AN MPI-BASED 2D DATA REDISTRIBUTION

LIBRARY FOR DENSE ARRAYS

By

Evelyn Namugwanya

Anthony Skjellum
Professor of Computer Science
(Chair)

Amanda Bienz
Professor of Computer Science
(Committee Member)

Michael Ward
Professor of Computer Science
(Committee Member)

AN MPI-BASED 2D DATA REDISTRIBUTION

LIBRARY FOR DENSE ARRAYS

By

Evelyn Namugwanya

A Thesis Submitted to the Faculty of the
University of Tennessee at Chattanooga
in Partial Fulfillment of the Requirements of the
Master of Science in Computer Science

The University of Tennessee at Chattanooga
Chattanooga, Tennessee

August 2021

ABSTRACT

In HPC, data reorganizations of dense arrays are used in parallel applications for performance and data-locality compatibility with parallel operation sequences. Data reorganizations change the arrangement of data across distributed memories. This is achieved through message passing plus algorithmic re-indexing.

Our primary goal is to generate a library capable of diverse data reorganizations. We aim for a high-level Application Programming Interface working compatibly with the Message Passing Interface to accomplish data redistributions in data-parallel applications and libraries, such as in conjunction with the Polymath matrix-multiplication poly-algorithm library.

We compared performance trends of the Polymath parallel matrix-multiplication library based on different grid shapes, problem sizes, and numbers of processes when combined with our reorganization algorithms, producing a choice of "compute as is" or "redistribute then compute." We found it more efficient to redistribute data in 37 cases, but, importantly, not in all cases. We also studied and demonstrated data transpose algorithms.

DEDICATION

I dedicate this work to my family and friends. A special feeling of honor to my loving parents, Emmanuel and Concepta Maviiri whose love and words of comfort ring in my ears. My sisters Angel, Jackie, Leticia, Martha and my brothers Vegas, and Ben have never left my side and are very special. I also dedicate this thesis to my friends at church who have prayed for me throughout the process. May God Bless you, especially Terry Ruff, Pastor. Mathew, Pastor. Kwaku and Devone. I dedicate this work and give special thanks to Dr. Maxwell Omwenga for all his kindness and the technology skills I have learned from him. I dedicate this work to my best friend Grace Nansamba for being my source of inspiration and having my back throughout the entire program. You have been my best cheerleader!

ACKNOWLEDGMENTS

First and foremost, I would like to praise and thank God, the Almighty, who has granted blessings, knowledge, opportunity, and the gift of life to me and everyone reading this thesis.

My sincere appreciation goes to my advisor, Dr. Anthony Skjellum, for his exceptional support, enthusiasm, knowledge, and all opportunities he has given me, which have been vital and fundamental towards my research and for challenging me with various tasks to learn and grow. Dr. Skjellum, thank you for being patient with me, and you bring out the best in everyone! I would also like to extend my appreciation to Derek Schafer, I would not have made it without you! Thank you for challenging me and teaching me all life coding skills!

I would also like to extend my appreciation to my entire committee Dr. Anthony Skjellum, Dr. Michael Ward and Dr. Amanda Bienz; thank you for reading my thesis and advising me through my tasks; I am forever grateful! I thank the University of Tennessee at Chattanooga, Department of Computer Science, faculty staff for giving me the chance to have a meaningful and outstanding education and experience. I want to say thanks to the SimCenter staff for providing me apparatus for my research, Am forever grateful!

I also thank Grace Nansamba for all the support towards my research team. I acknowledge Dr. Maxwell Omwenga, Dr. Amani Altarawneh, Dr. Joseph Kizza and Dr. Immaculate Kizza for affirming me whenever I needed support! Lastly, I would like to thank Kim Sapp, thank you for encouraging me and for all those lovely dresses.

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API — Application Programming Interface

BLAS — Basic Linear Algebra Subprograms

DRI — Data Reorganization Interface

FFT — Fast Fourier Transform

HPC — High Performance Computing

MPI — Message Passing Interface

CHAPTER 1

INTRODUCTION

Data redistribution is one of the most important operations performed on application programs to improve performance or support operation compatibility in high performance computing (HPC). No efficient, standalone or easily accessible data redistribution library has been found in our study of open-source software that redistributes two-dimensional (2D) matrices from one 2D logical process topology to another at constant or varying numbers of processes. Such operations are evidently interspersed in applications or other scientific libraries that require them, but without a systematic application program interface (API) that is application or library-independent.

While the cases of 1-to-all (scatter) and all-to-1 (gather) are relatively easy to implement with collective message-passing operations and derived data types in the Message Passing Interface (MPI) [14], the cases where the shapes change but are both concurrent in both matrix rows and columns is our primary focus; we address two-dimensional layouts across process memories here[1].

Ultimately, our goal is to support general data organizations beyond linear layouts of data, but we considered linear mappings of matrices to two-dimensional logical process grids (working via MPI communicators [14]) here.

This thesis focuses on building an API and prototype implementation that uses the Message Passing Interface (MPI) to provide a *Data Redistribution Library*.

We want to answer the research question about when it is useful to redistribute data involving two-dimensional data layouts as a proof of utility of this library. Thus, a key part of the experimental plan is to integrate loosely with linear algebra libraries like the Polymath library [29]. Ultimately, we will support what-if experiments about the performance benefits of redistributing

[1] Full or partial replication in a third dimension can be useful, but is not part of this thesis work.

versus working on data in a given 2D layout across a specific shape of logical process topology. In this work, we use compatible cases to show when Polymath would prefer to redistribute with our library, then compute, versus computing in the layout it was originally given to perform matrix multiplication. In future work, a library-to-library integration will be achieved so that Polymath gains redistribution capability at its current level of data-distribution flexibility.

Thus, our primary goal is to observe the performance trends of the Polymath library based on different grid shapes, problem sizes, numbers of processes and decide whether it is valid to redistribute data or not. Our secondary goal is to add data transposition functionality to our library, but the performance of that library is not addressed in our results.

1.1 Motivation

Data reorganization is considered vital and fundamental in several scientific linear algebra computations such as matrix transpose, swap, multiplication, shuffle. Data reorganization libraries generally provide optimized implementations of other high performance computing libraries (e.g., Polymath) to improve performance.

Poly-algorithms refers to a group of related algorithms grouped to perform associated operations. However, none of them is a faster algorithm, yet each can achieve the best performance based on the given parameters. The poly-algorithm library was developed by Skjellum and colleagues starting in 1991 [26]. This library contains algorithms that apply to the general case of rectangular matrix multiplication on rectangular process grids and are classified according to the communication primitives used. In the Polymath library, DGEMM kernel is used to perform a double-precision matrix multiplication operation. Data redistribution/movement depends on the basic operations such as rank one update, matrix-vector operations and all these influence the choice of matrix algorithm as presented by Gunnels [16].

This thesis was inspired by DRI (Data Reorganization Interface) developed by the Data Reorganization Forum in 2002 [36]. DRI provides a high-level API to accomplish data redistributions in the data-parallel application (matrix transpose, swap etc.).

One of the significant challenges of data redistribution operations is that it is both costly in terms of added memory and time-consuming. This thesis aims to produce a data reorganization

library that will prove to be efficient (fast and flexible). Our library manipulates different data movements required to complete multiplication in parallel linear algebra libraries. It also reorganizes different process shapes grids of the matrix (i.e., reorganizes a matrix on a 2D $R = P \times Q$ logical process grid into a $R = P' \times Q'$ grid).

In addition, the need to convert the DRI library specification from C language to C++ influenced this thesis. C++ features also motivated the development of the library, the object-oriented nature of C++, quality of portability or platform independence which allows the user to run the same program on different operating systems. The combination of MPI and C++ results can produce quality library codes using C++ classes, templates, inline functions, overloading, and many other features that enable maintainable, passing by reference and efficient code. MPI and C++ were used in this thesis, as you are yet to discover through this work.

1.2 Problem Statement and Objectives

These are the research questions that we want to answer:

1. Is it always, sometimes, or never worthwhile to redistribute a dense matrix on a 2D process grid? If so, in which cases should data be redistributed?

2. When combined with Polymath and redistribution proves useful for a given use case, does the best algorithm to use on a given problem case change also after redistribution?

3. For given problem cases (generated via Polymath), does data redistribution lead to significant performance enhancements for dense parallel matrix-matrix multiplication?

Our Data Redistribution Library will aim to reshape the logical 2D process grid $P \times Q$ to $P' \times Q'$ (with fixed total numbers of processes) and observe performance trends of the Polymath algorithm, thus revealing the impact of data redistribution and decide appropriately on the need to redistribute data.

With that in mind, the objectives of this thesis are as follows:

1. To redistribute dense matrices by reshaping the process grid $P \times Q$ to $P' \times Q'$ on which they are distributed, focusing on linear distributions.

2. To design, prototype, validate, and benchmark a high-level API to accomplish data redistribution in data-parallel applications (dense rectangular arrays/matrices on 2D logical process topologies).

3. We will test and compare the performance of Polymath use cases and the redistributed data cases, to see when we should redistribute or not.

4. We will use C++ to design and prototype our library and exploit modern object-oriented and meta-programming concepts to create a quality, initial library implementation.

1.3 Contributions

The following are the contributions we plan to make in our research:

- Design and implement a data redistribution library. We will design and implement a high level Application Programming Interface that directly works with the Message Passing Interface (MPI) to accomplish data redistributions of 2D arrays of data (matrices) on 2D logical process grids (topologies). Our data redistribution library will include algorithms to enhance data redistribution: Matrix transpose algorithm, this algorithm switches the rows of a matrix with its columns. Rotating a matrix at 90 degrees algorithm flips the matrix by 90 degrees. The resizing P, Q algorithm reshapes the matrix from a P, Q logical grid to P', Q' logical grid with the same number of processes. This thesis will mainly focus on the resizing P, Q, algorithm in terms of performance, but full performance for the transpose case will be future work.

- We analyze whether to redistribute data based on the redistribution costs varying with different shapes compared to the original computation costs in Polymath. This way, we can tell how data redistribution (resizing the dimensions of the process grid) influences the performance of Polymath algorithms. For the purpose of performance comparisons, we choose to consider that one matrix is being redistributed for these comparisons. In some cases, it may also be beneficial to redistribute both the A and B matrices of the form $C = A \times B$. We note that parallel applications determine input matrix shapes and sizes

based on their effort to reduce computation costs or because of algorithmic limits; providing flexibility at parallel-operation interfaces is a value of our library and study.

1.4 Outline

The remainder of the thesis is organized as follows: Chapter 2 includes the background and related work. Chapter 3 presents the methodology and the implementation of the algorithms. In Chapter 4, we discuss the results from experiments that we carried out. We conclude the thesis in Chapter 5 by discussing our findings and outlining future work.

CHAPTER 2

BACKGROUND AND LITERATURE REVIEW

This chapter discusses what has already been attempted in the parallel area of data redistribution of matrices. First, we discuss the motivating specification of the Data Reorganization Interface (DRI) 2002 by the DRI Forum, an informal standards group funded by the Defense Advance Research Projects Agency (DARPA) [36]. The remaining subsections cover complementary algorithms, libraries, and domain-specific issues related to data transpose and reorganization, all of which are relevant to this thesis. In particular, the Polymath library is a parallel algorithm collection to support dense matrix-matrix multiplication, and is used in significant ways to demonstrate and complement the work of this thesis. In the rest of this chapter, we consider relevant prior work by other researchers about data redistribution of matrices on logical process topologies using the message passing of parallel computing.

2.1 Data Reorganization Interface

This thesis was inspired by the DRI-1.0 (Data Reorganization Interface 2002) which was developed by Dr. Skjellum and colleagues [36] in the Data Reorganization Forum. DRI provided a high level API to accomplish data redistributions in data-parallel applications with dense matrices. DRI specifies application programmer interfaces to transpose and redistribute data over logical process topologies. This specification chose C as its descriptive implementation language, rather than a language-neutral specification.

2.2 The fftMPI Library

The fftMPI redistribution library by Plimpton et al. [32] includes data reorganization for multi-dimensional Fast Fourier Transform (FFT) operations. The fftMPI library computes 3D and 2D FFTs in parallel assets of 1D FFTs (via an external library) in each dimension of the FFT

grid interleaved with MPI communication to move data between processors. The fftMPI library has four classes: FFT3D, FFT2D, Remap3D, Remap2D. The FFT classes perform 3D and 2D ffts. The Remap classes perform a data remap, which communicates, reorders, and reorganises the distributed 3D and 2D arrays across processors. However, FFTs were not needed for the data redistribution library. Also, fftMPI library lacked some of the data redistribution functionalities like resizing grid shapes and rotating data at 90 degrees. Therefore, we decided to think of a library with all the missing functionalities from the prior libraries.

Note that our redistribution library is built to have some of the above functionality, which makes it unique and relevant for linear algebra operation especially in dense matrix multiplication.

2.3 Data Transposition

Researchers have explored the concept of data redistribution for a many years. Choi et al. [8] introduced parallel matrix transpose algorithms based on block-cyclic data distribution. They assumed that the matrix is distributed over a $P \times Q$ processor template with block-cyclic data distribution. Where P, Q and the block size could be arbitrary. The communication schemes of the algorithms were determined by the greatest common divisor (GCD) of P and Q. If P and Q were relatively prime, the matrix transpose algorithm involves complete exchange communication on a two-dimensional template. They approached the above mentioned problem with a point-to-point communication scheme. The algorithms use non-blocking (`MPI_Isend`, `MPI_Irecv`), point-to-point communication between processors to overlap messages sent to different MPI processes. The non-blocking calls also avoided unnecessary synchronization. The parallel matrix transpose algorithms were combined with matrix multiplication routines which comprise a general-purpose matrix multiplication package, called PUMMA [9]. This paper is motivates our transpose functions, which are similar.

Azari et al. presented algorithms for transposing a matrix on a mesh-connected array processor (MCAP) [2]. They discussed both synchronous and asynchronous algorithms. In the synchronized approach, algorithms apply a global clock to synchronize the communications between processing elements. They further explained that the number of time units required by synchronous algorithms for transposing an $m \times n$ matrix on an $n \times n$ MCAP is $2(n-1)$.

The synchronous algorithms eliminate simultaneous requests for using channels between process elements. An asynchronous approach was proposed to check the problem of clock skews and delays since it is a self-timed approach. They performed a feasibility of the asynchronous algorithm, and the simulation of the algorithm was demonstrated for different sizes of matrices.

There is evidently a significant additional literature in data transpositions beyond PUMMA. Since transposition was not the primary deliverable of this thesis, we do not provide further literature review here on this topic.

2.4 Poly-algorithms

Poly-algorithms refers to having a group of related algorithms that perform equivalent operations. However, none of them is always known to perform best for arbitrary problem shapes, sizes, and con-currencies. Jin Li et al., Nansamba [21, 29] asserted that Polymath forms a backbone in the building of high performance linear algebra libraries. They affirm that no single algorithm in a library is the best because its performance will deteriorate in different situations and excel in other scenarios. Different algorithms operate differently concerning computation, communication, overheads, and potential overlaps when one changes the dimensions of the grid or the matrices. This leads to some algorithms changing for the better, and some become worse in terms of performance. Polymath incorporates 14 algorithmic variants of parallel, dense matrix-matrix multiplication that generalize classical parallel multiplication algorithms in order to offer the best performance for a given 2D layout of data on logical topologies when the matrices are rectangular, and the process grids are, in general, rectangular.

2.5 Efficient Data Reorganization Research

Data redistribution aims to reshuffle data in order to optimize some objective for an algorithm. Cao et al. in Flexible Data Redistribution [5] presented a flexible and general redistribution algorithm for a task-based runtime system that supports any regular and irregular data distributions. They further provided an implementation in a task-based runtime PaRSEC. The practical evaluation of their implementation showed that it could achieve better performance compared with existing tools supporting some level of data redistribution. They noticed that PaRSEC performance results show great capability compared to ScaLAPACK [22]. Cao et al.

presents a two different approaches for efficient data reorganization [5]. It combines (i) a proposed DRAM-aware reshape accelerator integrated within 3D-stacked DRAM, and (ii) a mathematical framework that is used to represent and optimize the reorganization operations. The algorithm applies to dense matrices. They used blocked data layout and Morton data layouts which involves space filling curves and are based on recursive blocking. Large data sets were divided into blocks recursively until the small leaf blocks reach the desired size. Software Transparent Data Layout Transformation focuses on hardware based data layout transform for address remapping. The key findings in this paper included the observation that common data reorganization operations could be represented as permutations. A mathematical framework was utilized to manipulate the permutations for efficient operation within the stack.

Siegel et al. [35] developed an improved algorithms, MADRE, that combines the advantages investigated by Pinar and Hendrickson from two families of memory-limited redistribution algorithms. The first family had a couple of advantages however, it failed on specific inputs. Also, it was not implemented carefully; it may have led to an explosion in the number of local data copies. The second family eliminated the possibility of failure at the expense of considerable additional overhead. They were both evaluated according to the following specific criteria. They applied the data layout to avoid local movement during scheduled execution. The authors used a sparse format to represent a phase structure. They further developed the Local Copy Efficient (LCE) algorithm $O(m)$ local data copies on each process. It was observed that the LCE algorithm performs better than the other MADRE algorithms in most cases. They used MPI operations for different functionalities i.e mapping processes to one cores.

Sudarsan et al. [37] introduced a framework, "reshape," which enables parallel message-passing applications to be resized during execution. They extended the resizing functionality in reshaping to support redistribution of 2D block-cyclic matrices distributed across a 2D processor topology. The authors derived an algorithm for redistributing two-dimensional block-cyclic arrays from P to Q processors, organized as 2D processor grids. The algorithm ensures a contention-free communication schedule for data redistribution if $P\ r \leq Q\ r$ and $P\ c \leq Q\ c$. It implements circular row and column shifts on the communication schedule to minimize node contention. Block-cyclic

data redistribution was required to achieve computational efficiency. The authors applied four main stages in data redistribution: data identification and index computation, communication schedule generation, message packing and unpacking and data transfer. The algorithm builds an efficient communication schedule using non-all-to-all communication for data redistribution. The authors applied row and column transformations using the circulant matrix formalism to minimize node contentions in the communication schedule. They further discussed the modifications needed to port an existing scientific application to use the dynamic resizing capability of reshaping the algorithm using the given framework's API.

Prylli et al. [33] was the closest related work to the reshape redistribution algorithm because it supports redistribution on checkerboard processor topology. They planned to reshape a more extensible framework so that support for heterogeneous clusters, grid infrastructure, shared memory architectures, and distributed memory architectures could be implemented as individual plug-ins to the framework.

Omiecinski [30] presents research works about physical data reorganization in memory to improve performance. Physical data reorganization is significant because computations are free of data dependencies. However, due to limited data reuse, reorganization operations are bound to incur considerable energy and overheads in conventional systems as discussed by Akin et al. [1].

2.6 Neural Networks Data Reorganization Research

The authors see a growing need for data reorganization in recent neural networks for various applications such as Generative Adversarial Networks (GANs). Kang and Ha [24] proposed a novel technique, called tensor virtualization technique. This technique perform data reorganization efficiently with a minimal hardware addition for adder-tree based CNN accelerators. In the proposed technique, a data reorganization request is specified with a few parameters and data reorganization is performed in the virtual space without overhead in the physical memory. It is urged that the proposed technique reduces the computation workload of DCGAN and DiscoGAN significantly by skipping the ineffectual computation and efficient handling of zero paddings by tensor virtualization. It also reduces the DRAM access volume significantly for the U-Net model with NN upsampling and for SRGAN with sub-pixel convolution. It further extends the application

domain of traditional CNN accelerators drastically by adding a minimal hardware the module that executes some software functions.

2.7 Redistributing Data in Cyclic Blocks

Several past research efforts by Chung et al., Desprez et al., Guo and Pan, Hsu et al., Kalns et al., Kaushik et al., Park et al., Ramaswamy et al., Thakur et al., Thakur et al.,Walker et al., [10, 11, 17, 20, 23, 25, 31, 34, 38, 39, 40] aimed at redistributing cyclically distributed one-dimensional arrays between the same set of processors in a cluster on a 1D processor topology. Walker and Otto [40], and Kaushik et al. [25] proposed a K-step communication schedule based on modulo arithmetic and tensor products to reduce the redistribution overhead cost. The PITFALLS redistribution technique [11] uses line segments to map array elements to a processor. The algorithm can also handle any arbitrary number of source and destination processors. Thakur et al. [39, 38] use GCD and LCM methods for redistributing cyclically distributed one-dimensional arrays on the same processor set. Thakur et al. [38] and Ramaswamy et al. [34] describe algorithms that use a series of one-dimensional redistributions to handle multidimensional arrays. However, the approach can lead to significant redistribution overhead cost due to unwanted communication. Kalns et al. [23] presents a technique that reduces the total amount of data that must be communicated during redistribution. The technique is essential for mapping data to processors by assigning logical processor ranks to the target processors. Hsu et al. [20] extended this work by Kalns et al. [23] and proposed a generalized processor mapping technique for redistributing data from cyclic(kx) to cyclic(x), and vice versa. x denotes the number of data blocks assigned to each processor. However, this method is not flexible as it is applicable only when the number of source and target processors is the same.

Chung et al. [10] proposed an efficient method for index computation using basic-cycle calculation (BCC) technique for redistributing data from cyclic(x) to cyclic(y) on the same processor set was proposed. Hsu et al. [18] extended this work and used a generalized basic-cyclic calculation method to redistribute data from cyclic(x) over P processors to cyclic(y) over Q processors. The generalized basic-cycle calculation uses a bipartite matching approach for data

redistribution. Park et al. [31] developed a redistribution framework that could redistribute one-dimensional arrays from one block-cyclic scheme to another on the same processor set using a generalized circulant matrix formalism. In order to generate a conflict-free schedule, the algorithm applies row and column transformations on the communication schedule matrix.

Prylli and Tourancheau [33] assumed that the data are stored contiguously in a cyclic block fashion on the processors; the problem was to find which data items stored on processor Pi will be sent to processor Pj. These data items have to be packed in one message before being sent to Pj to avoid start-up delays. The authors introduced algorithms that could operate in one dimension. The algorithm scans the matrix indices of the data blocks stored on Pi and those stored on Pj simultaneously. The algorithm keeps two counters, one corresponding to Pi's data location in the global matrix and Pj's one. The counters are incremented progressively by block as in a merge sort. In order to determine the overlap areas, pack the data items corresponding to the overlap areas in one message to be sent to Pj. They argued that redistributing data improves the efficiency of parallel linear algebra routines. However, to ensure improved efficiency from data redistribution, the elapsed time for the redistribution of data has to be efficient. The algorithm complexity analysis shows that the scanning is negligible for arrays of standard size. They also applied optimization like distribution parameters only at run time instead of at compile time. Experimental results affirm that redistribution of data with the discovered algorithms can efficiently perform up to 80/100 gain and assure the average that the computation time stays close to the optimal even with bad initial data distribution choices for both block sizes and grid shape. Tests were run on a Cray T3D and an Intel Paragon. The computation of the data sets was negligible compared to the communication and packing and the global redistribution routine execution time. They plan to improve their algorithm by integrating redistribution routines in the linear algebra kernel themselves. The maybe efficient results of this research encourage the frequent use of these redistribution library routines in explicit parallel programming. The algorithms are implemented within the ScaLAPACK library [22].

Hsu and colleagues presented a processor replacement scheme to efficiently perform block-cyclic data redistribution on a symmetric matrix by minimizing the cost of interprocessor data exchange during runtime [19]. The key idea of the technique was to develop a replacement

function for reordering logical processors in the destination phase. A realigned sequence of destination processors is derived and used to perform data decomposition in the receiving phase in the replacement function. The desired destination distribution can be accomplished without interprocessor communication for some exceptional cases by remapping the ranks of destination processors combined with the matrix transposition scheme. They also affirmed that with local matrix and compressed CRS vector transposition schemes, the interprocessor communication could be eliminated during runtime. Since redistribution is performed at runtime, there is a performance tradeoff between the efficiency of the new data decomposition for a subsequent phase of an algorithm and the cost of redistributing data among processors. Secondly, the paper introduces a generalized symmetric redistribution algorithm to analyze the efficiency of the proposed technique. The algorithm proves that the technique is efficient, saving up to $(p-1)/p$ data transmission cost. For general cases, the symmetric redistribution algorithm saves $1/p$ of communication overheads compared with the traditional method. Experimental results also show a superior performance of the algorithm in most data redistribution instances. The techniques apply to both dense and sparse matrices.

Choi and Dongarra [6] demonstrated core factorization routines that can be parallelized easily to the corresponding ScaLAPACK routines with a small set of low-level modules which include: the sequential BLAS, the Basic Linear Algebra Communication Subprograms [13], and the PBLAS [7]. The PBLAS (Parallel BLAS) are particularly useful for developing and implementing a parallel dense linear algebra library relying on the block cyclic data distribution. They also noted that parallel routines implemented with the PBLAS have good performance. This is because the computation performed by each process within PBLAS routines could be performed using the assembly-coded sequential BLAS [12]. The designing and implementing software libraries had a tradeoff between performance and software design considerations, for example modularity and clarity. The authors illustrated the practicability of combining messages to reduce the communication cost in several places. They also described the chance of replacing the high level routines by calls to the lower level routines, such as the sequential BLAS and the

BLACS. The key finding in this paper was the ScaLAPACK factorization routines which have good performance and scalability on the Intel iPSC/860, the Delta, plus the Paragon systems.

Moreton-Fernandez et al. [27] presented a method based on four combinable operators to efficiently and redistribute partial domains selected by the programmer at run time. The four operators efficiently implement distributed memory algorithms, making the data partition, relocation and data movement transparent to the programmer. The solution presented in this paper provides programming abstractions to manage data redistributions. With the proposed solution, the programmer does not need to deal with data-redistribution implementation issues that are not related to the algorithms but are pivotal in performance. The proposed solution and optimized MPI codes have the same performance as optimized MPI codes with tailored data redistribution solutions hard-wired into the code. The authors argue that that its advantageous to use the proposed solution since the programming effort is significantly reduced in the proposed solution. This is an interesting idea to research about. However this is not similar to our research.

2.8 Summary

In this chapter, we discussed selected efforts regarding data redistribution of dense matrices on 2D logical process topologies. We considered additional relevant prior work by other researchers such as the Data Reorganization Interface, the fftMPI Library [32] (which computes 2D and 3D parallel FFTs), data transpositions in the PUMMA matrix multiplication algorithm, poly-algorithms, efficient data reorganization research, neural network data reorganizations, and redistributing data in cyclic blocks. Despite the discussed efforts about data redistribution libraries to improve the performance of parallel dense matrices, there is no standalone library that redistributes two-dimensional (2D) matrices from one 2D logical process topology to another at constant or varying numbers of processes. Such procedures are tightly coupled with specific algorithmic libraries (like PUMMA [9] or ScalaPack [22]), making it difficult to evaluate the cost of redistributions vs. computation in a potentially suboptimal data distribution. It would also be an important contribution to the literature to know when it is useful to redistribute data involving two-dimensional data layouts in conjunction with parallel algorithms or when not to redistribute data.

CHAPTER 3

METHODOLOGY

In this chapter, we discuss our approach for designing our data redistribution library for dense matrices on 2D logical grid topologies with MPI. The first section of this chapter describes the design methodology, which presents the core ideas and concepts used by the library along with other essential functionality used in the library. Section 3.2 details our implementation of the concepts presented in the first section of this chapter. The implementation discussion elaborates on the data structures used to represent the data layout, process layout, and mapping rules. Finally, Section 3.3 presents the algorithms provided by our library, including how they are coordinated.

3.1 Design methodology

Three key concepts drive our library: the data grid, the process grid, and the data mapping. A data grid is mapped on the process grid according to the rules of a given data mapping. Each component is defined and discussed in turn.

3.1.1 Data Grid

A data grid is a collection of elements assigned to a matrix A with dimensions $M \times N$. A data grid uses a similar notation as a matrix to allow individual elements to be referenced by their coordinates inside the matrix. For the remainder of this work, i will be used to denote the row of an element and j will be used to denote the column of a given element. Both coordinates are zero-based. Figure 3.1 shows an example of a 6×4 data grid of matrix A. As an example of the notation, element 14 in the data grid in Figure 3.1 has the coordinates: $i = 3, j = 2$.

3.1.2 Process Grid

A process grid is a collection of processes assigned to a shape of $P \times Q$ where $P \times Q$ is the total number of processes. A process grid notation allows the process to be referenced by

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23

Figure 3.1 An example of a data grid with $M = 6$ and $N = 4$

its coordinates (p,q) within a process grid, where p maps the P dimension and q maps the Q dimension. For a given grid shape $P \times Q$, the grid has indexes p such that $0 \leq p < P$ and q such that $0 \leq q < Q$, respectively. Figure 3.2 shows an example of a process grid where six processes are mapped to a 3×2 process grid. Different colors represent processes. In this example, the yellow process has the following coordinates in the process grid: $p = 2$ and $q = 0$.

Figure 3.2 An example of a process grid, with three rows and two columns

3.1.3 Data Mapping

Data mapping is the mapping (or allocation) of the data matrix onto the process grid. In other words, we map the process grid onto the data grid. Given a matrix A, the 6×4 data grid from Figure 3.1 is mapped onto a 3×2 (P, Q) process grid of six processes (from Figure 3.2). Here, the number of rows per process is derived from the number of rows in the data grid, M, divided by the

number of rows in the process grid, P. The number of columns per process is similarly derived from the number of columns in the data grid, N, divided by the number of columns in the process grid, Q. in the M dimension is derived from the row size of the matrix A divided by P processes. The number of columns in the N dimension is derived from the column size of matrix A divided by Q processes.

Figure 3.3 An example of a 6×4 data grid mapped onto a 3×2 process grid

Thus, the inner data grid for each process will be 2×2, as shown in picture Figure 3.3. Each inner data grid 2×2, meaning each process gets two rows and two columns worth of data. From the picture (Figure 3.3), we see that each process (show by the different colours) hold four total elements (the count of elements in each color). That is how data is distributed to processes. This is similar but not identical to how data is distributed in Polymath library [26]; when there is a lack of divisibility, this library will usually distribute that imbalance differently than Polymath's distribution algorithms. We change the process grid from $P \times Q$ to any shape $P' \times Q'$ as long as the data grid and the number of processes are fixed (i.e., $P \times Q$ to $P' \times Q'$ with fixed processes). This way, we can compare the effects of different process grid shapes on the total run time performance of the poly-algorithms.

3.1.3.1 Reshaping the Process Grid

Figure 3.4 shows $Q \times P$ (2×3) process grid reshaped from the $P \times Q$ (3×2) process grid in Figure 3.2. That example demonstrates that there are now two rows in the P dimension and three

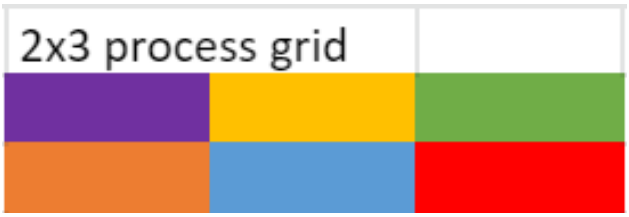

Figure 3.4 A 2×3 process grid

columns in the Q dimension. The new process grid shape shown in Figure 3.4 affects how the data grid is mapped onto the process, resulting in a different mapping as compared to Figure 3.2.

Furthermore, Figure 3.5 shows how the 6×4 data grid from Figure 3.1 will be mapped onto the 3×2 process grid in Figure 3.2. Similar to our prior example, the number of rows per process is derived from the number of rows in the data grid, M, divided by the number of rows in the process grid P (i.e., $M/P = 6/2 = 3$). The number of columns per process is also derived from the number of columns in the data grid, N, divided by the number of columns in the process grid, Q (i.e, $N/Q = 4/3 = 1$, with a remainder of one). In this case, all processes are observed to have an initial data distribution of three rows and one column.

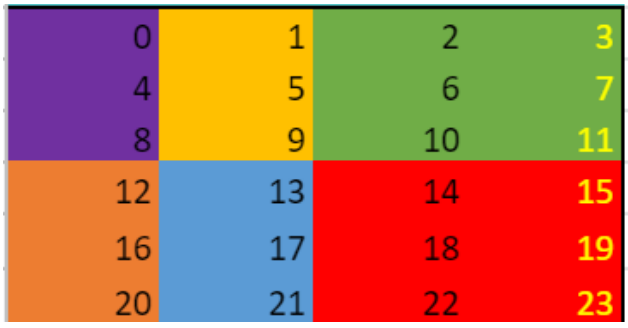

Figure 3.5 Four processes with a 3×1 data distribution and two processes with an extra column

However, because the division for the number of columns per process resulted in a remainder, there will extra columns that need to be assigned to a process; in Figure 3.5, these are the columns with the yellow numbers. As such, we will assign the last two processes in each row (the green and red processes) an extra column (the yellow colored column), as shown in Figure 3.5

above. This particular assignment is the result of our strategy; namely, "pizza cutter." Imagine cutting a pizza into four slices then serving it to only three people. After initially giving everyone an equal number of slices (in this example, everyone gets one slice), one slice remains and you assign it to the third person. As such, the third person will end up with two pieces while the first and second person will each only receive one slice of the pizza. If you have six slices and three people, everyone will be given two slices; if you had eight slices instead of six, you initially give everyone two slices, and then give the remaining two slices to the last person, who ends up with a total of four slices.

Going back to the example shown in Figure 3.5, we used a similar strategy to assign the extra column to the last process in each column in the process grid, resulting in a local data matrix that is 3×2 instead of 3×1. This is because we assign the extra last chunk of data to the last process in a given dimension. This is also illustrated by this pseudo code below (given matrix $A = M \times N$):

```
1 int i_rows_per_process = M / P;
2 int j_cols_per_process = N / Q;
3
4 if( (M % P != 0) && (p+1 == P) )
5     i_rows_per_process += ( M%P );
6
7 if( (N % Q != 0) && (q+1 == Q) )
8     j_cols_per_process += (N % Q);
```

Listing 3.1: One strategy of assigning extra rows and columns to processes

Listing 3.1 demonstrates that if the number of rows M is not evenly divisible by P processes and if the given p coordinate is the last process a row of the process grid, then that process adds the remaining chunk of data in the P dimension to the number of rows it was initially assigned. If the first condition is not true, the process is not assigned any extra rows (i.e., the process will be assigned normal data (rows)). Similarly, if the number of columns N is not evenly divisible by Q processes and if the given q coordinate is the last process in a column in the process grid,

then that process adds the remaining chunk of data in the Q dimension to the number of columns it was initially assigned. Likewise, if any part of the second condition is not true the process is not assigned any extra columns. Lastly, if both conditions are not true, the process is not assigned any extra data (i.e., the process will be assigned normal data (columns and rows)).

3.1.3.2 Additional Examples

Lastly, first consider a 1×4 ($P \times Q$) process grid consisting of four processes (each represented by a different color: purple, orange, yellow, blue), as shown in Figure 3.6 below:

Figure 3.6 A 1×4 process grid

Then consider how the 6×4 data grid from Figure 3.1 would be mapped on that process grid. Figure 3.7 shows the result of this mapping. The resizing of the process grid affects how the data grid is mapped onto each individual process, thus reshaping data. We carried out experiments to see the effects of different shapes of P, Q process grids on the performance of different poly-algorithms.

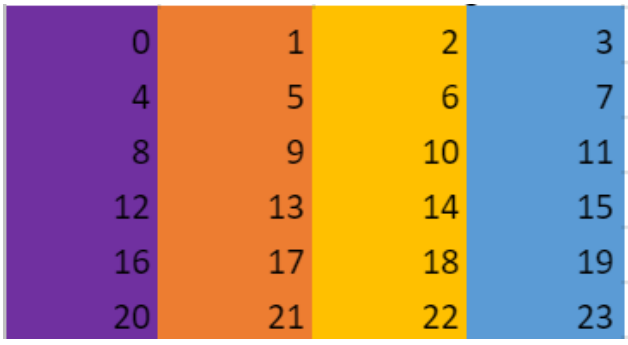

Figure 3.7 A 6×4 data grid

Figure 3.8 and Figure 3.9 provide another example of changing the process grid while keeping the data grid constant. In Figure 3.9, we are resizing the process grid $P \times Q$ from 3×3 to

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

Figure 3.8 A 9×9 matrix mapped on 3×3 process grid

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80

Figure 3.9 A 9×9 matrix after being resized to a 1×9 process grid

1×9, holding all other factors constant. As a result, each process will have a local data matrix that is 9×1 in size; each process will have nine rows and one column locally.

3.2 Implementation

In our code, we have several classes that combine together to help represent the concepts explored in the previous sections. This section explores our design for these classes, along with some of their features.

3.2.1 Data Layout

The `DataLayout` class describes the data grid that all processes will use. Each process has its own copy of the data grid. It represents rows, columns, number of row items in a block, number of column items in block. The main variables we use in this class are defined below:

- **I**: total number of rows of the matrix .
- **J** : total number of columns of the matrix.
- **NIIIB**: number of I items in a block.

- **NJIIB**: number of J items in a block.

We have a method, constructor that creates an object for the `DataLayout` class and updates the number of rows, columns, NIIIB and NJIIB of the object. We also have accessor functions for the NIB, NJB, as shown in Listing 3.2. We can use another function to update rows or columns in layout, or update both rows and columns using a particular function.

```
1 NIB = I / NIIIB
2 NJB = J / NJIIB
```

Listing 3.2: Number of I and J Blocks

After we have all rows and columns, we can swap them to obtain the transposed layout of data using a specific function. The method passes the `oldLayout` object and updates its rows and columns with swapped I and J hence the transposed layout that is used in Section 3.3.1. In addition, we have two methods to update the `DataLayout` object. The first method only updates the rows (i) and columns (j) of the entire layout holding all other factors constant.

The second method generates a new data layout with updated rows, columns, number of i items, and number of j items (NJIIB). The method passes the old data layout by constant reference because it has a non fundamental datatype (`DataLayout`) and passes new I, J, NIIIB, NJIIB by value because they have a fundamental datatype int. In other words, it can be called to create the new matrix size $I \times J$ and resizing the inner blocks using the new values of NIIIB and NJIIB. The method returns an updated `DataLayout` with the new values of I, J, NIIIB, NJIIB. lastly we can cross check `DataLayout` of a process by using a function print to print all variables I, J, P, Q, NIIIB, NJIIB. That way we can know whether the `DataLayout` will fit data exactly depending on the operation or algorithm behavior.

3.2.2 Process Layout

The `ProcessLayout` class is responsible for processes on the process grid. Variables we used in this class include:

- **P**: Processes in the row dimension

- **Q**: Processes in the column dimension
- **p**: Process coordinate p that references P processes
- **q**: Process coordinate q that references Q processes
- **NJBPP**: Number of column blocks in a process.
- **NIBPP**: Number of row blocks in a process.

The `ProcessLayout` class has P, Q, p, q and bases on the MPI communicator's group of ranked processes. The layout also has accessor functions that access different member objects of this class: In MPI, a communicator is a group of all processes that communicate to one another and a unique context for that communication [14]; each communicator has a particular size which depends on the number of processes in that communicator. We use an MPI API to access the size of a communicator.

Each element in a `DataLayout` is locally owned by process p in the row dimension, and it is locally owned by process q in the column dimension. We use accessors, methods to access the references of process coordinates p, q. Each process has a unique identification referred to as its rank; we use an accessor function to access the rank of a given MPI process in the underlying communicator.

We also calculate the number of column blocks in a process (NJBPP) using a method that takes in the number of blocks in the column dimension and divides it by the number of processes in the column dimension, Q. We use an accessor function to access this NJBPP. We also specify `constexpr` to suggest to the compiler to evaluate the value of NJBPP at compile time hence improving the performance of the program.

We calculate the number of row blocks in a process in a similar way to the number of column blocks in a process except that this time the method passes the number of blocks in the row dimension and divides it by the number of processes P in the row dimension.

3.2.3 Mapping Rules class

In this class, we discuss mapping patterns and rules between elements and processes (i.e., what elements each process is supposed to own). Variables that are defined in this class include:

- **local_i**: local row block, process index.
- **local_j**: local column block, process index.
- **global_i_block** : global row index of a block.
- **global_i_block**: global column index of a block.
- **i_size**: total number of rows

This class uses the `DataLayout` and `ProcessLayout` objects to map the process grid onto the data grid and allocate normal data and extra data to some processes based on the `MappingRules` object. We figure out which process owns what and which process owns only standard blocks and what processes own extra blocks as explained below: This class contains methods that identify processes that would own standard data and processes that will need extra work, data. We also have a method, constructor that overloads the assignment operator to create an object of `MappingRules` and updates the `DataLayout` and `ProcessLayout` objects.

We also have methods to update the `DataLayout` used by the `MappingRules` after creation. This method takes a `DataLayout` object by reference, and it updates the reference stored in the `MappingRules` object. We also have a similar, but different, method to update the `ProcessLayout` used. Here we pass the reference to a `ProcessLayout` object to the method and the `ProcessLayout` reference inside the `MappingRules` is updated.

In addition, we can also update the `ProcessLayout` and the `DataLayout` at once using an interesting function that seeds two objects: one for the `ProcessLayout`, second for the `DataLayout`. The method updates both objects at the same time using an assignment operator.

When we are mapping the `DataLayout` object onto the `ProcessLayout` object, it is important to mark, know where processes start or end in any dimension. A couple of lines below are discussing accessor methods to the exact start and end of processes. We have a method to

access the first row of a process. It returns a method that fetches P process and finds out where each will start).

We have a method to access the last row index of a process. It returns a method that fetches p process and finds out where each will end. It also determines which processes should own extra data. In our strategy, we allocate extra row data to the last process in the row dimension. Therefore having known that this the last row it will also check if its the last process and allocates extra chunks of data in row dimension to it.

In addition, we used similar functions to access the start of the first column index of a process. We also have a method to access the last column index of a process. It passes a method which fetches q process and finds out where each will end. It also determines which processes should own extra data in the column dimension. In our strategy we allocate extra column data to the last process in the column "q" in the column dimension. Therefore having known that this the last column it will also check if it's the last process and allocates extra chunks of data in column dimension to it.

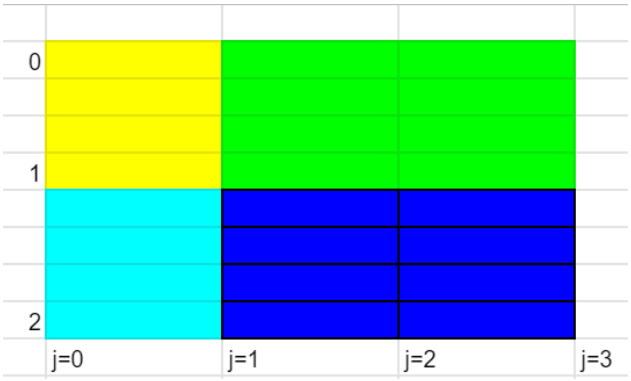

Figure 3.10 An example of start i, j and end i, j

Figure 3.10 and Table 3.1 provide a visual and numerical example, respectively, of the start i, j and end i, j for four processes. In our code, we used start i, j to mean the index of the first row and column of a process and End i, j to mean the index of the last row and column of a process.

Having known the boundaries of each process, we designed a couple of other methods which perform different functionalities to aid, support our mapping patterns: In our code, each

Table 3.1 Numerical example of `MappingRules` methods

Color	Process	Start (i, j)	End (i, j)
Yellow	0	(0,0)	(1,1)
Green	1	(0,1)	(1,3)
Light Blue	2	(1,0)	(2,1)
Dark Blue	3	(1,1)	(2,3)

element can be accessed globally (with the entire matrix, block) and locally (within a given process). This extends to our blocks, they can be accessed both locally and globally. Each element has a local location and a global location In cases we have a local row block index and we need to know its global position. we use a method which converts local row block index to global row block index. The method passes the local row block index and returns the global row block index of an item. We have another method which performs the reverse, a method which converts global row block index of an item Back to local row block index.

We also designed a method which converts local column block index to global column block index. The method passes the local column block index and returns the global column block index of a global item (i, j). However, we also have another method which performs the reverse. The method converts the global column index of an item back to the local column block index. We apply this function when we finding the local process, block column index of a global item given the items global column index "J".

We also used virtual methods to provide a possibility for the methods to be overwritten in the child class (i.e., we override a couple of the virtual methods in the `SpecialRules` class which is a child class to `MappingRules` class). A couple of these interesting methods have been discussed below:

A virtual method that calculates the row size (i size) of a localblock (i.e., how many rows are in the local block). The method passes the local_i, converts local_i into global_i_block and returns another method which passes global_i_block and returns the i_size of the local block.

A virtual method that calculates the row size (i size) of a block (i.e., how many rows are in the block). The method passes the block_i. The method determines whether the row size has extra blocks or whether it has standard blocks.

A virtual method which calculates the owner of the global index after a given operation such as transpose. The method passes the global i index and the global j index. It calculates Pprime and Qprime. Pprime owns the global I index and Qprime owns the global J index. In case Pprime is greater or equal to P(total number of processes in the P processes). Then it sets Pprime to be equal to be the last process in the P dimension (i.e., pprime == (p_layout.P - 1)). This is also our strategy to assign the extra work (cols or transposed rows) to the last process. The method returns the rank of the process that owns global I, J after a given operation (such as transpose).

A method to calculate blocks per process. It passes i, j blocks per process by reference. This method calls other two methods to calculate the i, j blocks per process. Thus ending up with blocks per process.

3.2.4 Special Rules

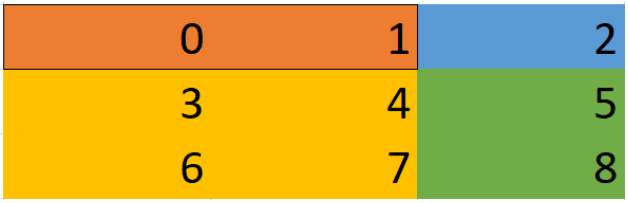

Figure 3.11 `SpecialRules` allocating extra work to the first column and extra rows to the last row

The `SpecialRules` class is a variation of the mapping rules class. This class contains rules that are responsible for assigning the extra work (i.e., rotating the matrix by 90 degrees). `SpecialRules` allocates extra chunk of data (columns) to the first processes in the q dimension. These rules perfectly fit the flipping of data hence rotating the matrix at 90 degrees. While `SpecialRules` allocate extra columns to the first q process in the column dimension, our strategy normally assigns extra slack to the last process in a given dimension. This means the application of rules to map data depends on the operation to be carried out. A brief description of how special

rules work is shown in the Figure 3.11. The special rules belong to the `SpecialRules` class, a child class to the `MappingRules` class.

Similar to the `MappingRules` class, In the `SpecialRules` class, we also have a method to calculate the column size of the block. This method overrides this function from its parent class. This is because instead of giving extra columns to the last process, we give it to the first process instead. The extra rows assignment is unchanged.

We also modify the methods to access the first and last process coordinates in the column dimension by assigning the possible extra data to the first q process in the column dimension. Lastly, we also have a method that calculates the owning process, the rank of an element once we provide it with the global coordinates of the element. This method also makes sure that the elements belong to the right processes.

3.2.5 RawBlock

The `RawBlock` class is the class responsible for holding the memory buffer that is large enough to hold all of the elements a process will be responsible for a given `DataLayout` and `ProcessLayout`. This class is created by first providing the number of rows and columns the local process will be responsible for, and is templated C++ templates to allow for flexibility in the type of matrix held(e.g., a matrix of doubles or integers).

We have provided two methods that can populate the local buffer in this class (i.e., populating the local process): one for local population (i.e., for the process(es) that already have all of the data they will use locally), and one for remote population from a predetermined root process. For the former, the function requires the starting row and column coordinates, along with the column size of the global matrix and uses those values to copy the data out of the provided source buffer. The second method utilizes MPI's non-blocking receive function to obtain the entire data set for the process calling the receive. Since the `RawBlock` knows how many elements it will end up holding, it already knows how many items to tell MPI to expect. The only unknown constants, the source rank and the MPI tag, are expected to be provided by the user[1]. Since this function is non-blocking, it returns the `MPI_Request` to the user to check on completion at a later

[1]MPI communication functions do not currently work with C++ templates, so extra work will be needed in the future to fully support templates in the `RawBlock` class.

point. Finally, for our tests, the matching `MPI_Send` call is called from a helper function that can be used to distribute a matrix from a root process.

In addition to accessor functions for the `RawBlock`'s row size, column size, and total number of elements, the `RawBlock` class also has methods to send and/or receive a chunk of its buffer. The latter two functions are useful for the algorithms in Section 3.3 that can vary in how many items they send at once[2]. These methods utilize non-blocking MPI communications, and therefore return an `MPI_Request` to the caller.

A method is also provided for local transfers between two `RawBlocks`. In this function, the caller provides the local i, j coordinates for each `RawBlock`, along with how many elements to copy over. The `RawBlock` object calling the method is used as the destination, while the provided `RawBlock` object is used as the source location in the underlying `memcpy`.

3.3 Algorithms

In this section, implement the libraries' functionalities. We also provide some code excerpts (see Appendix B) and figures to further illustrate how our algorithms operate.

3.3.1 Matrix Transpose

Transposing a matrix involves switching the row and column indices of matrix A; the product of this operation is another matrix, often denoted by A^T. An example of this product is shown in Figure 3.13, with the original matrix before the transpose in Figure 3.12.

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23

Figure 3.12 Before Transpose

In our matrix transpose algorithm, we begin with creating a new `DataLayout` object that contains the new dimensions of the transposed matrix, A^T. We then create a `MappingRules`

[2]Note that the current implementation of these functions does not allow for "slicing" of the local matrix (e.g., the functions cannot chunk only a given column). The chunks must be contiguous in the local buffer.

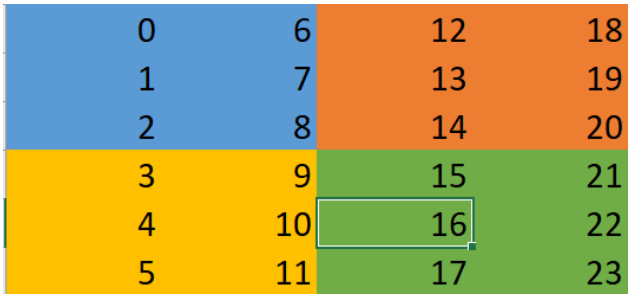

Figure 3.13 After Transpose

object using this new `DataLayout`. Seeding this `MappingRules` object with the "transposed" `DataLayout` helps our library correctly assign extra work to the appropriate process(es) even when rows and columns are swapped. Since each new local description will have a new `RawBlock`, each process will (temporarily) have a new buffer until the transpose is complete. Our algorithm also assumes a constant process grid during the operation, so the present implementation will only be able to transpose a matrix ($M \times N$) for a given process grid ($P \times Q$) if all of the following are true: $M/P \geq 1$, $N/P \geq 1$, $N/Q \geq 1$, and $M/Q \geq 1$. In our implementation, the matrix is not required to be square as long as the aforementioned conditions are met.

Similar to MPI point-to-point communication, we break our algorithm into two phases, a sending side and a receiving side. The sending phase is focused on making sure that all the data is sent to where it needs to go, while the receiving phase is focused on making sure all the data ends up where it needs to be. The receiving phase also handles any data that is to be locally transferred to the new buffer. The messages from different processes are identified by tags; in each case, we used the global index of elements as unique identification tags for messages. Before the sending phase, we also initialize a vector object that will store all `MPI_Requests` produced in each phase.

3.3.1.1 Sending Phase

For the first step of the sending phase, the algorithm first finds the first and last coordinate, in each dimension, that a given process currently holds (done with the functions from the original `MappingRules` object). From there, the algorithm iterates over each element in that range and "transpose" the coordinates so that it now knows where this element will end up in the transposed

matrix. After those calculations, the algorithm can then ask the `MappingRules` built with the transposed `DataLayout` which process rank owns this element. With this new rank, the underlying `RawBlock` is asked to send the element[3] to the process rank identified, using the element's coordinates as the tag. Because the `RawBlock` functions expect local coordinates, the algorithm first converts the coordinates from the global matrix to the coordinates in the local buffer. If the new rank is the same rank as our process, nothing happens and the algorithm moves onto the next element. For each send, the `MPI_Request` is saved into the vector to be checked on later.

3.3.1.2 Receiving Phase

For the receiving phase, the first step is to make a new memory allocation for the resulting local portion of the transposed matrix each process will end up with. We do this by using the "transposed" `DataLayout` and `MappingRules` to create a new `RawBlock`. Once finished, the algorithm then finds the first and last coordinate, in each dimension, that a given process will end up holding after the transpose is finished. For each of the elements in this range, the algorithm figures out the "non-transposed" version of the coordinates. With these coordinates, the algorithm then asks the original `MappingRules` which process rank owns these coordinates. Like in the sending phase, the algorithm then asks the new `RawBlock` to prepare to receive the element from the process rank identified, using the element's original coordinates as the tag. The returned `MPI_Requests` are also pushed into the vector. If the rank is the same as the current process, then it instead asks the new `RawBlock` to transfer the element from the old `RawBlock`.

Lastly, the algorithm then waits on all the `MPI_Requests` generated. Once this has finished, each process should now correctly have their chunk of the new, transposed matrix. Before finishing, the algorithm takes the new `DataLayout`, `MappingRules`, and `RawBlock` and replaces the old versions of them. This allows further algorithms to be run on the matrix.

3.3.2 Resizing the Process Grid

We have two algorithms for resizing the process grid. They are similar but differ in the number of items that can be sent or received from one process to another at one time. The first

[3] In the future, we aim to optimize this further by grouping items together, using collective MPI calls, and/or using blocks to send more efficiently. The algorithm described in Section 3.3.3 features one such optimization.

version can only send or receive one element per message while the second version can send or receive more than one element at a time. The two versions are explained below, and both versions can only handle a resize such that the total number of processes involved must remain the same ($P \times Q = P' \times Q'$).

Similar to the transpose algorithm, we begin this algorithm by first generating a new `ProcessLayout` as well as a `DataLayout`. While the actual shape of the matrix is not changing, we need to update some of the variables used in the `DataLayout` so that other algorithms will be able to use the resulting matrix (and `DataLayout`) in other algorithms. The new `ProcessLayout` is generated from the new P and Q provided to the algorithm. The two of these objects are then used to create a `MappingRules` object that can be queried to find the owners of elements in the matrix. Further, we again create a vector to hold all of the `MPI_Requests` generated in each phase.

3.3.2.1 Sending Phase

Having calculated where each process starts and ends in the global matrix, the algorithm can now begin sending out elements. As the algorithm iterates over the elements a process owns, it queries the new `MappingRules` object for the new owner of the current element. If the owner is a different process, the algorithm asks the `RawBlock` to send that item to the new owner, using the element's coordinates as the tag for the MPI communication. Otherwise, the algorithm simply moves onto the next element. Any `MPI_Requests` are appended to the current vector of `MPI_Requests`. In short, the sending phase for the first resizing algorithm is nearly identical to the sending phase of the transpose, only differing on the query to the new `MappingRules` object.

3.3.2.2 Receiving Phase

Unlike the sending phase, where we concentrate on finding the new owner (destination), here we are more interested in finding the old owner (source). This is possible by using the original `MappingRules` to calculate the owner of the global index. If the old owner is not the current process, then the process knows that it is to receive the item. In this case, the next step is to ask the `RawBlock` to receive the element from the original owner, which results in an `MPI_Request` being returned and then stored in the vector mentioned earlier.

However, if a process is receiving from itself, the algorithm uses the transfer method to transfer data elements from the old `RawBlock` to the new `RawBlock`. It should be noted that the transfer method is like an explicit send and receive locally; that is why we do not use MPI calls for communication. The transfer method transfers one item from one process to another or from the old `RawBlock` to the new `RawBlock`. For future work, we plan to optimize the transfer method to transfer more than one item at a time, which can mean transferring all items in the block at once as long as they belong to the same destination.

The next important step is to call `MPI_Waitall` to wait on all requests current stored in the vector to complete since we used MPI non-blocking calls. Lastly, we update variables in the class to new variables which we used in the resize method. Like with the transpose algorithm, we now update the `DataLayout`, `ProcessLayout`, `MappingRules`, and `RawBlock` with the corresponding objects created by this algorithm.

3.3.3 Second Resizing Algorithm

In our first process grid resizing algorithm, we were sending and receiving elements one at a time. However, we discovered that sending and receiving one item was inefficient and time-costly in terms of communication. So we modified the resizing algorithm to send more than one item at a time. This algorithm still uses the same general structure as the prior algorithm, but it instead collects several sequential elements that all go to the same process together before actually sending them.

We introduced a new variable called `send_amount`. This variable represents the count for every send or receive to or from another process. Unlike our first algorithm, where the count is always to be one, this variable enables the modified resizing algorithm to send varying numbers of elements at one time, based on certain conditions. There are a couple of conditions favoring our modified strategy. For example, when the `MappingRules` reports that the current element has the same destination as the last element, we increase the number of items in the `send_amount` by one. When the destination rank for a given element is different from that of the last element, the algorithm then sends the last chunk since `send_amount` holds how many elements go to that

process. The exact rules for incrementing the `send_amount` are discussed in the next section. Excerpts of the source code will be provided in Appendix B and in full on GitHub.

3.3.3.1 Sending Phase

Since the modified resizing algorithm only changes the number of items sent at one time, the setup of this algorithm is the same as the setup discussed in Section 3.3.2. The first steps of the sending phase are also the same for this algorithm.

As the algorithm iterates over each element, it compares the destination of the current element with the destination of the last element. If the current element has the same destination, the algorithm increments `send_amount` and moves on to the next element. If the algorithm finds that the current element has a different destination, it then asks the `RawBlock` to send the entire chunk of elements that have the same destination. Since the `RawBlock` expects the coordinates of the item to send, the modified resizing algorithm also keeps track of the element's coordinates when the destinations change. As with all algorithms in our library, the algorithm adds the returned `MPI_Request` to the vector as the sends are fired off. While the algorithm iterates over the elements, if it hits the end of a row in the current data grid, it goes ahead and sends the collective of elements it has. This is to avoid having to pack or unpack any data in both phases, as the chunk of memory being transmitted may not be locally contiguous on both processes.

3.3.3.2 Receiving Phase

In the receiving phase, we perform similar operations, as for every `MPI_Send` there should be a matching `MPI_Recv`. However, this time round we are more interested in who we are receiving from and how many items are we receiving. Like in the sending phase, the algorithm keeps track of the source of the current element, and whether the that MPI process is the same as the last element. If so, the algorithm keeps going; if not, the algorithm then asks the new `RawBlock` to receive the current value of the `send_amount` variable at the first element in that chunk. If the data originates from itself, the process instead asks the new `RawBlock` to transfer it from the old `RawBlock`. We then wait on all `MPI_Requests` and update the same objects as the first resizing algorithm.

3.3.4 Rotating a Matrix by 90 degrees

This operation turns the matrix 90 degrees to the left or right (based on the desired direction). The first step is transposing data elements, so we call on the transpose algorithm to transpose the data. Note that corner-turn is given a transposed `DataLayout` from the transpose algorithm. After transposing the matrix, the algorithm simply needs to flip all items to complete the rotation. To assist with this flip, we create a set of rules called `SpecialRules` to allocate extra data appropriately.

We apply this `SpecialRules` object to map the `ProcessLayout` object to the transposed `DataLayout` object. In our transpose algorithm, we swap $M \times N$ and in our resizing algorithms, we resize $P \times Q$, but in corner-turn, all are constant; the matrix is now only flipping. That way, we can neither use old rules nor new rules, so we use special rules that fit and allocate our extra data appropriately; the choice of rules to use is about the desired goal of an operation.

3.3.4.1 Sending Phase

The first step is to find the first and last process coordinates from either dimension. This is to map where exactly each process starts and ends. Each process will use functions from the `SpecialRules` object to determine these values.

At this point, we start planning to flip the matrix rows. We go over all elements in the matrix using two for loops to loop over rows and columns in a process.

```
1 flip_i = curr_i //for flipping the rows
2 flip_j = the_layout.J - 1 - curr_j //for flipping columns.
```

Listing 3.3: Flipping i and j

Listing 3.3 will in turn, flip the matrix by 90 degrees right. We go over the elements so that we know which process will own which element after the flip. Having successfully flipped the elements, we do some math to find out which process rank owns the flipped element. The next step is to identify items which do not "flip." In the case that those items do not flip, the algorithm does nothing; In the case that the elements do flip, the algorithm asks `RawBlock` to send them to the process rank identified, using the element's coordinates as the tag. Because the `RawBlock` functions expect local

coordinates, the algorithm first converts the coordinates from the global matrix to the coordinates in the local buffer. If the new rank is the same rank as the current process, nothing happens and the algorithm moves onto the next element. Lastly, we add the send requests to `MPI_Request`. This tracks all sends on the network to complete.

3.3.4.2 Receiving Phase

Similar to the send side, on the receiver side, We have two for loops to loop over rows and columns of all elements. Like at the send side, we flip current rows and columns of elements at the receiving end as well in order to have the matrix corners turned at 90 degrees and also to know which coordinates own the "flipped" element. Unlike the sending phase where we mainly care about the destination processes and who is sending. At the receive side we are interested in the source of the message and which process are we receiving from. We calculate which process is to receive from what process. In the next step, with the help of the original `MappingRules`, the algorithm finds the local coordinates of the element given its global coordinates (i,j). This is because the `Rawblock` expects local coordinates. Next, the algorithm asks `Rawblock` to receive the item sent to it, from an identified source, rank and a tag (global coordinates of an element) which uniquely identifies each receive message. In case the sending rank is not the same as the receiving rank we receive the element. For each receive, the `MPI_Request` is saved, held until the receive is complete.

However, if the element stays in the same process after flipping (i.e., the sending process is same as the receiving process), the algorithm transfers the element to the new `RawBlock`. It transfers one item at a time, copying the item from the old buffer to the new buffer.

Next, we use `MPI_Waitall` to wait on all send and receive requests to make ensure that each process has the correct data of the flipped matrix.

Finally, the algorithm updates all of the old versions of Class variables, objects such as the `DataLayout` and `ProcessLayout`. This allows further algorithms to be run on the matrix.

3.4 Summary

In this chapter, we first defined three key concepts: the data grid, which allows individual coordinates of a matrix to be referenced; the process grid, which allows individual coordinates of a

process to be referenced; and the data mapping, where we map the process grid onto the data grid. We then presented our object-oriented, C++ implementation of these concepts, and described how they will be used as the basis for our algorithms. Finally, we described the three algorithms our library provides: matrix transpose, process grid resizing, and matrix rotation.

CHAPTER 4

PERFORMANCE EVALUATION

In this chapter, we first discuss the experimental setup (Section 4.1) and describe, in particular, the specifications and configurations of the UTC one-seventeen cluster (117) x86-64 cluster used for our experiments. We then discuss the experiments performed and the input parameters (matrix dimensions, number of processes, etc.) and results we gathered for each test case. Section 4.2 specifically discusses additional Polymath runs and the need to perform them for the purpose of comparison with data redistribution times. In Section 4.3, we provide the results showing tables and graphs generated from our experiments. We then conclude this chapter with comparisons of the key observations from our results in Section 4.4.

4.1 Experimental Setup

One of our goals is to integrate features of our redistribution library with the Polymath library based on different grid shapes, problem sizes (matrix sizes), and numbers of processes. Our second goal is to observe the performance trends and determine whether it is worthwhile to redistribute data or not prior to performing parallel matrix-matrix multiplication (in terms of minimum time to solution). We set up experiments to to test this hypothesis.

The simulations (runs) discussed below were carried out on the 117 cluster at the SimCenter, at the University of Tennessee at Chattanooga. This cluster was used since it is one of the clusters used to obtain performance results for various poly-algorithms from the Polymath library [29]. By using the same cluster, we eliminates some potential sources of variance between our results and the poly-algorithm results. And, what is more important, we can directly compare multiplication times for given problem sizes, linear distributions, and grid shapes with our distribution times without rerunning those problems.

The 117 cluster incorporates 33 computer nodes and one login node. The compute nodes are configured as follows: two (dual-socket) Intel Xeon E5-2680 v4, 2.4GHz chips, each with 14 cores for a total of 28 cores per compute node. There is 128 GB of RAM per compute node for about 4.5 GB of RAM per core. One NVIDIA 16GB P100 GPU with 1,792 double precision cores is available on each node. The theoretical peak performance of about 1 TFLOPs (CPU only) or roughly 5.7 TFLOPs (CPU+GPU). Since Polymath used only multicore BLAS, and did not use the GPUs, we did not use GPUs either. Redistribution algorithms are not floating point intensive, but rather message-passing and indexing intensive[1].

4.1.1 Test Cases

We designed experiments to test whether data redistribution impacts the performance of the algorithms in the Polymath library. This was done by calculating the redistribution costs and comparing it to the matrix multiplication costs. Then we check whether the impact is positive or negative towards a given algorithm's performance, thereby answering the hypothesis of whether it is worthwhile or not to redistribute data prior to parallel computation for minimum time to solution.

Our initial experiments used a square matrix size of $M \times M$, with $M = 20,000$, distributed across 16 processes. This is a one-process per node MPI layout consistent with how Polymath runs were done in Nansamba's thesis [29]. We started testing from the smaller problem size of 20,000 because we wanted to scale upwards to larger identify if there were significant differences in the value of redistribution vs. computation efficiency of the parallel matrix multiplication. We conserved shapes, sizes and data layouts from the Polymath library for validity. We also tested square matrices with $M = 40,000$ and $M = 80,000$ distributed across 16 processes because we wanted to check the redistribution costs when it comes to a larger problem size. The 40,000 square shape ran successfully but the 80,000 case didn't run successfully because when $M = 80,000$, $M^2 = 6,400,000,000$. As 6,400,000,000 is greater than the the maximum integer limit (2,147,483,647), our library is currently not set up to handle this size of a problem. This is left for future resolution (and is essentially a code formulation issue).

[1]Nonetheless, if a future version of Polymath uses GPUs, we would have to consider portions of our redistribution algorithm also to be GPU offloaded, in future.

Note that the change from 20,000 to 40,000 reflects and eight-fold increase in algorithmic work for the poly-algorithms, but only a four-fold work in redistribution effort. Therefore, this size difference tests if, generically speaking, we see continued value in redistribution or computing in place. For larger and larger problems, one asymptotically expects to redistribute more often because of the communication to computation ratio. Nonetheless, both costs are significant in practical cases noted in the next Section.

We also tested our redistribution algorithms on 24 processes on square shapes $M = 20004$, $M = 40008$ because we wanted to see how the increased number of processes could affect the performance of polymath algorithms. These shapes are all divisible by 24 and are close to square shapes 20,000 and 40,000 that we had tested prior.

4.1.2 Results Gathered

The test runs were designed based on the need to align with the performance and use cases of the Polymath algorithms for different grid shapes and sizes. The capacity of the cluster (117) was also another factor we considered. We used 16 processes and each MPI process occupied one node. These 16 nodes were tested with the following grid shapes (P,Q): 1×16, 16×1, 2×8, 8×2 and 4×4 on suquare shapes $M = 20,000$ and $M = 40,000$. As noted above, intra-node concurrency was utilized with parallel BLAS called by algorithms implemented in Polymath, while we performed single-threaded data redistributions in our library[2].

We varied P, Q five times but this depends on the factors of P, Q thus we keep on resizing P, Q until all its factors are exhausted. The total number of processes is held constant through redistribution: $R = P \times Q = P' \times Q'$.

We resized this P, Q (1×16) into four different P, Q process grid shapes; 1×16 to 1×16, 8×2, 2×8 and 4×4. We resized, reshaped each of the above shapes into new P, Q grid shapes; we resized 1×16 to 16×1, 2×8, 8×2, and 4×4. They are four more P, Q shapes we used for resizing grids for 16×1 grid into other shapes. Tables with more varying grid shapes based on the factors of the processes, nodes are in given in the Appendix A.

[2]Multithreaded redistributions are a generalization to be considered in future, but not here.

We also tested 24 processes and each MPI process occupied one node. These 24 nodes were tested with the following grid shapes (P,Q): 1×24, 24×1, 2×12, 12×2, 3×8, 8×3, 6×4, and 4×6 on square shapes $M = 20004, M = 40008$. We resized, reshaped each of the above grid shapes into new P, Q grid shapes; we resized 1×24 to 24×1, 2×12, 12×2, 3×8, 8×3, 6×4, and 4×6. They are seven other P, Q shapes we used for resizing grids for 24×1 grid into other shapes.

From the experiments, we measured the time it takes for a given matrix shape and process grid to covert P, Q to the new shape P' and Q' (i.e., time to resize the matrix). The experiments were run ten times and the maximum time over all processes was recorded for each run. Then, the mean of the maximum run times was used to as the measure performance, together with the standard deviation of this mean (standard error)[3]. This mean value is what we consider to be the redistribution (or reshaping) time ($\pm$ the standard error). The redistribution times are listed in Tables 4.1, 4.2, 4.3, and 4.4 discussed below in Section 4.3.

4.2 Additional Polymath Runs

One of our goals is to integrate our Data Redistribution Library with the Polymath library. Based on that goal, we integrated some of the results from the polymath dense matrix multiplication experiments presented in Nansamba's thesis [29][4]. In particular, we used the results from the 16 processes (with one MPI process per node) for square matrix shapes $20,000\times 20,000$ in that work. For 16 processes five grid shapes were used: 1×16, 16×1, 2×8, 8×2 and 4×4. For our calculations, we infer that if we have a given grid shape, we can get the results show in Nansamba's thesis and thus when we resize, we can compare the original shape's performance to another shape's performance plus the cost of redistribution. Further, we define A as the time a given shape (P_A, Q_A) and matrix size completes its matrix multiplication using the fastest poly-algorithm, and define B as the time it takes to go from P_A, Q_A to P_B, Q_B plus the time for the shape P_B, Q_B to complete its optimal poly-algortihm. If the time recorded for B is less than, or close to, A, then it

[3]This is the same measurement methodology used by Nansamba in her thesis for measuring Polymath algorithm times [29]. It is statistically acceptable, from an engineering viewpoint, to model these as a normal distribution. We did not formally test for normality, but this is not a usual step in taking parallel run data.

[4]We are still in the software engineering process of creating a unified code application that does the work in a single program run. That is also future implementation work, not crucial to demonstrating our hypothesis.

means redistributing is likely effective for a given initial shape, matrix size, and final shape. For example, for the 16×1 shape, mm5_row is the best algorithm, it takes 2.68 seconds to perform a matrix multiplication, and 8×2 is another shape with mm5_row as the best algorithm and its cost of multiplication is 2.47 seconds. Therefore, we compare the performance of 16×1 which is 2.68 seconds and the performance of 8×2, which is 2.47 plus the cost, time it takes to reshape 16×1 to 8×2, in seconds.

As part of the experiment, we performed additional tests for the poly-algorithms in order to obtain more results sets for larger square shapes. The new tests include using square shapes, $40,000 \times 40,000$, $20004 \times 20,004$ and $40,008 \times 40,008$. We started with 16 processes with 5 grid shapes: 1×16, 16×1, 2×8, 8×2 and 4×4 ran on square shapes $40,000 \times 40,000$ In addition, we used 24 processes with 7 different grid shapes on square shapes $20,004 \times 20,004$, $40,008 \times 40,008$. Running larger shapes was vital in our research. This is because with this larger shape, we were able to obtain different results cases that are data redistribution worthwhile. We also noticed process grids that perform better after reshape. The results of the Polymath runs are presented in Appendix A, and discussed with our results in Section 4.3.

4.3 Discussion of Results

The first test case was a square shape of size $20,000 \times 20,000$ and we utilized 16 nodes for the five possible grid shapes mentioned in Section 4.2. For each initial shape, we choose the best algorithm (denoted as the original Algorithm in Table 4.1) time and compared it to the new shape (Alg)time plus reshape time (i.e., redistribution algorithm time = New Alg time + reshape time). This concept is exemplified in Table 4.1. Our main goal was to compare the original algorithm's time and redistribution algorithm time for the different shapes to determine whether it is worthwhile to redistribute data. Table 4.1 shows the best algorithm and the redistribution time for each shape to all other possible shapes using 16 processes.

The results for the 16 processes, with matrix size $20,000 \times 20,000$, shows no cases where redistribution of data improves the performance of the poly-algorithms. It also clearly shows that there are no shapes for which it is worthwhile to redistribute to yield a better time to solution for the subsequent parallel matrix multiplication. Grid shapes such as 16×1 and 1×16 show that

Table 4.1 Results for reshaping a matrix size, $M = N = 20,000$ using 16 processors

Shape	**Original Algorithm**	**Original Alg Time**	**New. Shape**	**Dest. Algorithm**	**New. Alg Time**	**Reshape Time**	**Total Reshaped + Multiplication, NewAlg Time**
16x1	mm5_row	2.68	16x1	mm5_row	2.68	0	2.68
		2.68	8x2	mm5_row	2.47	2.24	4.71
		2.68	4x4	cannon_cg	2.52	2.28	4.8
		2.68	2x8	mm5_col	2.54	2.22	4.76
		2.68	1x16	mm5_col	2.78	2.35	5.13
8x2	mm5_row	2.47	16x1	mm5_row	2.68	2.24	4.92
			8x2	mm5_row	2.47	0	2.47
			4x4	cannon_cg	2.52	2.34	4.86
			2x8	mm5_col	2.54	2.49	5.03
			1x16	mm5_col	2.78	2.56	5.34
4x4	cannon_cg	2.52	16x1	mm5_row	2.68	2.32	5
			8x2	mm5_row	2.47	2.62	5.09
			4x4	cannon_cg	2.52	0	2.52
			2x8	mm5_col	2.54	2.47	5.01
			1x16	mm5_col	2.78	2.56	5.34
2x8	mm5_col	2.54	16x1	mm5_row	2.68	2.34	5.02
			8x2	mm5_row	2.47	2.48	4.95
			4x4	cannon_cg	2.52	2.34	4.86
			2x8	mm5_col	2.54	0	2.54
			1x16	mm5_col	2.78	2.69	5.47
1x16	mm5_col	2.78	16x1	mm5_row	2.68		2.68
			8x2	mm5_row	2.47	2.32	4.79
			4x4	cannon_cg	2.52	2.53	5.05
			2x8	mm5_col	2.54	2.7	5.24
			1x16	mm5_col	2.78	0	2.78

the original shape takes relatively less time than other grid shapes, although we observe that it does not pay to redistribute data for the either 16×1 or 1×16 to other shapes. For example, the cost of multiplication of square matrices of size $20,000 \times 20,000$ on a 16×1 process grid originally takes 2.68 seconds. After reshaping the data to an 8×2 grid, the cost of multiplication

after redistribution increased to 4.71 seconds. The cost of multiplication after redistribution is the cost of multiplication of a shape (original algorithm time) plus the time it takes to convert from one shape to another shape (reshape time). This means that this grid shape performed better with its original shape of 16×1; therefore, there is no need to redistribute data.

None of the grids shapes used alongside the $20,000 \times 20,000$ matrix were worth redistributing. This is because we observe that for a couple of grid shapes, the redistribution time is almost equal, more than cost of multiplication for the original shape. The reshape time for all grid shapes ranges between 2.22 to 2.7 seconds. This is not efficient because it makes the cost of redistribution high and in turn increases the cost of multiplication yet one of our goals is to minimize the cost of computation, multiplication in the Polymath library.

Since the $20,000 \times 20,000$ matrix had no shape that was worthy of redistribution, we decided to test a larger, $40,000 \times 40,000$ matrix (while holding all other factors constant).

The results in Table 4.2 were obtained from 16 nodes using square matrix shapes of size $40,000 \times 40,000$. The table shows the best algorithm (derived by Polymath's set of algorithms) and the redistribution time for each shape to all the other shapes. It also clearly shows which shapes are ought to be redistributed in order to achieve better performance for the Polymath algorithm.

The results show the cases where redistribution using our data redistribution library significantly improve the performance of the poly-algorithms. Reshaping the extreme grid shapes such as 16×1 and 1×16 to other shapes shows that it pays to redistribute data. For example, the cost of multiplication of square shape $40,000 \times 40,000$ on a 1×16 process grid originally takes 72.15 seconds. After reshaping into 4×4 grid the cost of multiplication significantly reduced to 50.2 seconds which was our goal.

It is important to note that for the shapes for which it was worthwhile to redistribute data the decrease in time was approximately more than 20 seconds (i.e., the redistributed cases runs 20 seconds faster).

The reshape time for all grid shapes does not exceed 17 seconds, which is an interesting trend. The best reshape time is 8.81, reshaping 16×1 to 8×2. The worst reshape time is 16.95 reshaping 1×16 to 4×4.

Table 4.2 Results for reshaping a matrix size, $M = N = 40,000$ using 16 processors

Shape	**Original Algorithm**	**Original Alg Time**	**New. Shape**	**Dest. Alg**	**New. Alg Time**	**Reshape Time**	**Total Reshaped + Multiplication, NewAlg Time**
16x1	mm3_row	66.9	16x1	mm3_row	66.9	0	66.9
			8x2	mm3_row	38.85	8.81	**47.66**
			4x4	mm3_row	33.25	15.34	**48.59**
			2x8	mm3_col	40.39	9.65	**50.04**
			1x16	mm5_col	72.15	9.65	81.8
8x2	mm3_row	38.85	16x1	mm3_row	66.9	14.73	81.63
			8x2	mm3_row	38.85	0	38.85
			4x4	mm3_row	33.25	16.26	49.51
			2x8	mm3_col	40.39	9.78	50.17
			1x16	mm5_col	72.15	16.26	88.41
4x4	mm3_row	33.25	16x1	mm3_row	66.9	9.74	76.64
			8x2	mm3_row	38.85	16.62	55.47
			4x4	mm3_row	33.25	0	33.25
			2x8	mm3_col	40.39	10.36	50.75
			1x16	mm5_col	72.15	16.38	88.53
2x8	mm3_col	40.39	16x1	mm3_row	66.9	9.89	76.79
			8x2	mm3_row	38.85	9.82	48.67
			4x4	mm3_row	33.25	16.91	50.16
			2x8	mm3_col	40.39	0	40.39
			1x16	mm5_col	72.15	16.52	88.67
1x16	mm5_col	72.15	16x1	mm3_row	66.9	9.9	76.8
			8x2	mm3_row	38.85	15.73	**54.58**
			4x4	mm3_row	33.25	16.95	**50.2**
			2x8	mm3_col	40.39	10.63	**51.02**
			1x16	mm5_col	72.15	0	72.15

For the $40,000 \times 40,000$ cases from the polymath library, Fox's algorithm (as generalized in Polymath) had the best performance and it is the one we considered to redistribute data.

Table 4.3 Results for reshaping a matrix size, $M = N = 20,004$ using 24 processors

Shape	**Original Algorithm**	**Original Alg Time**	**New Shape**	**New Alg**	**New Alg Time**	**Reshape Time**	**Total Reshaped + Multiplication, NewAlg Time**
24x1	mm3_row	24.26	24x1	mm3_row	24.26	0	24.26
			2x12	mm3_col	13.9	2.53	**16.43**
			12x2	mm5_row	12.06	2.39	**14.45**
			3x8	mm5_col	10.14	2.45	**12.59**
			8x3	mm5_row	9.91	1.48	**11.39**
			6x4	mm5_row	8.6	2.42	**11.02**
			4x6	mm5_col	9.2	2.47	**11.67**
			1x24	mm5_col	24.49	2.68	27.17
2x12	mm3_col	13.9	2x12	mm3_col	13.9	0	13.9
			24x1	mm3_row	24.26	3.14	27.4
			12x2	mm5_row	12.06	2.6	14.66
			3x8	mm5_col	10.14	3.03	**13.17**
			8x3	mm5_row	9.91	2.73	**12.64**
			6x4	mm5_row	8.6	2.8	**11.4**
			4x6	mm5_col	9.2	3.02	**12.22**
			1x24	mm5_col	24.49	2.49	26.98
12x2	mm5_row	12.06	24x1	mm3_row	24.26	2.43	26.69
			2x12	mm3_col	13.9	2.6	16.5
			12x2	mm5_row	12.06	0	12.06
			3x8	mm5_col	10.14	2.71	12.85
			8x3	mm5_row	9.91	2.79	12.7
			6x4	mm5_row	8.6	2.74	**11.34**
			4x6	mm5_col	9.2	2.61	**11.81**
			1x24	mm5_col	24.49	2.61	27.1
3x8	mm5_col	10.14	24x1	mm3_row	24.26	2.51	26.77
			2x12	mm3_col	13.9	3.04	16.94
			12x2	mm5_row	12.06	2.63	14.69
			3x8	mm5_col	10.14	0	10.14
			8x3	mm5_row	9.91	2.65	12.56

Table 4.3 (continued)

Shape	Original Algorithm	Original Alg Time	New Shape	New Alg	New Alg Time	Reshape Time	Total Reshaped + Multiplication, NewAlg Time
			6x4	mm5_row	8.6	3.02	11.62
			4x6	mm5_col	9.2	2.9	12.1
			1x24	mm5_col	24.49	3.23	27.72
8x3	mm5_row	9.91	24x1	mm3_row	24.26	2.49	26.75
			2x12	mm3_col	13.9	2.59	16.49
			12x2	mm5_row	12.06	2.66	14.72
			3x8	mm5_col	10.14	2.69	12.83
			8x3	mm5_row	9.91	0	9.91
			6x4	mm5_row	8.6	2.59	11.19
			4x6	mm5_col	9.2	2.67	11.87
			1x24	mm5_col	24.49	2.78	27.27
6x4	mm5_row	8.6	24x1	mm3_row	24.26	2.45	26.71
			2x12	mm3_col	13.9	2.65	16.55
			12x2	mm5_row	12.06	2.85	14.91
			3x8	mm5_col	10.14	2.86	13
			8x3	mm5_row	9.91	2.87	12.78
			6x4	mm5_row	8.6	0	8.6
			4x6	mm5_col	9.2	2.88	12.08
			1x24	mm5_col	24.49	2.92	27.41
4x6	mm5_col	9.2	24x1	mm3_row	24.26	2.49	26.75
			2x12	mm3_col	13.9	2.84	16.74
			12x2	mm5_row	12.06	2.57	14.63
			3x8	mm5_col	10.14	2.77	12.91
			8x3	mm5_row	9.91	2.83	12.74
			6x4	mm5_row	8.6	2.79	11.39
			4x6	mm5_col	9.2	0	9.2
			1x24	mm5_col	24.49	2.85	27.34
1x24	mm5_col	24.49	1x24	mm5_col	24.49	0	24.49
			24x1	mm3_row	24.26	2.59	26.85
			2x12	mm3_col	13.9	3.33	**17.23**
			12x2	mm5_row	12.06	2.61	**14.67**

Table 4.3 (continued)

Shape	Original Algorithm	Original Alg Time	New Shape	New Alg	New Alg Time	Reshape Time	Total Reshaped + Multiplication, NewAlg Time
			3x8	mm5_col	10.14	3.17	**13.31**
			8x3	mm5_row	9.91	2.71	**12.62**
			6x4	mm5_row	8.6	2.76	**11.36**
			4x6	mm5_col	9.2	2.9	**12.1**

Table 4.4 Results for reshaping a matrix size, $M = N = 40,008$ using 24 processors

Shape	Original Algorithm	Original Alg Time	New Shape	New Alg	New Alg Time	Reshape Time	Total Reshaped + Multiplication, New Alg Time
24x1	mm3_row	63.14	24x1	mm5_row	63.14	0	63.14
			2x12	mm5_col	52.52	10.34	**62.86**
			12x2	mm5_row	47.01	9.47	**56.48**
			3x8	mm5_col	34.88	10.29	**45.17**
			8x3	mm5_row	38.02	5.39	**43.41**
			6x4	mm3_col	39.21	10.33	**49.54**
			4x6	mm5_col	30.97	10.19	**41.16**
			1x24	mm3_col	73.85	5.54	79.39
2x12	mm5_col	52.52	2x12	mm5_col	52.52	0	52.52
			24x1	mm5_row	63.14	10.49	73.63
			12x2	mm5_row	47.01	10.7	57.71
			3x8	mm5_col	34.88	11.13	**46.01**
			8x3	mm5_row	38.02	11.43	**49.45**
			6x4	mm3_col	39.21	11.4	**50.61**
			4x6	mm5_col	30.97	11.56	**42.53**
			1x24	mm3_col	73.85	11.54	85.39
12x2	mm5_row	47.01	24x1	mm5_row	63.14	9.53	72.67
			2x12	mm5_col	52.52	10.46	62.98
			12x2	mm5_row	47.01	0	47.01
			3x8	mm5_col	34.88	10.998	**45.878**
			8x3	mm5_row	38.02	11.48	49.5
			6x4	mm3_col	39.21	10.87	50.08
			4x6	mm5_col	30.97	10.36	**41.33**

			1x24	mm3_col	73.85	10.86	84.71
3x8	mm5_col	34.88	24x1	mm5_row	63.14	10.41	73.55
			2x12	mm5_col	52.52	11.76	64.28
			12x2	mm5_row	47.01	11.11	58.12
			3x8	mm5_col	34.88	0	34.88
			8x3	mm5_row	38.02	11.58	49.6
			6x4	mm3_col	39.21	11.76	50.97
			4x6	mm5_col	30.97	11.64	42.61
			1x24	mm3_col	73.85	11.44	85.29
8x3	mm5_row	38.02	24x1	mm5_row	63.14	10.17	73.31
			2x12	mm5_col	52.52	11.09	63.61
			12x2	mm5_row	47.01	10.89	57.9
			3x8	mm5_col	34.88	11.12	46
			8x3	mm5_row	38.02	0	38.02
			6x4	mm3_col	39.21	10.51	49.72
			4x6	mm5_col	30.97	11.18	42.15
			1x24	mm3_col	73.85	10.71	84.56
6x4	mm3_col	39.21	24x1	mm5_row	63.14	10.47	73.61
			2x12	mm5_col	52.52	11.18	63.7
			12x2	mm5_row	47.01	11.38	58.39
			3x8	mm5_col	34.88	11.64	46.52
			8x3	mm5_row	38.02	10.92	48.94
			6x4	mm3_col	39.21	0	39.21
			4x6	mm5_col	30.97	11.36	42.33
			1x24	mm3_col	73.85	10.74	84.59
4x6	mm5_col	30.97	24x1	mm5_row	63.14	10.62	73.76
			2x12	mm5_col	52.52	11.36	63.88
			12x2	mm5_row	47.01	10.57	57.58
			3x8	mm5_col	34.88	11.16	46.04
			8x3	mm5_row	38.02	11.32	49.34
			6x4	mm3_col	39.21	11.497	50.707
			4x6	mm5_col	30.97	0	30.97
			1x24	mm3_col	73.85	11.41	85.26
1x24	mm3_col	73.85	1x24	mm3_col	73.85	0	73.85
			24x1	mm5_row	63.14	10.57	**73.71**

			2x12	mm5_col	52.52	12.1	**64.62**
			12x2	mm5_row	47.01	10.92	**57.93**
			3x8	mm5_col	34.88	11.9	**46.78**
			8x3	mm5_row	38.02	11.11	**49.13**
			6x4	mm3_col	39.21	11.11	**50.32**
			4x6	mm5_col	30.97	11.6	**42.57**

The results in Table 4.3 and Table 4.4 were obtained from 24 nodes using square matrix shapes of size $20,004 \times 20,004$ and $40,008 \times 40,008$. The tables show the best algorithm (derived by Polymath's set of algorithms) and the redistribution time for each shape to all the other shapes. It also clearly shows which shapes are ought to be redistributed in order to achieve better performance for the Polymath algorithm. The results show several cases where redistribution using our data redistribution library significantly improves the performance of the poly-algorithms. Reshaping the extreme grid shapes such as 24×1 and 1×24 to other shapes shows the highest number of shapes where its' worthwhile to redistribute data. For example, the cost of multiplication of square shape $20,004 \times 20,004$ on a 1×24 process grid originally takes 24.49 seconds. After reshaping into 6×4 grid the cost of multiplication significantly reduced to 11.36 seconds which was our goal. Eighteen cases significantly improved performance with redistribution of square shapes 20,004x20,004 using our library. Nineteen cases improved performance with our redistribution library for square shapes 40,008x40,008. In other words from both Table 4.3 and Table 4.4, there are thirty seven cases of worthwhile redistribution. It is important to note that for most shapes that were worthy to redistribute data for square shapes $40,008 \times 40,008$ the decrease in time was approximately more than 10 seconds (i.e., some of the redistributed cases can run 20 seconds faster).

Graph (Figure 4.1) shows reshaping of 16×1 grid to other shapes: 16×1, 8×2, 4×4, 2×8, 1×16. The color purple represents original algorithm time, color orange represents new algorithm time. The color green is the reshape time (i.e., time taken to convert from one grid shape to another).

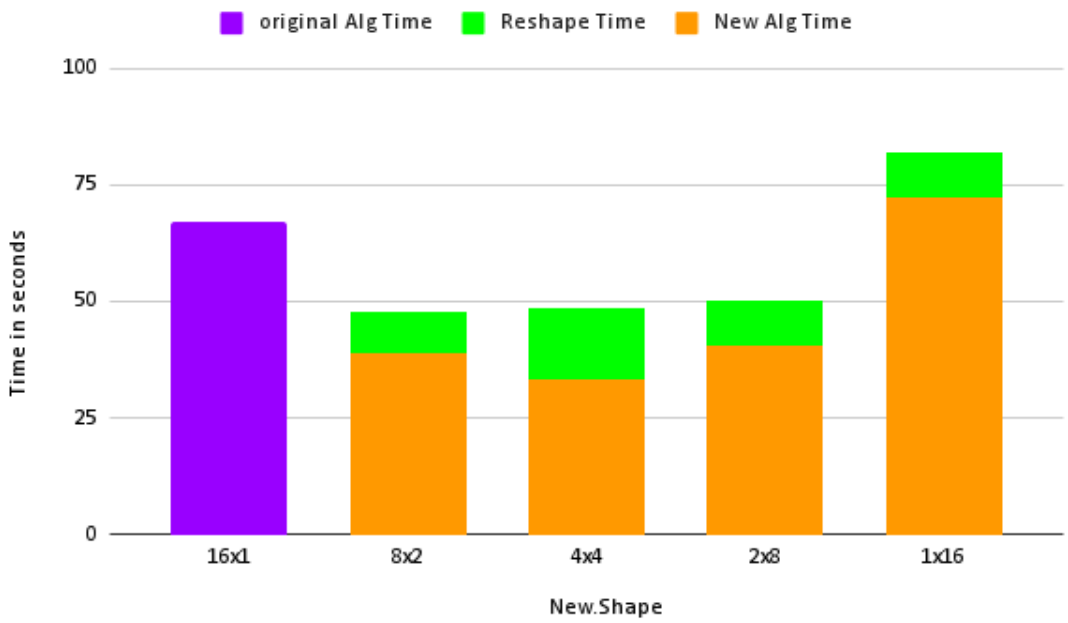

Figure 4.1 Reshaping 16×1 grid to other shapes, matrix size M = N = 40,000

From the Figure 4.1, we see three cases that are worthy of redistribution (i.e., their Total Reshape Time plus the new algorithm multiplication time, is less than original multiplication, algorithm time).

We observed that it takes less than 50 seconds to convert 16×1 to 8×2, 4×4, 2×8 which is lower than the initial multiplication time. This means that shapes 16×1 to 8×2, 4×4, 2×8 perform significantly faster after redistribution. Therefore it is worthwhile to redistribute data in those shapes.

Reshaping from one extreme to another extreme shape takes relatively less time than reshaping to other shapes. However, the total reshape time plus the multiplication time is seen to be high. For example, we see that it takes more than 75 seconds total reshape time plus the multiplication time for 16×1 to 1×16. This is because the cost of multiplication for the poly-algorithms is high so even when the reshape time is good, The multiplication algorithm slows it down hence high redistribution costs (total reshape time plus the multiplication time).

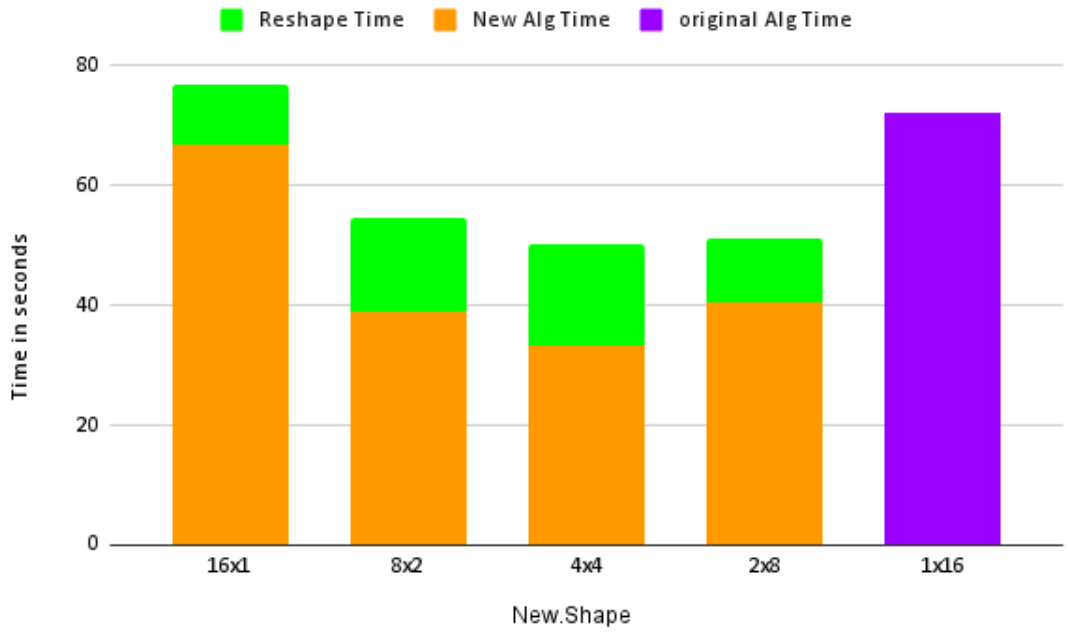

Figure 4.2 Reshaping 1×16 grid to other shapes, matrix size M = N = 40,000

Graph (Figure 4.2), shows reshaping of 1×16 grid to other shapes: 16×1, 8×2, 4×4 and 2×8, matrix size M = M = 40,000. We see a similar trend to the graph in Figure 4.1, for the grid shapes 16×1 to 8×2, 4×4, 2×8. Also, for reshaping from one extreme shape to another extreme has the worst performance.

Graph (Figure 4.3), shows six cases that are worthy of redistribution (i.e., their Total Reshape Time plus new New Alg Time, Multiplication Time is less than original Algorithm, Multiplication Time). We observed that, for a given matrix size $M \times M$, $M = 20,004$ it takes less than four seconds to convert 1×24 to 24×1, 2×12, 12×2, 3×8, 8×3, 6×4, 4×6 which is lower than the initial multiplication time. We observe that in some grid shapes, the redistribution cost, time(New Alg Time + Reshape Time) is lower than the original polymath algorithm time. This means that for grid shapes 2×12, 12×2, 3×8, 8×3, 6×4, 4×6 significantly improve performance after redistribution. Therefore it is worthwhile to redistribute data in those shapes.

Figure 4.4 shows reshaping of 24×1 grid to other shapes: 2×12, 3×8, 8×3, 6×4, 4×6, 1×24, matrix size $M \times M$, $M = 20,004$. We see a similar trend to the graph in Figure 4.3, for the

Figure 4.3 Reshaping 1×24 grid to other shapes, matrix size M = M = 20,004

grid shapes 24×1 to 2×12, 12×2, 3×8, 8×3, 6×4, 4×6. In both graphs reshaping from one extreme shape to another extreme has the worst performance (i.e., has the highest redistribution costs).

Figure 4.5 illustrates a stacked bar graph reshaping grid shape 1×24 to other grid shapes, matrix size $M \times M$, $M = 40,008$. We observe six shapes which significantly improve performance when redistributed to other shapes. Shapes which improve performance after redistribution include: 1×24 to 2×12, 12×2, 3×8, 8×3, 6×4, 4×6. We also observe that its only the extreme case which registers no performance improvement after redistribution and this is because of the polymath multiplication algorithm time is high.

Figure 4.6 shows reshaping of 24×1 grid to other shapes: 2×12, 12×2, 3×8, 8×3, 6×4, 4×6 and 1×24, matrix size $M \times M$, $M = 40,008$. We see a similar trend to the graph in Figure 4.5, for the grid shapes 24×1 to 2×12, 12×2, 3×8, 8×3, 6×4, 4×6 which improve performance after redistribution. For both graphs we observed a trend of 13 grid shapes worthwhile redistribution. Reshaping from one extreme shape to another extreme still has the

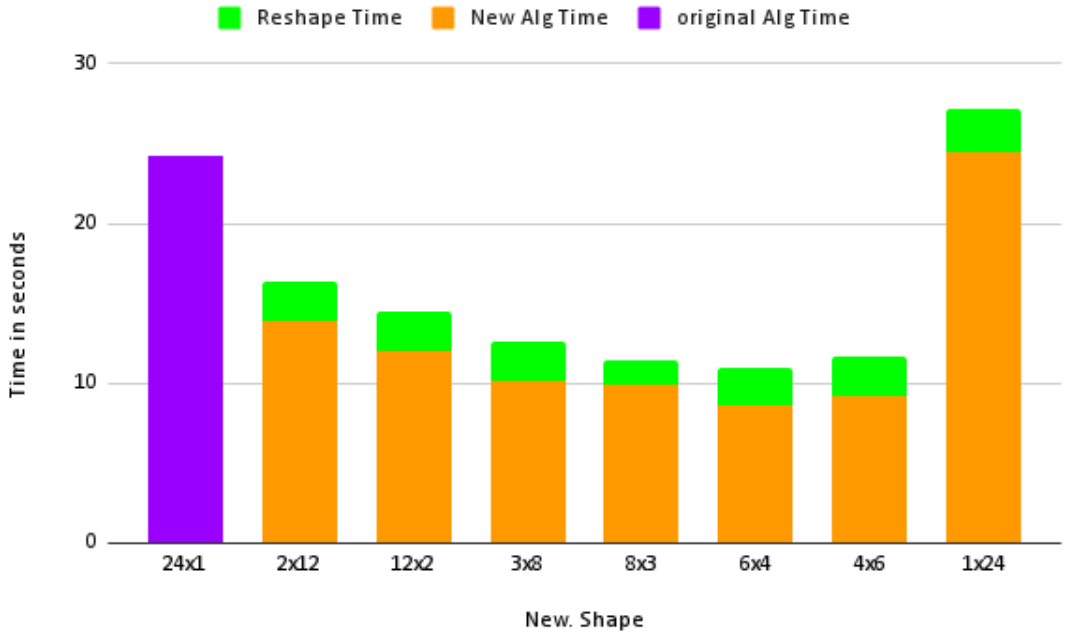

Figure 4.4 Reshaping 24×1 grid to other shapes, matrix size $M \times M$, $M = 20,004$

highest redistribution costs in both graphs. Further, we observed that in figure 4.5 reshaping from grid shape 1x24 to grid shape 24x1 performs slightly better after redistribution and this is the first occurrence in our experiments were it pays off to redistribute from one extreme grid shape to another.

4.4 Comparisons

Performance comparisons follow:

- Using 16 processes, the $20,000 \times 20,000$ square shapes, the Polymath algorithms were significantly faster in their original shape compared to the redistributed version of the algorithm (mm3_row, mm3_row, mm3_row, mm3_col, mm5_col)[5]. Therefore, there was no need to redistribute data.

- For the 16 processes testing the $40,000 \times 40,000$ matrix shape was key in our research. This is because, with this larger matrix size, we could obtain different results cases worthy of data

[5]These are all variants of Fox's algorithm.

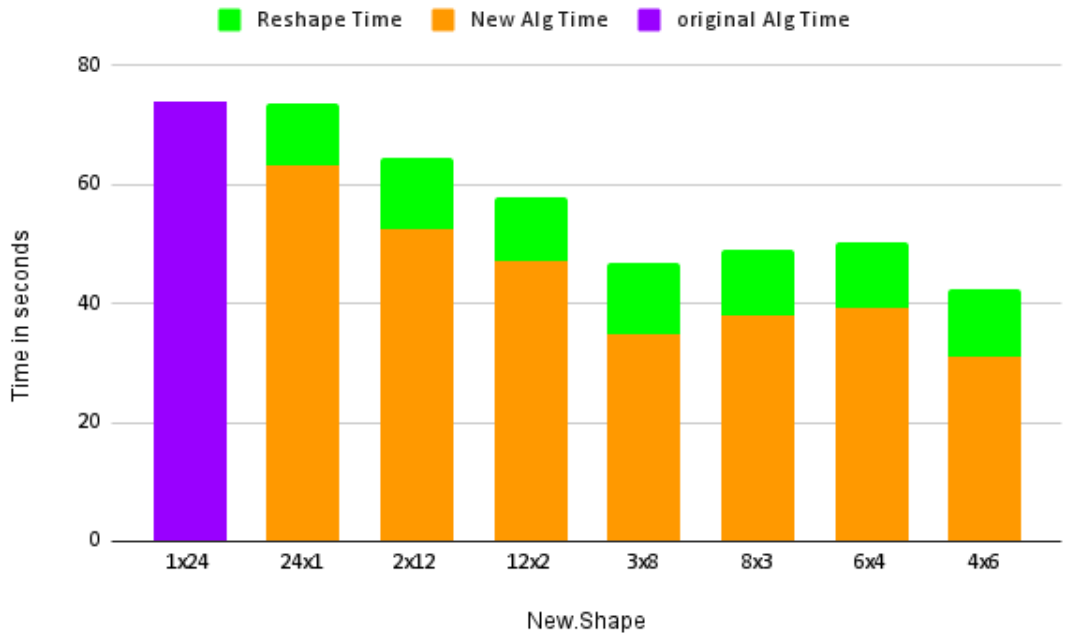

Figure 4.5 Reshaping 1×24 grid to other shapes, matrix size $M \times M$, $M = 40,008$

redistribution. We also noticed process grids that perform better after reshape. The general trend with $40,000 \times 40,000$ shows the significant need to redistribute data.

- From Table 4.1 and Table 4.2, we observe that the cost of computation for multiplication and reshaping of the matrix increases exponentially as the size of the matrix increase. Multiplication of a $40,000 \times 40,000$ matrix size on a 16 process grid takes 66.9 seconds compared to 2.68 seconds for a $20,000 \times 20,000$ matrix on the same grid size. The reshape time for $40,000 \times 40,000$ matrix size from an 8 grid to a 16 is 14.73 seconds as compared to 2.24 seconds for a $20,000 \times 20,000$ matrix size.

- For the 24 processes testing $20,004 \times 20,004$, $40,008 \times 40,008$ matrix shapes, we observed the effect of the increased number of processes towards the performance trend of polymath algorithms. While matrix shape $40,000 \times 40,000$ takes maximum reshape time of 16.95, it takes matrix shape $40,008 \times 40,008$ has only 12.1 seconds maximum reshape time. As

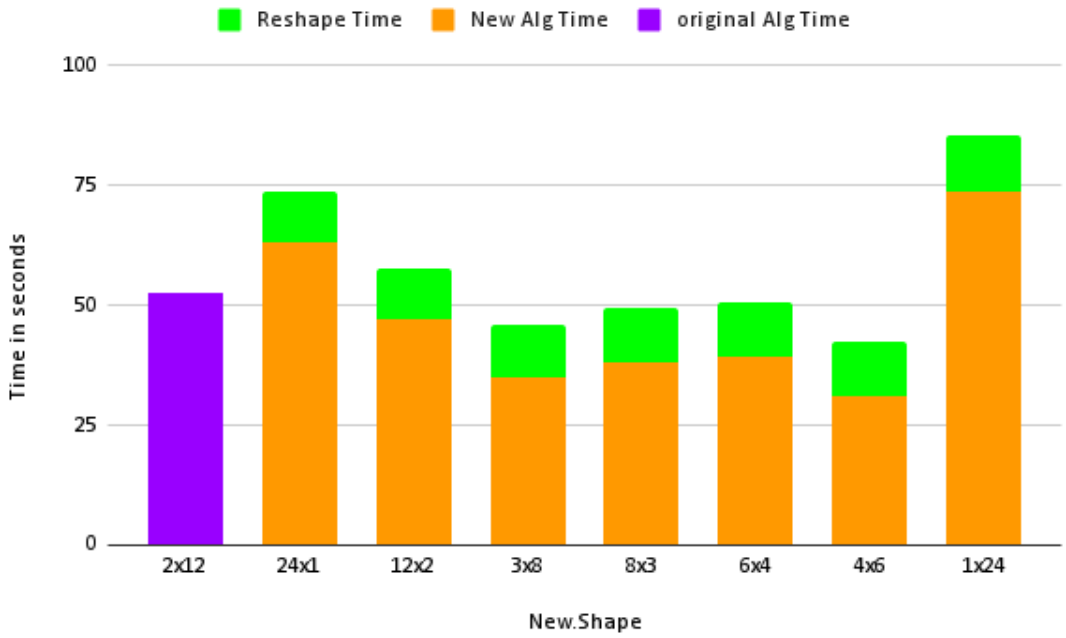

Figure 4.6 Reshaping 24×1 grid to other shapes, matrix size $M \times M$, $M = 40,008$

we increased the number of processes from 16 to 24, there was a reduction in the time to solution.

4.5 Summary

In this chapter, we presented our results from the designed experiments and described the observed performance. We showed the experimental setup based on the goals of our thesis. We included specifications of the cluster used, the test cases, and the results gathered. In Section 4.3, we discussed the results where tables and graphs were presented and explained in detail. Finally, we gave comparisons from the observations we made from the experiments. Overall, we found that redistribution is useful in some situations, but not in others.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

We offer conclusions and recommendations for future work.

5.1 Conclusions

Our primary goal was to generate a library capable of diverse data reorganizations of two-dimensional (2D) dense matrices laid out on 2D logical process topologies in distributed memory; this library was created and tested for correctness. We aimed to develop a high-level Application Programming Interface (API) that works with the Message Passing Interface (MPI) and message-passing primitives to accomplish such data redistributions in data-parallel applications and libraries.

We prototyped data reorganization of 2D matrices on 2D grid shapes with constant total number of processes, restricted to linear data layouts in rows and columns with a particular layout of any load imbalance on rows and columns to the highest process number in the respective dimension. We computed the elapsed time for data redistributions for several square problem sizes (most easy to compare with the Polymath library), as well as non-square matrix and grid cases, during testing.

For specific cases, we measured the elapsed times to redistribute between two grid shapes with the times the fastest Polymath parallel matrix-multiplication algorithm took in each data distribution. We established that it is better to redistribute and compute in the modified data layout with Polymath (perhaps with a different algorithm) in some of our cases, and in others not. A major goal of this thesis is to see when it is and it is not beneficial to do redistributions; the fact that it is sometimes beneficial, but not always, is important because it demonstrates the continued need for polyalgorithmic approaches in Polymath, for example. Overall, even with the singular algorithm for redistribution of grid shapes now supported in this thesis' work, the Polymath algorithmic

policy of compute-as-stored is now generalized to vary from compute-in-place to redistributed-and-compute, when seeking minimal time to solution. This was a key goal of our work.

Specifically, we tested square matrix shapes 20,000 and 40,000 on 16 processes and 20,004 and 40,008 on 24 processes, with one process per node and multicore BLAS kernels. We varied grid shapes, changing how data is mapped across the process grid $P \times Q$, at a fixed number of processes $R = P \times Q$. We tested different matrix sizes because we wanted to check the redistribution costs with varying shapes, and if the cost of redistribution became less significant as lower order work when we increased the total work of matrix multiplication markedly. We observed that it pays to redistribute data for larger shapes as compared to smaller shapes. For example, for square matrices of size $40,000 \times 40,000$, we have six cases in which data redistribution is better and for the square sizes $20,004 \times 20,004$, $40,008 \times 40,008$ we have identified thirty seven cases in which its worthwhile to redistribute data; answering our hypothesis question that it is worthwhile to redistribute data for lower time to solution at least in certain situations. Even though communication represents lower order work than parallel multiplication asymptotically speaking, the factors for finite problem sizes clearly made this a conditional outcome, not always favoring redistribution.

In addition to changing the size of the data in terms of its layout on a logical process topology ($P \times Q$), we also studied and demonstrated data transpose algorithms in Section 3.3.1 and how to rotate data at 90 degrees in Section 3.3.4, which are useful redistribution mechanisms for dense linear algebra in distributed memory computing.

5.2 Future Work

As discussed in Section 3.2.3, our library currently only allows for the mapping pattern demonstrated in Section 3.1.3.1 (e.g., any extra rows and/or columns are assigned to the last process in each dimension in the process grid), and a special variant used to assist when turning a matrix 90-degrees (see Section 3.2.4). However, the library was designed to take advantage of C++ class polymorphism to support future mapping patterns to be be added in as `MappingRules` class extensions. These extensions would simply alter how they report which processes own certain elements, and how many elements each process owns; the other components of our library already

expect such behavior from the abstract `MappingRules` class. Therefore, adding extra mapping patterns can be achieved with minimal changes elsewhere in the code. Additionally, we are also exploring addition of support for a matrix transpose algorithm that can conduct the transpose alongside a remapping of the process grid (e.g., a matrix $M \times N$ on process grid $P \times Q$ is transposed to $N \times M$ on process grid $P' \times Q'$). The simpler form will have $P \times Q = P' \times Q'$. But, the goal is also to allow, eventually varying the total number of processes used, and to support both overlapping and non-overlapping cases. In this thesis, all operations involved a single MPI intra-communicator (single group or clique).

Our aggregation for data reorganization is row-based only at present, and needs to be extended to be multi-row aggregation to reduce latency impacts. This type of gather optimization can best be done with a persistent interface, which would also be excellent for future work. In particular, either derived datatypes for sends/receives or else `MPI_Alltoall*` variants would come into play.

Lastly, the Polymath libraries provides a data-distribution-independent (DDI) set of criteria for matrix multiplication, that are far more general than linear, block linear, or scatter (wrapped) distributions. Further, users can define other valid mappings as extensions. Conforming to the full generality of Polymath's ability to work with matrices mapped in general distributions to/from the sequential (mathematical indexing in rows and columns) to $P \times Q$ grid topologies will be a significant extension of this work. Supporting $P \times Q \times R$ (3D) topologies, including replication, and transpose as well as partial distribution is another set of useful extensions we will consider for future work.

Our work did not exploit intra-node concurrency (threads). We focused on one process per node, and the MPI+X model, where X=threads or OpenMP for intra-node concurrency. However, we have made single threaded redistributions. We did not wish to use multithreaded MPI in this thesis because there is significant evidence of slow down of message passing when full, multithreaded support is activated using `MPI_THREAD_MULTIPLE` [15]. But, we foresee two immediate technologies: partitioned point-to-point communication and persistent collective communication (part of MPI-4) [28], and forthcoming partitioned collective communication (e.g.,

MPI_Alltoall* variants) to be proposed in MPI-5. Focusing on what is actually in MPI-4, we offer these items of future work. Partitioned point-to-point can be used to support multithreaded, efficient communication, that may help ensure that we can completely achieve the injection rate of the network from each node. While we did not test this issue in this work, other work from Bienz et al. [3, 4] shows that injection rates from single processes is insufficient in modern nodes. But, partitioned point-to-point communication aims to resolve this issue and enable greater potential for communication and computation overlap. Second, persistent collective communication [28] could be an efficient way to do certain redistributions, because we expect those planned-transfer operations to be optimizable better than MPI_Ialltoall*, which might be another way to implement our redistributions apart from using point-to-point MPI sends and receives. Regarding the partitioned communication operations, *combining* portions of the redistributions with portions of the poly-algorithms is a future topic. We could be computing on compatible portions of matrices and redistributing. While that is beyond our current scope, it is possible that merging that redistribution and the matrix multiplication can do even better, and partitioned communication could help support such overlapped work (assuming asynchronous progress in the MPI implementation).

REFERENCES

[1] Akin, B., Franchetti, F., and Hoe, J. C. (2015). Data reorganization in memory using 3d-stacked dram. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 131–143. 10

[2] Azari, N. G., Bojanczyk, A. W., and Lee, S.-Y. (1988). Synchronous And Asynchronous Algorithms For Matrix Transposition On MCAP. In Luk, F. T., editor, *Advanced Algorithms and Architectures for Signal Processing III*, volume 0975, pages 277 – 288. International Society for Optics and Photonics, SPIE. 7

[3] Bienz, A., Gropp, W. D., and Olson, L. N. (2018). Improving performance models for irregular point-to-point communication. In *Proceedings of the 25th European MPI Users' Group Meeting*, EuroMPI'18, New York, NY, USA. Association for Computing Machinery. 60

[4] Bienz, A., Olson, L. N., Gropp, W. D., and Lockhart, S. (2020). Modeling data movement performance on heterogeneous architectures. *CoRR*, abs/2010.10378. 60

[5] Cao, Q., Bosilca, G., Wu, W., Zhong, D., Bouteiller, A., and Dongarra, J. (2020). Flexible data redistribution in a task-based runtime system. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 221–225. 8, 9

[6] Choi, J. and Dongarra, J. (1995). Scalable linear algebra software libraries for distributed memory concurrent computers. In *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 170–177. 13

[7] Choi, J., Dongarra, J., Ostrouchov, S., Petitet, A., Walker, D., and Whaley, R. C. (1995). A proposal for a set of parallel basic linear algebra subprograms. In *International Workshop on Applied Parallel Computing*, pages 107–114. Springer. 13

[8] Choi, J., Dongarra, J., and Walker, D. (1993). Parallel matrix transpose algorithms on distributed memory concurrent computers. In *Proceedings of Scalable Parallel Libraries Conference*, pages 245–252. 7

[9] Choi, J., Walker, D. W., and Dongarra, J. J. (1994). Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6(7):543–570. 7, 14

[10] Chung, Y.-C., Hsu, C.-H., and Bai, S.-W. (1998). A basic-cycle calculation technique for efficient dynamic data redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 9(4):359–377. 11

[11] Desprez, F., Dongarra, J., Petitet, A., Randriamaro, C., and Robert, Y. (1998). Scheduling block-cyclic array redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):192–205. 11

[12] Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. S. (1990). A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17. 13

[13] Dongarra, J. J. and Whaley, R. C. (1997). Lapack working note 94 a user's guide to the blacs v1. *Tech.£ eport*. 13

[14] Forum, M. P. I. (2015). *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. High-Performance Computing Center Stuttgart, University of Stuttgart. 1, 23

[15] Grant, R. E., Dosanjh, M. G. F., Levenhagen, M. J., Brightwell, R., and Skjellum, A. (2019). Finepoints: Partitioned multithreaded MPI communication. In Weiland, M., Juckeland, G., Trinitis, C., and Sadayappan, P., editors, *High Performance Computing - 34th International Conference, ISC High Performance 2019, Frankfurt/Main, Germany, June 16-20, 2019, Proceedings*, volume 11501 of *Lecture Notes in Computer Science*, pages 330–350, Frankfurt, Germany. Springer. 59

[16] Gunnels, J., Lin, C., Morrow, G., and Van De Geijn, R. (1998). A flexible class of parallel matrix multiplication algorithms. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 110–116. IEEE. 2

[17] Guo, M. and Pan, Y. (2005). Improving communication scheduling for array redistribution. *Journal of Parallel and Distributed Computing*, 65(5):553–563. 11

[18] Hsu, C.-H., Bai, S.-W., Chung, Y.-C., and Yang, C.-S. (2000). A generalized basic-cycle calculation method for efficient array redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 11(12):1201–1216. 11

[19] Hsu, C.-H., Chen, M.-H., Yang, C.-T., and Li, K.-C. (2006). Optimizing communications of dynamic data redistribution on symmetrical matrices in parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 17(11):1226–1241. 12

[20] Hsu, C.-H., Chung, Y.-C., Yang, D.-L., and Dow, C.-R. (2001). A generalized processor mapping technique for array redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 12(7):743–757. 11

[21] https://software.intel.com/content/www/us/en/develop/documentation/onemkl-tutorial-fortran/top/multiplying-matrices-using dgemm.html (2021). Multiplying matrices using dgemm. 8

[22] Jaeyoung Choi and Dongarra, J. J. (1995). Scalable linear algebra software libraries for distributed memory concurrent computers. In *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pages 170–177. 8, 12, 14

[23] Kalns, E. and Ni, L. (1995). Processor mapping techniques toward efficient data redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1234–1247. 11

[24] Kang, D. and Ha, S. (2020). Tensor virtualization technique to support efficient data reorganization for cnn accelerators. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. 10

[25] Kaushik, S. D., Huang, C.-H., Johnson, R. W., and Sadayappan, P. (1994). An approach to communication-efficient data redistribution. In *Proceedings of the 8th International Conference on Supercomputing*, ICS '94, page 364–373, New York, NY, USA. Association for Computing Machinery. 11

[26] Li, J., Skjellum, A., and Falgout, R. D. (1995). A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. 2, 17

[27] Moreton-Fernandez, A., Sierra, Y. T. D. L., Gonzalez-Escribano, A., and Llanos, D. R. (2021). Operators for data redistribution: Applications to the stl library and raytracing algorithm. *IEEE Access*, 9:38557–38570. 14

[28] MPI Forum (2021). MPI: A Message-Passing Interface Standard, Version 4.0 ; June 9 2021 . Technical report, Univ. of Tennessee, Knoxville, TN, USA. 59, 60

[29] Nansamba, G. (2020). Second-generation polyalgorithms for parallel dense-matrix multiplication. masters thesis, the university of tennessee at chattanooga. Master's thesis, University of Tennessee at Chattanooga. 1, 8, 38, 39, 41

[30] Omiecinski, E., Liehuey Lee, and Scheuermann, P. (1994). Performance analysis of a concurrent file reorganization algorithm for record clustering. *IEEE Transactions on Knowledge and Data Engineering*, 6(2):248–257. 10

[31] Park, N., Prasanna, V., and Raghavendra, C. (1999). Efficient algorithms for block-cyclic array redistribution between processor sets. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1217–1240. 11, 12

[32] Plimpton, S., Kohlmeyer, A., Coffman, P., Blood, P., and USDOE (2018). fftMPI, a library for performing 2d and 3d ffts in parallel. 6, 14

[33] Prylli, L. and Tourancheau, B. (1996). Efficient block cyclic data redistribution. In *European Conference on Parallel Processing*, pages 155–164. Springer. 10, 12

[34] Ramaswamy, S., Simons, B., and Banerjee, P. (1996). Optimizations for efficient array redistribution on distributed memory multicomputers. *J. Parallel Distrib. Comput.*, 38(2):217–228. 11

[35] Siegel, S. F. and Siegel, A. R. (2009). A memory-efficient data redistribution algorithm. In Ropo, M., Westerholm, J., and Dongarra, J., editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 219–229, Berlin, Heidelberg. Springer Berlin Heidelberg. 9

[36] Skjellum, A., Cain, K., and Chelmsford, S. (2002). Document for the data reorganization interface (dri-1.0) standard. In *Proceedings of IEEE Scalable High Performance Computing Conference*. 2, 6

[37] Sudarsan, R. and Ribbens, C. J. (2007). Efficient multidimensional data redistribution for resizable parallel computations. In Stojmenovic, I., Thulasiram, R. K., Yang, L. T., Jia, W., Guo, M., and de Mello, R. F., editors, *Parallel and Distributed Processing and Applications*, pages 182–194, Berlin, Heidelberg. Springer Berlin Heidelberg. 9

[38] Thakur, R., Choudhary, A., and Fox, G. (1994). Runtime array redistribution in HPF programs. In *Proceedings of IEEE Scalable High Performance Computing Conference*, pages 309–316. 11

[39] Thakur, R., Choudhary, A., and Ramanujam, J. (1996). Efficient algorithms for array redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 7(6):587–594. 11

[40] Walker, D., Otto, S. W., Walker, D. W., and Otto, S. W. (1996). Redistribution of block-cyclic data distributions using MPI. 11

APPENDIX A

RAW DATA FROM 117 CLUSTER RUNS

In this section, we have included the new tests for Polymath library on 117 cluster. The graphs include results for the 16 and 24 nominal nodes. We used square matrix sizes of M=N=40,000 for 16 nodes and M=N=20,004 and M=N=40,008 for 24 nodes. Performance is measured as run-time in seconds. The bold figures represent the fastest algorithm in each case.

Table A.1 Run-Time (seconds) $40,000 \times 40,000 \times 40,000$ on 4×4 grid

Algorithm name	**avg max**	**dev max**	**avg min**	**dev min**
mm3_row	**33.254113**	0.633873	29.897032	0.464219
mm3_col	34.382885	1.338233	30.941374	0.93666
mm4_row	33.831046	0.400532	30.497511	0.338755
mm4_col	34.529835	1.351721	31.103818	0.94884
bb	35.5486	0.418969	32.047524	0.32428
cannon_c	35.740456	0.476861	32.364492	0.344026
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	34.285817	0.112407	32.392118	0.113691
cannon_ag	37.351858	0.14432	35.247181	0.139169
cannon_bg	37.547968	0.264141	35.261905	0.260334
summa	N/A	N/A	N/A	N/A
mm5_row	34.036908	0.181725	30.83477	0.148285
mm5_col	33.929283	0.116187	30.822928	0.087851

Table A.2 Run-Time (seconds) $40,000 \times 40,000, \times 40,000$ on 2×8 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	**33.254113**	0.633873	29.897032	0.464219
mm3_col	34.382885	1.338233	30.941374	0.93666
mm4_row	33.831046	0.400532	30.497511	0.338755
mm4_col	34.529835	1.351721	31.103818	0.94884
bb	35.5486	0.418969	32.047524	0.32428
cannon_c	35.740456	0.476861	32.364492	0.344026
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	34.285817	0.112407	32.392118	0.113691
cannon_ag	37.351858	0.14432	35.247181	0.139169
cannon_bg	37.547968	0.264141	35.261905	0.260334
summa	N/A	N/A	N/A	N/A
mm5_row	34.036908	0.181725	30.83477	0.148285
mm5_col	33.929283	0.116187	30.822928	0.087851

Table A.3 Run-Time (seconds) $40,000 \times 40,000, \times 40,000$ on 8×2 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	**33.254113**	0.633873	29.897032	0.464219
mm3_col	34.382885	1.338233	30.941374	0.93666
mm4_row	33.831046	0.400532	30.497511	0.338755
mm4_col	34.529835	1.351721	31.103818	0.94884
bb	35.5486	0.418969	32.047524	0.32428
cannon_c	35.740456	0.476861	32.364492	0.344026
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	34.285817	0.112407	32.392118	0.113691
cannon_ag	37.351858	0.14432	35.247181	0.139169
cannon_bg	37.547968	0.264141	35.261905	0.260334
summa	N/A	N/A	N/A	N/A
mm5_row	34.036908	0.181725	30.83477	0.148285
mm5_col	33.929283	0.116187	30.822928	0.087851

Table A.4 Run-Time (seconds) $40,000 \times 40,000, \times 40,000$ on 16×1 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	**66.900337**	2.021473	57.232377	1.203567
mm3_col	72.976277	3.657021	71.44682	3.620058
mm4_row	70.084598	1.40222	60.266901	0.851871
mm4_col	71.854932	1.399811	70.276195	1.338071
bb	73.45339	1.712192	72.250716	1.654977
cannon_c	68.540718	1.592156	58.975267	0.91932
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	70.252436	1.987365	68.746892	1.987617
cannon_ag	71.470746	2.007677	69.567602	2.011387
cannon_bg	84.58694	0.570892	83.008737	0.570831
summa	N/A	N/A	N/A	N/A
mm5_row	75.076822	1.083263	63.843674	1.129209
mm5_col	N/A	N/A	N/A	N/A

Table A.5 Run-Time (seconds) $40,000 \times 40,000, \times 40,000$ on 1×16 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	75.280708	1.691336	74.021984	1.623202
mm3_col	73.73954	1.794747	66.270352	0.902797
mm4_row	83.628078	1.82404	82.391084	1.784602
mm4_col	74.654968	1.390811	66.135745	1.092652
bb	81.541418	1.867861	80.438429	1.789672
cannon_c	76.81882	0.887838	67.310265	0.650028
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	75.718356	1.928887	74.097879	1.93107
cannon_ag	89.484543	0.792618	87.895551	0.794086
cannon_bg	76.588868	2.466179	74.642605	2.47397
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**72.14789**	1.593486	63.862657	1.506018

Table A.6 Run-Time (seconds) $20,004 \times 20,004 \times 20,004$ on 1×24 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	32.126764	1.138611	31.56031	1.149369
mm3_col	24.571067	0.405932	18.154426	0.39846
mm4_row	37.060411	1.225749	36.4689	1.22918
mm4_col	25.573608	0.632522	18.745904	0.341602
bb	38.69272	0.972386	38.129276	0.964089
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**24.493409**	0.456245	18.170936	0.235164

Table A.7 Run-Time (seconds) $20,004 \times 20,004 \times 20,004$ on 24×1 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	**24.259016**	0.420932	17.606711	0.310861
mm3_col	34.077548	2.075687	33.465203	2.124443
mm4_row	24.656898	0.519319	17.909911	0.504169
mm4_col	33.36001	2.089585	32.762191	2.113774
bb	35.469228	1.004401	4.84613	0.978436
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	24.525999	0.372059	17.821397	0.336642
mm5_col	N/A	N/A	N/A	N/A

Table A.8 Run-Time (seconds) $20,004 \times 20,004 \times 20,004$ on 12×2 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	12.311184	0.190739	9.637038	0.168978
mm3_col	13.061551	0.381882	12.47813	0.378797
mm4_row	12.638561	0.297857	9.789833	0.18354
mm4_col	12.772932	0.373636	12.222574	0.345714
bb	13.658	0.53542	12.073132	0.290487
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**12.05694**	0.354816	9.81608	0.212761
mm5_col	N/A	N/A	N/A	N/A

Table A.9 Run-Time (seconds) $20,004 \times 20,004 \times 20,004$ on 2×12 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	16.553113	0.760291	15.906824	0.761173
mm3_col	**13.904026**	0.452834	10.178519	0.255405
mm4_row	19.11215	0.853616	18.490721	0.845348
mm4_col	14.72021	0.366877	10.495772	0.201792
bb	19.007306	0.764379	14.101596	0.482258
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	14.287087	0.274112	11.369886	0.189267

Table A.10 Run-Time (seconds) $20,004 \times 20,004 \times 20,004$ on 4×6 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	10.045426	0.304763	8.467761	0.224316
mm3_col	10.152595	0.279306	9.361909	0.228181
mm4_row	10.442765	0.257658	8.549252	0.134465
mm4_col	10.387568	0.174849	9.16824	0.135761
bb	10.33925	0.396859	9.075637	0.325211
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**8.598746**	0.133511	7.212485	0.076502
mm5_col	N/A	N/A	N/A	N/A

Table A.11 Run-Time (seconds) $20,004 \times 20,004 \times 20,004$ on 6×4 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	12.069932	0.327869	10.866063	0.297861
mm3_col	11.699591	0.216094	10.544954	0.12409
mm4_row	12.855406	0.32507	11.105147	0.281859
mm4_col	10.944176	0.572482	9.277613	0.394484
bb	12.083687	0.352172	10.465361	0.278856
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**9.196804**	0.132236	8.200973	0.15033

Table A.12 Run-Time (seconds) $20,004 \times 20,004 \times 20,004$ on 8×3 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	11.096646	0.318037	9.236121	0.244168
mm3_col	12.545522	0.477853	11.944165	0.468415
mm4_row	11.361386	0.264217	9.266572	0.159199
mm4_col	12.4835	0.603744	11.823791	0.60914
bb	12.738451	0.533568	10.604853	0.468815
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**9.906581**	0.413822	8.1708	0.456576
mm5_col	N/A	N/A	N/A	N/A

Table A.13 Run-Time (seconds) $20,004 \times 20,004 \times 20,004$ on 3×8 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	12.934521	0.504478	12.341157	0.493996
mm3_col	13.152129	0.175908	12.340832	0.147994
mm4_row	13.301971	0.250738	12.495185	0.249373
mm4_col	12.640834	0.342474	10.403088	0.240938
bb	13.624726	0.423979	11.406636	0.387692
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**10.14335**	0.283347	8.61099	0.209421

Table A.14 Run-Time (seconds) $40,008 \times 40,008 \times 40,008$ on 1×24 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	306.497297	4.40205	305.3313	4.381831
mm3_col	**73.850106**	1.481671	62.329668	0.956101
mm4_row	426.877681	22.28555	425.712302	22.263371
mm4_col	74.614984	1.690119	63.960414	1.340545
bb	382.206457	4.218542	381.20749	4.269252
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	74.533606	1.705108	62.630069	1.285857

Table A.15 Run-Time (seconds) $40,008 \times 40,008 \times 40,008$ on 24×1 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	67.788511	0.895554	55.455848	0.593719
mm3_col	211.941884	86.826466	210.787712	86.800768
mm4_row	63.723799	0.993078	55.392326	1.228665
mm4_col	86.677524	1.351848	85.542924	1.31184
bb	90.675234	1.077123	89.700924	1.049238
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**63.142095**	2.32502	53.4251	1.00021
mm5_col	N/A	N/A	N/A	N/A

Table A.16 Run-Time (seconds) $40,008 \times 40,008 \times 40,008$ on 12×2 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	48.490661	0.571914	41.09129	0.581311
mm3_col	76.090198	1.965552	73.999222	1.973108
mm4_row	48.56446	1.233408	40.896441	0.471799
mm4_col	75.314394	1.061524	73.17014	1.011489
bb	77.34745	0.828151	66.033895	0.886008
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**47.010735**	0.915232	40.108942	0.94563
mm5_col	N/A	N/A	N/A	N/A

Table A.17 Run-Time (seconds) $40,008 \times 40,008 \times 40,008$ on 2×12 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	166.137791	2.613467	164.471879	2.63009
mm3_col	53.79967	1.046521	47.372701	1.062687
mm4_row	542.99172	942.437144	225.830247	5.263537
mm4_col	54.647212	0.543676	45.267252	0.531023
bb	197.114613	2.996338	133.192461	3.359988
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**52.51538**	0.255013	46.721189	0.267024

Table A.18 Run-Time (seconds) $40,008 \times 40,008 \times 40,008$ on 6×4 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	48.033233	1.428994	42.484039	1.350775
mm3_col	**39.213013**	0.475651	36.846906	0.506682
mm4_row	46.005574	0.596393	38.373345	0.381021
mm4_col	40.193191	0.703863	36.852325	0.599731
bb	49.619109	1.698659	46.187199	1.628646
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	39.464672	0.736959	33.321532	0.754795
mm5_col	N/A	N/A	N/A	N/A

Table A.19 Run-Time (seconds) $40,008 \times 40,008 \times 40,008$ on 4×6 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	62.970895	2.201086	55.312284	1.748688
mm3_col	45.418044	1.498516	42.284341	0.716468
mm4_row	65.620416	1.093503	57.15345	1.27367
mm4_col	39.744764	1.054325	34.769688	0.667138
bb	62.27375	1.101748	55.516404	0.734022
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**30.970746**	0.204961	28.595523	0.132144

Table A.20 Run-Time (seconds) $40,008 \times 40,008 \times 40,008$ on 8×3 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	47.419885	1.479107	42.752619	1.325647
mm3_col	46.963586	0.922736	45.477892	0.915265
mm4_row	42.081046	0.797277	36.610543	0.588133
mm4_col	47.295925	0.445786	43.870235	0.493232
bb	57.5859	0.952177	53.801772	0.727345
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	**38.020289**	0.324202	32.397931	0.341339
mm5_col	N/A	N/A	N/A	N/A

Table A.21 Run-Time (seconds) $40,008 \times 40,008 \times 40,008$ on 3×8 grid

Algorithm name	avg max	dev max	avg min	dev min
mm3_row	78.894098	2.163963	76.893758	2.080837
mm3_col	52.998941	0.471795	48.553697	0.924383
mm4_row	96.147678	3.266939	93.966794	3.400491
mm4_col	42.553911	0.455742	36.307063	0.348278
bb	80.34257	2.425976	66.519325	1.994653
cannon_c	N/A	N/A	N/A	N/A
cannon_a	N/A	N/A	N/A	N/A
cannon_b	N/A	N/A	N/A	N/A
cannon_cg	N/A	N/A	N/A	N/A
cannon_ag	N/A	N/A	N/A	N/A
cannon_bg	N/A	N/A	N/A	N/A
summa	N/A	N/A	N/A	N/A
mm5_row	N/A	N/A	N/A	N/A
mm5_col	**34.875001**	0.607888	31.265559	0.392019

APPENDIX B

SOURCE CODE OF ALGORITHMS PRESENTED

The code below is for the Transpose Algorithm. It transposes a matrix by switching the row and column indices of matrix A and the product of this operation is another matrix, often denoted by A^T. This is done by creating a transposed Data Layout $A^T = N \times M$ object, mapping it to the Process Layout object, and then asking mapping rules to manage extra load imbalance. Lines 14-18, we find where each process row, column starts or ends using accessors. Lines 20-46,47, go over all elements in matrix A, as they swap rows and columns indices in lines 24, 25 to transpose the matrix. Lines 27,28, find local coordinates of a given element then ask MappingRules in line 30 to find the process owner of the element. Line 33, if I don't own these the local coordinates ask Blockview to send to them the correct rank by calling MPI_Isend in lines 37-42. In line 44, push the send request to MPI request Vector, save the request to be checked later. Lines 49-102 operate the MPI receive Phase, this is similar to the lines discussed above. Line 104, Wait on all MPI Requests produced. Lines 107-112, Update global variables

```
1  void transpose()
2  { //  vector of MPI requests for this transpose
3    std::vector<MPI_Request> request_vector;
4
5    // new layout we will use to get some info (and then swap to at
         end)
6    DataLayout transposed_layout = DataLayout::transpose(
7      global_description.get_layout(), global_description.
           get_process_layout());
8
9    // Rules for how to deal with extra work. In this case, we can
       use original
10   MappingRules &rules_to_use = global_description.get_rules();
11
12   MappingRules new_rules = rules_to_use;
13   new_rules.seed(transposed_layout);
14
```

```
15  int old_start_i = rules_to_use.get_start_i();
16  int old_end_i   = rules_to_use.get_end_i();
17  int old_start_j = rules_to_use.get_start_j();
18  int old_end_j   = rules_to_use.get_end_j();
19
20  for(int curr_i = old_start_i; curr_i <= old_end_i; curr_i++)
21  {
22    for(int curr_j = old_start_j; curr_j <= old_end_j; curr_j++)
23    {
24      int global_Iprime = curr_j;
25      int global_Jprime = curr_i;
26
27      int local_i = curr_i - old_start_i;
28      int local_j = curr_j - old_start_j;
29
30      int my_rankprime =
31        new_rules.calculate_owner_of_global_index(global_Iprime,
             global_Jprime);
32
33      if(my_rankprime != global_description.get_my_rank())
34      {
35        BlockView<int> &my_block = local_description.get_ij(
             local_i, local_j);
36
37        MPI_Request temp_send_var =
38          my_block.send_part(local_i - my_block.get_sub_i(),
39            local_j - my_block.get_sub_j(),
40            1,
41            my_rankprime,
42            global_Iprime * the_layout.I + global_Jprime);
```

```
43
44          request_vector.push_back(temp_send_var);
45        }
46      }
47    }
48
49    LocalDescription<int> new_description =
50      global_description.generate_local_description(
          transposed_layout, new_rules);
51
52    int new_start_i = new_rules.get_start_i();
53    int new_end_i   = new_rules.get_end_i();
54    int new_start_j = new_rules.get_start_j();
55    int new_end_j   = new_rules.get_end_j();
56
57    for(int curr_i = new_start_i; curr_i <= new_end_i; curr_i++)
58    {
59      for(int curr_j = new_start_j; curr_j <= new_end_j; curr_j++)
60      {
61        int global_Iprime = curr_j;
62        int global_Jprime = curr_i;
63
64        int local_i = curr_i - new_start_i;
65        int local_j = curr_j - new_start_j;
66
67        int my_rankprime =
68          rules_to_use.calculate_owner_of_global_index(
              global_Iprime, global_Jprime);
69
70        if(my_rankprime != global_description.get_my_rank())
```

```
71        {
72          BlockView<int> &my_block  = new_description.get_ij(
               local_i, local_j);
73          MPI_Request temp_send_var = my_block.recv_part(
74            local_i - my_block.get_sub_i(),
75            local_j - my_block.get_sub_j(),
76            1,
77            my_rankprime,
78            curr_i * the_layout.I + curr_j);
79
80          request_vector.push_back(temp_send_var);
81        }
82        else
83        {
84          int old_local_i = curr_j - old_start_i;
85          int old_local_j = curr_i - old_start_j;
86
87          // Note: when going to ourself, we copy one at a time
88          BlockView<int> &my_block_original =
89            local_description.get_ij(old_local_i, old_local_j);
90
91          BlockView<int> &my_block_transposed =
92            new_description.get_ij(local_i, local_j);
93          my_block_transposed.transfer(
94            local_i - my_block_transposed.get_sub_i(),
95            local_j - my_block_transposed.get_sub_j(),
96            old_local_i - my_block_original.get_sub_i(),
97            old_local_j - my_block_original.get_sub_j(),
98            1,
99            my_block_original);
```

```
100         }
101       }
102     }
103
104     MPI_Waitall(request_vector.size(), request_vector.data(),
          MPI_STATUSES_IGNORE);
105
106     // Now, we need to adjust all of the variables in this class!
107     local_description = new_description;
108
109     the_layout = transposed_layout;
110
111     global_description.update_layout(transposed_layout);
112     global_description.update_rules(new_rules);
113 }
```

The code below is for our second resizing Algorithm, It reshapes processes grid $P \times Q$ to $P' \times Q'$ holding the number of processes constant. We used this algorithm for all the experiments in this thesis. The algorithm has both the send and the receive phase. The algorithm attempts to send all elements with the same destination process at once and this makes it different from the first resizing algorithm which sends a count of one element at a time. The send phase starts from line 10 and the receive phase starts from line 98.

```
1 void resize_PQ(int P, int Q)
2 {
3     std::vector<MPI_Request> actions;
4
5     DataLayout &   old_layout   = global_description.get_layout();
6     ProcessLayout &process_grid = global_description.
          get_process_layout();
7
```

```
8   const MappingRules &old_rules = global_description.get_rules();
9
10  DataLayout new_layout =
11    DataLayout::generate_new_layout_blocks(
12      old_layout, old_layout.I / P, old_layout.J / Q);
13
14  ProcessLayout new_grid = ProcessLayout(P, Q, MPI_COMM_WORLD);
15  LocalDescription<int> new_description =
16    global_description.generate_local_description(new_layout,
        new_grid);
17
18  MappingRules new_rules = old_rules;
19  new_rules.seed(new_layout, new_grid);
20
21  int old_start_i = old_rules.get_start_i();
22  int old_end_i   = old_rules.get_end_i();
23  int old_start_j = old_rules.get_start_j();
24  int old_end_j   = old_rules.get_end_j();
25
26  int new_start_i = new_rules.get_start_i();
27  int new_end_i   = new_rules.get_end_i();
28  int new_start_j = new_rules.get_start_j();
29  int new_end_j   = new_rules.get_end_j();
30
31  int last_sender = global_description.get_my_rank();
32  int send_amount = 0;
33  int start_i     = old_start_i;
34  int start_j     = old_start_j;
35
36  for(int curr_i = old_start_i; curr_i <= old_end_i; curr_i++)
```

```
37  {
38    for(int curr_j = old_start_j; curr_j <= old_end_j; curr_j++)
39    {
40      int my_rankprime = new_rules.
           calculate_owner_of_global_index(curr_i, curr_j);
41
42      if(my_rankprime != last_sender)
43      {
44        if(last_sender != global_description.get_my_rank())
45        {
46          int local_i = start_i - old_start_i;
47          int local_j = start_j - old_start_j;
48
49          BlockView<int> &my_block = local_description.get_ij(
              local_i, local_j);
50
51          MPI_Request temp_send_var =
52            my_block.send_part(local_i - my_block.get_sub_i(),
53                               local_j - my_block.get_sub_j(),
54                               send_amount,
55                               last_sender,
56                               start_i * the_layout.I + start_j);
57
58          actions.push_back(temp_send_var);
59        }
60
61        send_amount = 1;
62        start_i     = curr_i;
63        start_j     = curr_j;
64        last_sender = my_rankprime;
```

```
65        }
66        else
67        {
68          send_amount++;
69        }
70      }
71      if(send_amount != 0)
72      {
73        if(last_sender != global_description.get_my_rank())
74        {
75          int local_i = start_i - old_start_i;
76          int local_j = start_j - old_start_j;
77
78          BlockView<int> &my_block =
79            local_description.get_ij(local_i, local_j);
80
81          MPI_Request temp_send_var =
82            my_block.send_part(local_i - my_block.get_sub_i(),
83                               local_j - my_block.get_sub_j(),
84                               send_amount,
85                               last_sender,
86                               start_i * the_layout.I + start_j);
87
88          actions.push_back(temp_send_var);
89        }
90
91        send_amount = 0;
92        last_sender = global_description.get_my_rank();
93        start_i     = curr_i + 1;
94        start_j     = old_start_j;
```

```
95      }
96    }
97
98    last_sender = -1;
99    send_amount = 0;
100   start_i     = new_start_i;
101   start_j     = new_start_j;
102   for(int curr_i = new_start_i; curr_i <= new_end_i; curr_i++)
103   {
104     for(int curr_j = new_start_j; curr_j <= new_end_j; curr_j++)
105     {
106       int old_owner = old_rules.calculate_owner_of_global_index(
            curr_i, curr_j);
107
108       if(old_owner != last_sender)
109       {
110         int local_i = start_i - new_start_i;
111         int local_j = start_j - new_start_j;
112
113         if(last_sender == -1)
114         {
115           // Do nothing
116         }
117         else if(last_sender != global_description.get_my_rank())
118         {
119           BlockView<int> &my_block =
120             new_description.get_ij(local_i, local_j);
121
122           MPI_Request temp_send_var =
123             my_block.recv_part(local_i - my_block.get_sub_i(),
```

```
124                     local_j - my_block.get_sub_j(),
125                     send_amount,
126                     last_sender,
127                     start_i * the_layout.I + start_j);
128           actions.push_back(temp_send_var);
129         }
130         else
131         {
132           int old_local_i = start_i - old_start_i;
133           int old_local_j = start_j - old_start_j;
134
135           BlockView<int> &my_block_original =
136             local_description.get_ij(old_local_i, old_local_j);
137
138           int new_local_i = start_i - new_start_i;
139           int new_local_j = start_j - new_start_j;
140
141           BlockView<int> &my_block =
142             new_description.get_ij(local_i, local_j);
143
144           my_block.transfer(local_i - my_block.get_sub_i(),
145                     local_j - my_block.get_sub_j(),
146                     old_local_i - my_block_original.get_sub_i(),
147                     old_local_j - my_block_original.get_sub_j(),
148                     send_amount,
149                     my_block_original);
150         }
151         send_amount = 1;
152         start_i     = curr_i;
153         start_j     = curr_j;
```

```
154         last_sender = old_owner;
155       }
156       else
157       {
158         send_amount++;
159       }
160     }
161
162     if(send_amount != 0)
163     {
164       int local_i = start_i - new_start_i;
165       int local_j = start_j - new_start_j;
166
167       if(last_sender != global_description.get_my_rank())
168       {
169         BlockView<int> &my_block =
170           new_description.get_ij(local_i, local_j);
171
172         MPI_Request temp_send_var =
173           my_block.recv_part(local_i - my_block.get_sub_i(),
174                              local_j - my_block.get_sub_j(),
175                              send_amount,
176                              last_sender,
177                              start_i * the_layout.I + start_j);
178         actions.push_back(temp_send_var);
179       }
180       else
181       {
182         int old_local_i = start_i - old_start_i;
183         int old_local_j = start_j - old_start_j;
```

```
184
185         BlockView<int> &my_block_original =
186           local_description.get_ij(old_local_i, old_local_j);
187
188         int new_local_i = start_i - new_start_i;
189         int new_local_j = start_j - new_start_j;
190
191         BlockView<int> &my_block =
192           new_description.get_ij(local_i, local_j);
193
194         my_block.transfer(local_i - my_block.get_sub_i(),
195                   local_j - my_block.get_sub_j(),
196                   old_local_i - my_block_original.get_sub_i(),
197                   old_local_j - my_block_original.get_sub_j(),
198                   send_amount,
199                   my_block_original);
200       }
201
202       send_amount = 0;
203       last_sender = -1;
204       start_i     = curr_i + 1;
205       start_j     = new_start_j;
206     }
207   }
208
209   MPI_Waitall(actions.size(), actions.data(), MPI_STATUSES_IGNORE
        );
210
211   // Now, we need to adjust all of the variables in this class!
212   local_description = new_description;
```

```
213
214   the_layout   = new_layout; // ProcessLayout
215   the_p_layout = new_grid;
216
217   global_description.update_layout(new_layout);
218   global_description.update_p_layout(new_grid);
219   global_description.update_rules(new_rules);
220 } // closes resize
```

The code below is for our algorithm to turn a matrix 90 degrees to the left or right based on the desired direction. The algorithm first transposes a matrix by calling the transpose algorithm and then flips it on lines 34, 35 to complete the rotation. We use SpecialRules to allocate extra data appropriately to different processes even after the flip. The algorithm has a send phase starting at line 14 and receive phase starting at line 60.

```
1  void rotation()
2  { // Turns the matrix 90 degrees to the left or right (based on
       direction)
3    transpose();
4
5    // Vector for MPI actions
6    std::vector<MPI_Request> request_vector;
7
8    // A copy to the original rules
9    MappingRules old_rules = global_description.get_rules();
10
11   // Now we just need to flip. We have already updated
12   // the DataLayout in the transpose, and now we just
13   // need to flip (which needs a new rule set).
14   SpecialRules new_rules(global_description.get_layout(),
15                 global_description.get_process_layout());
```

```
16
17  LocalDescription<int> new_description =
18    global_description.generate_local_description(new_rules);
19
20  int old_start_i = old_rules.get_start_i();
21  int old_end_i   = old_rules.get_end_i();
22  int old_start_j = old_rules.get_start_j();
23  int old_end_j   = old_rules.get_end_j();
24
25  int new_start_i = new_rules.get_start_i();
26  int new_end_i   = new_rules.get_end_i();
27  int new_start_j = new_rules.get_start_j();
28  int new_end_j   = new_rules.get_end_j();
29
30  for(int curr_i = old_start_i; curr_i <= old_end_i; curr_i++)
31  {
32    for(int curr_j = old_start_j; curr_j <= old_end_j; curr_j++)
33    {
34      int flip_i = curr_i;
35      int flip_j = the_layout.J - 1 - curr_j;
36
37      int my_rankprime =
38      new_rules.calculate_owner_of_global_index(flip_i, flip_j);
39
40      if(my_rankprime != global_description.get_my_rank())
41      {
42        int local_i = curr_i - old_start_i;
43        int local_j = curr_j - old_start_j;
44
45        BlockView<int> &my_block =
```

```
46            local_description.get_ij(local_i, local_j);
47
48          MPI_Request temp_send_var =
49            my_block.send_part(local_i - my_block.get_sub_i(),
50                       local_j - my_block.get_sub_j(),
51                       1,
52                       my_rankprime,
53                       flip_i * the_layout.I + flip_j);
54
55          request_vector.push_back(temp_send_var);
56        }
57      }
58    }
59
60    for(int curr_i = new_start_i; curr_i <= new_end_i; curr_i++)
61    {
62      for(int curr_j = new_start_j; curr_j <= new_end_j; curr_j++)
63      {
64        int flip_i = curr_i;
65        int flip_j = the_layout.J - 1 - curr_j;
66
67        int my_rankprime =
68        old_rules.calculate_owner_of_global_index(flip_i, flip_j);
69
70        int local_i = curr_i - new_start_i;
71        int local_j = curr_j - new_start_j;
72
73        if(my_rankprime != global_description.get_my_rank())
74        {
75          BlockView<int> &my_block =
```

```
76            new_description.get_ij(local_i, local_j);
77
78          MPI_Request temp_send_var =
79            my_block.recv_part(local_i - my_block.get_sub_i(),
80                       local_j - my_block.get_sub_j(),
81                       1,
82                       my_rankprime,
83                       curr_i * the_layout.I + curr_j);
84
85          request_vector.push_back(temp_send_var);
86        }
87        else
88        {
89          int old_local_i = curr_i - old_start_i;
90          int old_local_j = flip_j - old_start_j;
91
92          // Note: when going to ourself, we copy one at a time
93          BlockView<int> &my_block_original =
94            local_description.get_ij(old_local_i, old_local_j);
95
96          BlockView<int> &my_block =
97            new_description.get_ij(local_i, local_j);
98          my_block.transfer(local_i - my_block.get_sub_i(),
99                    local_j - my_block.get_sub_j(),
100                   old_local_i - my_block_original.get_sub_i(),
101                   old_local_j - my_block_original.get_sub_j(),
102                   1,
103                   my_block_original);
104       }
105     }
```

```
106     }
107
108     MPI_Waitall(request_vector.size(), request_vector.data(),
            MPI_STATUSES_IGNORE);
109
110     // Now, we need to adjust all of the variables in this class!
111     local_description = new_description;
112
113     global_description.update_rules(new_rules);
114 }
```

VITA

Evelyn Namugwanya was born in Mityana, Uganda, on October 10th 1992. She attended high school in Rubaga girl's school and St. Lawrence schools and colleges, Horizon campus, in Kampala Uganda. She graduated in 2015 from Makerere University with a Bachelor of Information Technology. She worked at Makerere University as a systems administrator, doing end-user support for two years before she came to America in 2019 to pursue a master's degree in Computer Science at the University of Tennessee at Chattanooga; She worked as a graduate research assistant for Dr. Anthony Skjellum, focusing on high-performance computing. Evelyn plans to advance her education further through a PhD program in Computational Science at the University of Tennessee at Chattanooga.