tau}$. We also observe that $y'_{\tau} = \frac{y_{\tau}}{|\mathcal{h}|}$.

By inductive assumption, objective output errors of $\delta_{\tau}$ are $\Delta_{\tau} + \sum_{i=\tau+1}^{t_{\text{max}}} \frac{\Delta_i}{|\mathcal{h}|_{i-\tau-1}}$. Hence $y_{\tau}$ has objective error $\sum_{i=\tau+1}^{t_{\text{max}}} \frac{\Delta_i}{|\mathcal{h}|_{i-\tau-1}}$, which implies $y'_{\tau}$ has objective errors $\frac{1}{|\mathcal{h}|} \sum_{i=\tau+1}^{t_{\text{max}}} \frac{\Delta_i}{|\mathcal{h}|_{i-\tau-1}}$. Thus, $\gamma_{\tau} = \Delta_{\tau} + \frac{1}{|\mathcal{h}|} \sum_{i=\tau+1}^{t_{\text{max}}} \frac{\Delta_i}{|\mathcal{h}|_{i-\tau-1}} = \sum_{i=\tau}^{t_{\text{max}}} \frac{\Delta_i}{|\mathcal{h}|_{i-\tau}}$. Since the objective error of $y_{\tau-1}$ is identical of $\gamma_{\tau}$ by chain rule, it follows that the inductive assumption holds for $t = \tau - 1$. \qed

A direct implication of this theorem is that the objective error function to be minimized at any time stamp contains all errors in its future. Moreover, the weights of errors decay in terms of time. Unlike traditional back propagation error definition, the output errors in later timestamps are repeatedly used at the gradient computation before gradient averaging phase in our proposed method. Experimental results have shown the the proposed error computation can improve both convergence speed and accuracy for time series data.
9.2.3. Complexity Analysis

The complexity for computing all outputs of an recurrent neural network is $O(t_{\text{max}}(|a||h| + |o||h| + |h|^2))$, so Algorithm 18 has complexity $O(n_1pt_{\text{max}}(|a||h| + |o||h| + |h|^2))$ excluding line 13 given $p$ particles and $n_1$ epochs as input. Algorithm 19 can reuse the MSEs computed in the same epoch. The rest part does not depend on $t_{\text{max}}$. Thus, the overall complexity for exploration phase is $O(n_1pt_{\text{max}}(|a||h| + |o||h| + |h|^2))$.

The complexity for exploitation can be divided into two parts: error back propagation and weights update. For error back propagation, the unfolded network has $t_{\text{max}}$ number of layers. Computation for errors has complexity $O(t_{\text{max}}|h|(|h| + |a| + |o|))$. Gradient computation based on errors has the same complexity. Weights update operates on folded recurrent neural network, so this process has complexity of $O(|a||h| + |o||h| + |h|^2)$. Suppose $AL$ has size $m$, the entire exploitation phase has complexity $O(n_2mt_{\text{max}}(|h|^2 + |a||h| + |o||h|))$, given $n_2$ epochs.

\[
\frac{n_1t_0 + n_2mt_1}{mk^2} = \frac{n_1p}{mk^\beta} + \frac{n_2}{k^\alpha}
\]

Let $t_0$ and $t_1$ be the execution time of one epoch in synchronous PSO and enhanced back propagation respectively. Equation 9.14 shows the ratio of hybrid algorithm to traditional back propagation for $m$ number of recurrent neural networks with $k$ number of epochs. $\alpha$ and $\beta$ are constants. We can compare the end-to-end execution time of proposed hybrid algorithm and traditional back propagation using Equation 9.14.
9.3. Experimental Results

We use four datasets to evaluate our algorithm: the QT ECG dataset [159], two NASA space shuttle datasets [161], and MGH/MF dataset [160] recording three-dimensional heart pressures. Table 9.1 summarizes all datasets.

Table 9.1. Chapter 9 datasets characteristics.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Timestamps</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>QT ECG</td>
<td>1000</td>
<td>1</td>
</tr>
<tr>
<td>Space Shuttle Puppet Withdraw</td>
<td>4000</td>
<td>1</td>
</tr>
<tr>
<td>Space Shuttle Puppet Pulled out</td>
<td>2000</td>
<td>1</td>
</tr>
<tr>
<td>MGH/MF</td>
<td>3600</td>
<td>3</td>
</tr>
</tbody>
</table>

QT ECG dataset measures the electrical voltage of human heart beats at different timestamps [159]. NASA space shuttle datasets illustrate the sensor records of energizing phases of space shuttle Marotta Valves. MGH/MF dataset [160] contains multiple electrical recordings of patients’ conditions. For MGH/MF dataset, we focus on three attributes: arterial pressure, pulmonary arterial pressure, central venous pressure in particular because it has been argued in [169] that the heart pressures are correlated, which enables the use of AR model.

9.3.1. Enhanced Recurrent Back Propagation

Current research works focus on gradient descent phase of back propagation algorithm in order to improve convergence speed and accuracy. Our proposed enhanced back propagation changes the computation of objective error in the error back propagation stage, so it can be combined with existing gradient descent methods. In this section, we demonstrate the convergence speed and final MSE improvement for two state-of-art methods:
Equation 9.15 defines discrete average MSE convergence speedup. It presents that the average percentage of error reduction given the same execution time. Given a recurrent neural network, \( \text{old}[i] \) refers to the MSE of the traditional back propagation and \( \text{new}[i] \) refers to the MSE of enhanced back propagation for running \( i \) number of epochs. Because the proposed error aggregation bears the overhead of extra matrix-vector multiplications per epoch, the comparison of \( \text{old} \) and \( \text{new} \) should have a phase shift in order to compute average MSE convergence speedup given the same execution time. The phase shift percentage is represented by \( \alpha \) in Equation 9.14 and 9.15.

We initialize 32 recurrent neural networks with randomized weights. For each of the networks, we apply the same gradient descent algorithm using errors computed by Equations 9.5 and 9.13 separately to the same training data with the same number of epochs. Learning rate ranging from 0.5 to 1.0 and \( \lambda = 0.01 \) are used for rBP. We evaluate convergence speedup and error reduction at convergence using testing data with two-fold cross validation and average results of 32 networks.

Figures 9.3a, 9.3b, 9.3c, 9.4a, 9.4b, and 9.4c illustrate testing error convergence in terms of execution time. Table 9.2 and Table 9.3 summarizes the average convergence speedup and MSE percentage reduction at convergence for all datasets. Although Equation 9.13 results longer execution time per epoch, the gradient descent algorithm converges faster given the same execution time. Moreover, the new error definition gives
lower convergence errors. In addition, as we increase the size of dataset, both average MSE convergence speedup and MSE percentage reduction at convergence are improved.

Table 9.2. The improvements from new error definition for rBP gradient descent.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>MSE Convergence Speedup</th>
<th>MSE reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSPW</td>
<td>18.1%</td>
<td>9.38%</td>
</tr>
<tr>
<td>SSPO</td>
<td>16.8%</td>
<td>6.83%</td>
</tr>
<tr>
<td>QT-ECG</td>
<td>10.2%</td>
<td>13.3%</td>
</tr>
<tr>
<td>MGH/MF</td>
<td>17.2%</td>
<td>28.3%</td>
</tr>
</tbody>
</table>

Table 9.3. The improvements from new error definition for ADAM gradient descent.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>MSE convergence Speedup</th>
<th>MSE reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSPW</td>
<td>24.0%</td>
<td>17.4%</td>
</tr>
<tr>
<td>SSPO</td>
<td>23.6%</td>
<td>11.8%</td>
</tr>
<tr>
<td>QT-ECG</td>
<td>9.74%</td>
<td>25.7%</td>
</tr>
<tr>
<td>MGH/MF</td>
<td>15.2%</td>
<td>27.5%</td>
</tr>
</tbody>
</table>

9.3.2. Hybrid Algorithm

Table 9.4. MSE percentage reductions of hybrid algorithm (including all phases) for rBP and ADAM gradient descent.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>rBP MSE Reduction</th>
<th>Reduction</th>
<th>Time Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSPW</td>
<td>29.0%</td>
<td>38.6%</td>
<td>52%</td>
</tr>
<tr>
<td>SSPO</td>
<td>28.9%</td>
<td>35.1%</td>
<td>53%</td>
</tr>
<tr>
<td>QT-ECG</td>
<td>64.5%</td>
<td>66.4%</td>
<td>41%</td>
</tr>
<tr>
<td>MGH/MF</td>
<td>45.6%</td>
<td>44.8%</td>
<td>50%</td>
</tr>
</tbody>
</table>

With the help of exploration phase, the hybrid algorithm can further reduce convergence error. \( n_1 = 10 \) is set to be the number of epochs for exploration phase. \( m = 32 \)
recurrent neural networks with randomized weights are trained using both ADAM and rBP using traditional back propagation as benchmarks. To make a fair comparison, the activation list size is set to be 31, so $m = 32$ recurrent neural networks are produced by exploration phase and trained with enhanced back propagation algorithm.

Table 9.4 summarizes the results of two-fold cross validation for improvement of MSE using the hybrid algorithm. MSE reduction column shows the percentage MSE reductions achieved by the hybrid method when different gradient descent methods are used. The average MSE reductions are higher compared with the results shown in Table 9.2 for all datasets. However, the execution time of the hybrid algorithm is approximately 40% to 50% longer. According to Equation 9.14 if $n_2$ and $m$ become large, the execution time ratio of old method to new method approach to $\alpha$ given fixed $n_1$. Moreover, decreasing either of $n_1$ and $n_2$ reduces execution time, but resulting larger error. Thus, there is a trade off between execution time and accuracy, users should tune the values of $n_1$ and $n_2$ depending on the timing and accuracy constraints.
Figure 9.3. MSE of testing data against epoch under rBP. (a) Space shuttle puppet withdraw (b) Space shuttle puppet pulled out. (c) QT ECG database record 0606. (d) MGH/MF database record 0003.
Figure 9.4. MSE of testing data against epoch under ADAM. (a) Space shuttle puppet withdraw (b) Space shuttle puppet pulled out. (c) QT ECG database record 0606. (d) MGH/MF database record 0003.
CHAPTER 10

Conclusion and Future Work

This thesis archived the works I did during my Ph.D. study for scalable algorithms of MPI intergroup communication, MPI-IO, and miscellaneous machine learning applications. The proposed algorithms can benefit large-scale workflow performance from the angles of communication, data storage, resource scheduling, and system diagnosis.

In the intergroup communication work, optimal algorithms for fully connected and ring topologies are proposed. Modern supercomputers have topologies that are in between these two extreme connectivities. For instance, ALCF Theta and NERSC Cori have a dragon-fly topology. OLCF Summit has a fat-tree topology. Although algorithms designed for the fully-connected topology are effective for these topologies according to the experiments, we can gain further performance improvement if we customize the communication algorithms for special topologies. MPI implementations can dynamically determine the optimal algorithm to use depending on the topology of compute nodes in a communicator. Furthermore, in Chapter 3 we developed notations for concurrent sequences on ring topology. The next step can start from extending the intergroup Allgather algorithm for ring topology to adapt to N-dimensional torus. The extended work can be a general mathematical theory that optimally solves the problem for the intergroup all-to-all broadcast pattern for N-dimensional torus.

The MPI-IO works mentioned in this thesis are focusing on the improvement of the two-phase I/O communication phase. These improvements are client-side improvements,
involving layers of aggregation and rearrangement of communication orders for client MPI processes to reduce communication contention. For future works, adjusting two-phase I/O designs to adapt to new communication and I/O hardware on new supercomputers can benefit the application performance. For example, with the knowledge learned from the collective communication work, it is possible to further improve the communication phase of the two-phase I/O by designing layers of I/O requests aggregation taking the network switch topology into account. Moreover, my ongoing work is focusing on improving the delegated I/O server performance, named Benvolio \[170\]. Instead of using client processes as I/O aggregators, a few server processes, called providers, are handling I/O requests for different applications in this model. Objected-centric storage systems such as proactive data container \[171\] also uses delegated I/O servers to manage metadata and data for I/O requests from clients. Existing techniques, such as caching, can be implemented at the providers to coalesce I/O requests for better performance. There is a lot of room for further improving the performance of delegated I/O service using the knowledge I have discussed in this thesis. For example, we can take advantage of the two-layered aggregation method (TAM) at the client-side and caching at the provider side to provide faster I/O services. Another possible extension is to apply contention controls similar to throttling to I/O servers. This approach can avoid communication contention from an overwhelming number of client RPCs. My future work will continue on these topics.
References


[10] Qiao Kang, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao. Optimal algorithms for half-duplex inter-group all-to-all broadcast on fully connected and ring topologies.


with the International Conference for High Performance Computing, Networking, Storage and Analysis, November 2018.


[97] Sheng Di, Hanqi Guo, Eric Pershey, Marc Snir, and Franck Cappello. Characterizing and understanding hpc job failures over the 2k-day life of ibm bluegene/q system.


international conference on Modeling and Mining Ubiquitous Social Media, pages 118–137, 2011.


