

REDUCING COMMUNICATION VOLUME INCREASE IN LATENCY EFFICIENT STORE-AND-FORWARD SCHEME

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Salih Deniz Uzel
September 2025

REDUCING COMMUNICATION VOLUME INCREASE IN LATENCY EFFICIENT STORE-AND-FORWARD SCHEME

By Salih Deniz Uzel

September 2025

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Cevdet Aykanat(Advisor)

Uğur GÜDÜKBAY

ÖZCAN ÖZTÜRK

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

ABSTRACT

REDUCING COMMUNICATION VOLUME INCREASE IN LATENCY EFFICIENT STORE-AND-FORWARD SCHEME

Salih Deniz Uzel

M.S. in Computer Engineering

Advisor: Cevdet Aykanat

September 2025

Sparse matrix operations such as sparse matrix-vector (SpMV) are latency bound applications where number of messages sent by processors dominate the overall communication overhead. In prior work, Store and Forward (STFW) scheme was proposed and implemented to scale such fine-grain operations as SpMV by operating on specialized Virtual Process Topologies (VPTs). However, the STFW scheme incur an increase in the communication volume due to the STFW overhead. This increase in total communication volume disturbs the scalability of the STFW scheme, especially in sparse matrix-matrix (SpMM) kernels where a sparse matrix is post-multiplied by a tall-and-skinny dense matrix. In this work, we propose and implement a Kernighan-Lin-based (KL) heuristic to iteratively improve the one-to-one mapping produced by the initial task partition. The objective of this heuristic is to reduce the increase in the total communication volume to be incurred by the STFW scheme in SpMV, and SpMM type of kernels. We evaluated the effectiveness of the proposed KL-based heuristic in improving the Total Communication Volume and Maximum Weighted Communication Volume metrics through experimental performance comparisons using VPTs of various dimensions and constructive mapping methods.

Keywords: Parallel computing, HPC, sparse matrices, latency-bound applications, communication bottlenecks, topology-aware mapping methods, iterative improvement heuristics, Hypergraph partitioning..

ÖZET

GEÇİKME VERİMLİ SAKLA-VE-YÖNLENDİR ŞEMASINDA İLETİŞİM HACMİ ARTIŞININ AZALTILMASI

Salih Deniz Uzel
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Cevdet Aykanat
Eylül 2025

Seyrek matris-vektör (SpMV) gibi seyrek matris işlemleri, işlemciler tarafından gönderilen mesaj sayısının genel iletişim yükünü belirlediği gecikim-sınırlı uygulamalardır. Önceki çalışmalarda, SpMV gibi ince-taneli işlemleri ölçeklendirmek için Sanal İşlemci Topolojileri (VPT'ler) üzerinde çalışan Sakla-ve-Yönlendir (SvY) şeması önerilmiş ve uygulanmıştır. Ancak SvY şeması SvY yükünden dolayı iletişim hacminde bir artışa yol açar. Toplam iletişim hacmindeki bu artış, özellikle seyrek matrisin uzun ve ince yoğun bir matrisle sağdan çarpıldığı seyrek matris-matris (SpMM) çekirdeklerinde, SvY şemasının ölçeklenebilirliğini bozar. Bu çalışmada, başlangıçtaki görev bölümlenmesi tarafından üretilen birebir atamayı yinelemeli olarak iyileştirmek için Kernighan-Lin tabanlı (KL) bir sezgisel öneriyor ve uyguluyoruz. Bu sezgiselin amacı, SpMV ve SpMM türü çekirdeklerde SvY şemasının yaratacağı toplam iletişim hacmindeki artışı azaltmaktır. Önerilen KL tabanlı sezgiselin etkinliğini, farklı boyutlardaki VPT'ler ve yapıcı atama yöntemleri kullanılarak yapılan deneysel performans karşılaştırmalarıyla Toplam İletişim Hacmi ve Maksimum Ağırlıklı İletişim Hacmi metrikleri açısından değerlendirdik.

Anahtar sözcükler: Paralel hesaplama, Yüksek Başarımli Hesaplama, seyrek matrisler, gecikim-sınırlı uygulamalar, iletişim darboğazları, topoloji-farkında atama yöntemleri, yinelemeli iyileştirme sezgiselleri, Hiperçizge bölümlenme..

Acknowledgement

I would like to thank my advisor, Prof. Dr. Cevdet Aykanat, for his guidance and support during my graduate studies.

I would like to thank the Scientific and Technological Research Council of Turkey (TÜBİTAK) for supporting me during my Master of Science studies under the grant EEEAG-121E391.

Contents

1	Introduction	1
2	Preliminaries	3
2.1	Dense Matrix Partitioning	4
2.2	Sparse Matrix Partitioning	7
2.2.1	Sparse Matrix Storage Formats	8
2.2.2	Communication Matrix	10
2.2.3	Partitioning Strategies	12
2.2.4	Load balancing	14
2.2.5	Symmetric and Nonsymmetric Partitioning	18
2.3	Topology and Message passing	19
2.4	Communication Overhead	23
2.5	Optimization Strategies for Irregular HPC Workloads	24
2.6	Virtual Processor Topology and Store and Forward Scheme	27

2.6.1	Virtual Processor Topology	27
2.6.2	Store-and-Forward Scheme	32
3	Motivation and Related Works	39
4	Iterative Optimization Heuristic for Reducing Total Communi- cation Volume	48
4.1	Reduction of the Store-and-Forward Communication Volume Through Pairwise Task Swap on VPT	49
4.2	Algorithm Design for 2D Topology	53
4.3	Generalization to N-Dimensional Topology	58
4.4	Improved Algorithm Design	61
4.4.1	Generalized Formulation for Communication Volumes In- curred by a Task	61
4.4.2	Optimizing Task Volume Calculations through Post- Processing of the Communication Matrix	62
4.4.3	Extended Notation and Terminology for the Improved Al- gorithm	64
4.4.4	Pre-computation of Communication Volumes for Potential Swap Positions	65
4.4.5	Updating Communication Volumes After Task Swaps	68
4.4.6	General Structure of the Improved Algorithm	70
4.4.7	Complexity Analysis	71

4.5	Vectorization and Parallelization of the KL-based Heuristic	77
5	Experiments	80
5.1	Data Set	80
5.2	Experimental Setup	81
5.3	Experimental Results and Analysis	83
6	Conclusion and Future Work	93

List of Figures

2.1	Sequential Assignment (left), Random Assignment (right)	6
2.2	2-dimensional visualization of the sparse matrix mip1 [1].	7
2.3	Processor-to-Processor Communication Matrix (left), and its graph representation (right) where processors are represented as vertices and communication paths as edges.	11
2.4	Row-wise Partitioning of the Sparse Matrix (left), Column-wise Partitioning of the Sparse Matrix (right).	13
2.5	Row-wise Partitioning of the Sparse Matrix (left), Column-wise Partitioning of the Sparse Matrix (right).	15
2.6	Star Topology (left), Full Mesh Topology (right).	20
2.7	4×4 Mesh Topology	22
2.8	Comparison of a 4×4 physical mesh topology (left) and a virtual processor topology (right).	28
2.9	Dimension-wise processor groups: row-wise (left) and column-wise (right).	29

2.10	Illustration of the neighboring relationships within a processor group: row-wise (left) and column-wise (right).	30
2.11	Stages of Communication in a VPT $T_2(4, 4)$ Using the Store-and-Forward Scheme.	35
4.1	The swapping of the assignments of tasks t_i and t_j within the same row in a 2-dimensional 4×4 VPT $T_2(4, 4)$.	50
4.2	Illustration of pairwise task swaps for a 3×2 VPT $T_2(3, 2)$. The first row shows row-wise swaps, and the last two rows show column-wise swaps.	57
4.3	3-dimensional VPT $T_3(3, 3, 3)$. Task t_i is mapped to p_1 , and its adjacent processors $Adj(map(t_i))$, are highlighted using red, blue, and orange colors.	65
4.4	Plot of the remaining number of available pairwise swaps after each locking operation throughout a single pass for different values of K in 2-Dimensional VPTs.	73
4.5	Plot of the change in the total number of available pairwise swaps after each locking operation throughout a single pass for different values of K in $\sqrt{K} \times \sqrt{K}$ 2-dimensional VPTs.	74
4.6	Plot of the remaining number of available pairwise swaps after each locking operation throughout a single pass for different N-dimensional VPTs with fixed K .	75
5.1	Total communication volume and maximum weighted communication volume for K=1024 processors in VPT dimensions of 2, 6, and 10.	87

5.2	Total communication volume and maximum weighted communication volume for K=1024 processors in VPT dimensions of 2, 6, and 10.	88
5.3	Total communication volume and maximum weighted communication volume for K=512 processors in VPT dimensions of 2, 6, and 9.	89
5.4	Total communication volume and maximum weighted communication volume for K=512 processors in VPT dimensions of 2, 6, and 9.	90
5.5	Total communication volume and maximum weighted communication volume for K=128 processors in VPT dimensions of 2, 5, and 7.	91
5.6	Total communication volume and maximum weighted communication volume for K=128 processors in VPT dimensions of 2, 5, and 7.	92

List of Tables

2.1	Processor communication pattern after row-wise partitioning . . .	11
5.1	Properties of the square sparse matrices used in the experiments.	81

Chapter 1

Introduction

Scientific and technological progress is increasing the demand for high computational capacity. Scientific computing, engineering problems, artificial intelligence, and data-intensive applications often require linear algebra operations such as matrix-vector multiplication and matrix-matrix multiplication. And the required computational capacity increases polynomially with the size of the matrix. To complete computations within a specific time frame, the workload is divided into smaller tasks and distributed across computing units to be executed in parallel using high-performance computing (HPC) or other parallel computing environments. In HPC, the goal is to distribute the workload across computing units in a balanced manner, including both computation and communication. The desired outcome is for each unit to complete its assigned tasks as quickly as possible, collectively minimizing overall execution time. However, many problems have a heterogeneous structure, which can lead to nonuniform workload distribution.

Sparse matrix-vector (SpMV) and sparse matrix-matrix (SpMM) multiplications are fundamental operations used in many HPC applications. Storing only nonzero elements and performing linear algebra operations on these compressed nonzero entries introduces several challenges. The irregular distribution of nonzero entries and low density in sparse matrices make them difficult to partition and distribute across parallel computing units. These partitions often require

data from other partitions, leading to high messaging traffic and irregularities in the HPC system. This irregular communication results in increased total number of messages transmitted over the network, leads to higher latency, congestion, and communication volume overhead. Balancing these communication-induced overheads is a critical challenge in HPC, as it directly impacts the scalability and performance of applications on the underlying infrastructure.

This irregular communication pattern often makes SpMV and SpMM latency-bound applications. To address this, Selvitopi and Aykanat [2] proposed a novel virtual processor topology (VPT) combined with a store-and-forward communication scheme. In this approach, tasks are organized into the VPT topology, and communication between tasks is realized by the store-and-forward scheme. The store-and-forward scheme regularizes the irregular communication pattern, thereby reducing latency. However, this approach increases communication volume due to the overhead introduced by the store-and-forward mechanism. Moreover, as the dimensionality of the VPT increases, the resulting communication volume grows significantly, which limits the scalability of the HPC system and can become a bottleneck. This communication volume can be reduced by repositioning heavily communicating task pairs closer to each other on the VPT in terms of topological distance. To this end, a Kernighan-Lin-based [3] heuristic is proposed to refine the one-to-one task-to-processor mapping produced by the initial partitioning onto the VPT, with the goal of further reducing communication overhead.

The remainder of this thesis is organized as follows: Chapter 2 provides background information on the core concepts and builds up to the significance of this work. Chapter 3 reviews prior and related work on partitioning, the quadratic assignment problem, constructive and iterative heuristics on task-to-processor problems, leading to the motivation for this thesis. Chapter 4 presents the proposed heuristic, formulates and analyzes it, and details an improved algorithm. Each chapter includes specific preliminaries. Section 4.4 extends and modifies the notation and terminology to better explain the improved algorithm. Chapter 5 provides the experimental setup, results, and evaluation. Finally, Chapter 6 concludes the thesis and provides recommendations for future work.

Chapter 2

Preliminaries

Scientific and engineering simulations, artificial intelligence (AI), and big data applications often require matrix-matrix or matrix-vector multiplications, particularly when performing with linear algebra operations. These matrices and vectors may have varying specifications and sizes. Therefore, the development of efficient computation strategies becomes both a research question and an engineering challenge for modern computational hardware. The computational cost of matrix multiplication can be expressed as follows: In the naive multiplication of two $n \times n$ matrices, n^3 multiplication and $(n^3 - n^2)$ addition operations are required. For example, when $n = 10^6$, a total of 10^{18} multiplication operations are executed. Storing the two input matrices in memory requires $2 \cdot 10^{12}$ matrix entries. Considering each entry has 32-bit precision, the storage requirement for a single matrix of 10^{12} entries is 4 terabytes. An increase of one unit in n results in a 1000-fold increase in the number of multiplications and additions, while the required storage grows by a factor of 100. In practice, the matrix dimension n ranges from 10^3 to 10^9 . Consequently, computational capability with the capacity to handle such operations is required for applications ranging from fundamental engineering and scientific computations to more complex tasks such as climate modeling and large-scale artificial intelligence workloads. In scenarios with high computational load, the required matrix multiplication task must be completed within a specified time frame. For example, in a simulation conducted for a

weather forecast one week ahead, the desired outcome is for the computation to be completed before the one-week mark.

The computation task can be performed by a single computational unit executing arithmetic and logic operations (referred to as a single-core CPU in this context), completing the task within a certain time frame. However, when the task is distributed across multiple processors in parallel, it can be completed faster. This approach provides a significant advantage, particularly in the execution of large-scale and complex computations. Task partitioning, distribution, and execution in accordance with the nature of the problem fall within the scope of High Performance Computing (HPC).

2.1 Dense Matrix Partitioning

Consider the dense matrix and vector multiplication, represented as $A\mathbf{x} = \mathbf{y}$, where A denotes the matrix, and $\mathbf{x}$ and $\mathbf{y}$ denote vectors. Bold symbols are used to represent vectors. A dense matrix is a matrix in which the majority of the elements are nonzero. The computation process required for the multiplication can be divided into four equal parts, with each part assigned to a different processor. The dense matrix-vector multiplication $A\mathbf{x} = \mathbf{y}$ is illustrated using the following coloring scheme:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ a_{31} & a_{32} & \cdots & a_{3n} \\ a_{41} & a_{42} & \cdots & a_{4n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} \quad (2.1)$$

In the case of the $A\mathbf{x}$ multiplication, the work (i.e., task) is distributed to processors row-wise on matrix A , where each partition corresponds to a section of the matrix that is color-coded for easier identification. The multiplication of matrix A and vector $\mathbf{x}$ is performed as follows:

$$\Pi_1 = [a_{11}, a_{12}, \dots, a_{1n}] \quad (2.2)$$

$$\Pi_1 \mathbf{x} = a_{11} \cdot x_1 + a_{12} \cdot x_2 + \cdots + a_{1n} \cdot x_n = y_1 \quad (2.3)$$

Each processor performs the multiplication for one part of the matrix. Here, the Greek letter Π is used to denote the first letter of the word 'partition'. Π_1 corresponds to the first row of matrix A , which is a $1 \times n$ vector. For larger matrices, a partition may correspond to a submatrix containing multiple rows. The first row of matrix A , i.e., the first partition, does not affect the results y_2 , y_3 , or y_4 . Therefore, the matrix multiplication operation can be divided and assigned to different processors (i.e., computation units, processing units, or nodes) to be computed concurrently. However, the entire vector $\mathbf{x}$ is sent to all processors for the multiplication operation. Throughout the complete computation, the matrix–vector multiplication task is divided into four equal parts, distributed to four different processors, and the multiplication outputs are combined to obtain the final result. For ease of explanation, each processor can be considered an independent computing node, with all nodes in the system operating concurrently and communicating with one another.

$$\Pi = \begin{bmatrix} \Pi_1 \\ \Pi_2 \\ \Pi_3 \\ \Pi_4 \end{bmatrix}, \quad \mathcal{P} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix} \quad (2.4)$$

In the preliminary, the set of partitions are denoted by $\Pi = \{\Pi_1, \Pi_2, \dots, \Pi_K\}$, where K is the number of processors. The partitioned tasks are then assigned to processors, with a one-to-one mapping between partitions and processors. The set of processors to which partitions are assigned is denoted by $\mathcal{P} = \{p_1, p_2, \dots, p_K\}$. Assignment can be performed sequentially, randomly, or using mapping strategies. Smart mapping methods are discussed in the Section 3. It should be noted that the number of partitions does not need to match the number of processors.

In the context of HPC, a task refers to a specific portion of work assigned to a processor. A task may contain not only computations associated with a single partition of a matrix multiplication but also additional operations, such as sending and receiving data or aggregating results. In this thesis, the workloads assigned

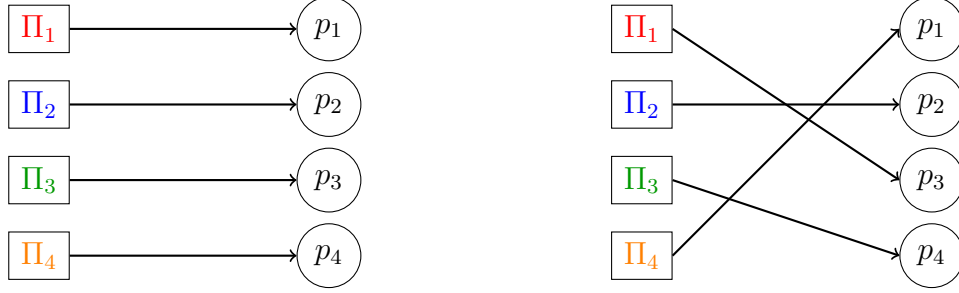


Figure 2.1: Sequential Assignment (left), Random Assignment (right)

to processors are referred to as tasks. In the example of dense matrix–vector multiplication, a partition is used to divide the matrix multiplication operation into sections and may correspond to a part or the entirety of the work executed by a processor. In contrast, a task contains all operations to be performed on a partition. Therefore, a partition can be a subset of a task, but a task does not necessarily correspond exactly to a single partition. In other words, tasks represent the actual workload executed by processors, including both computational and auxiliary operations (e.g., communication, reduction operations). Task-to-processor assignment strategies can be categorized as one-to-one, one-to-many, or many-to-one mappings. In a one-to-many mapping, a single partition is distributed across multiple processors, whereas in a many-to-one mapping, multiple partitions are assigned to a single processor. When each partition is assigned to exactly one processor, the mapping is referred to as one-to-one. In this context, partitions represent the computational tasks derived from the matrix multiplication. To illustrate the definition of a task, the assignment of multiple partitions to a single processor in a many-to-one mapping can be considered. Alternatively, when exactly one partition is assigned to each processor and the results of the multiplications of other partitions are accumulated by the same processor, the task can be regarded as comprising both the partition and the corresponding reduction operation.

In the remainder of the thesis, the task set $\mathcal{T}$ will be denoted as $\mathcal{T} = \{t_1, t_2, \dots, t_K\}$, where K corresponds to the number of partitions in the one-to-one mapping method. The processor to which a task t_i is assigned will be denoted as $map(t_i)$, while the task assigned to a processor p_i will be denoted as

$imap(p_i)$, where *imap* is an abbreviation for “inverse map.”

2.2 Sparse Matrix Partitioning

Another important matrix type is the sparse matrix, in which the majority of elements are zero. Sparse matrices commonly arise in real-world applications such as graph theory, scientific computing, and artificial intelligence. During matrix-matrix or matrix-vector multiplication, any multiplication involving zero-valued elements yields a result of zero. Consequently, for sparse matrices of size $n \times n$, where n may reach magnitudes such as 10^6 or even 10^9 , performing operations as if the matrix were dense results in significant inefficiencies in both memory usage and computational cost.

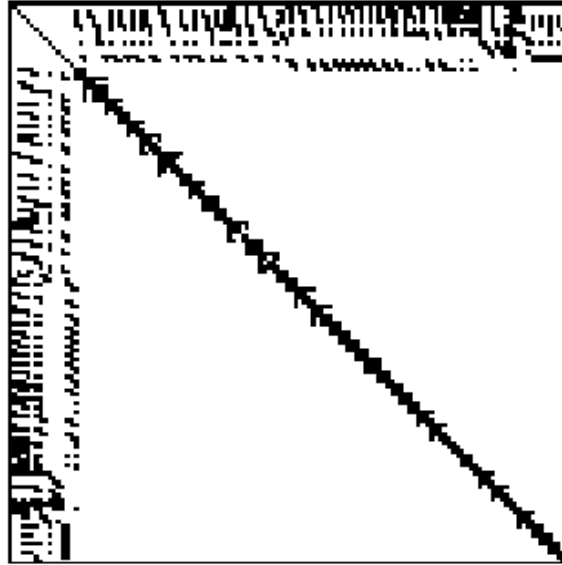


Figure 2.2: 2-dimensional visualization of the sparse matrix `mip1` [1].

The sparse matrix-vector multiplication example is given in Equation 2.5. Each partition of the matrix, which in this example corresponds to a row, is assigned to a different processor. The contribution of each nonzero entry to the

final result vector is highlighted by color. When the multiplication is examined, it can be observed that zero-valued elements do not contribute to the result. In this case, excluding zero-valued elements from the multiplication can provide significant computational gains.

In contrast to the dense matrix–vector multiplication presented in Equation 2.1, the vector $\mathbf{x}$ is also partitioned in the sparse matrix–vector multiplication. In sparse matrices, since many elements are zero, the entire vector $\mathbf{x}$ is not required for the multiplication. Therefore, when distributing vector to processors, it is sufficient to send only the portions of $\mathbf{x}$ (excluding zeros) needed for the local computations. This strategy of sending only the nonzero entries of $\mathbf{x}$ that are required for the computations of each partition becomes critical for efficiency in Sparse Matrix-Vector (SpMV) and Sparse Matrix-Matrix (SpMM) multiplications, where matrix dimensions may reach the order of 10^8 .

$$\begin{bmatrix} 0 & a_{12} & 0 & 0 \\ 0 & 0 & a_{23} & 0 \\ a_{31} & 0 & 0 & 0 \\ 0 & a_{42} & 0 & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} a_{12} \cdot x_2 \\ a_{23} \cdot x_3 \\ a_{31} \cdot x_1 \\ a_{42} \cdot x_2 + a_{44} \cdot x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} \quad (2.5)$$

2.2.1 Sparse Matrix Storage Formats

A dense $10^6 \times 10^6$ matrix with 32-bit precision per entry requires 4 terabytes of storage. In sparse matrices, if zero elements are not stored, the storage size can be reduced to kilobytes (KB), megabytes (MB), or gigabytes (GB), depending on the matrix size and sparsity. To optimize storage and computation, various sparse matrix storage formats are used. The most common are the Coordinate List (COO), Compressed Sparse Row (CSR), and Compressed Sparse Column (CSC) formats, each utilizing different data structures and architectures specialized for the requirements of problems. The choice between CSR and CSC depends on the access pattern and the memory organization of the target hardware.

The *Coordinate List (COO)* format stores nonzero elements of a sparse matrix as triplets consisting of its row index, column index, and value: $\langle \text{row}, \text{column}, \text{value} \rangle$. These coordinates can be stored for efficient access in three separate arrays for row indices, col indices and values, if sparsity is low. COO is efficient for constructing sparse matrices or reading data streams, as well as for direct access when access patterns are irregular. However, it is not as efficient for matrix operations such as SpMV due to its lack of data locality.

The *Compressed Sparse Row (CSR)* format represents a sparse matrix using three one-dimensional arrays. CSR is optimized for row-based operations, such as row-wise partitioning or matrix-vector multiplications, especially in row-major memory layouts. The *Values* array stores the non-zero elements of the matrix in row-major order. The *ColumnIndices* array stores the column indices of these nonzero elements, aligned with the *Values* array in row-major order. In other words, for each nonzero element stored in *Values*, its corresponding column index in the original matrix is stored at the same position in the *ColumnIndices* array. What remains is to track where the nonzero element of each row begin within the *Values* array. This is achieved using the *RowPointer* array. The *RowPointer* array stores the starting index of the nonzero element of each row, in row-major order. The first element of the *RowPointer* array is always equal to 1, and the last element of the *RowPointer* array is always equal to the total number of nonzero elements in the matrix plus one using 1-based indexing. The CSR representation (using 1-based indexing) of the sparse matrix A from Equation 2.5 is as follows :

$$\begin{aligned} \text{Values} &= [a_{12}, a_{23}, a_{31}, a_{42}, a_{44}] \\ \text{ColumnIndices} &= [1, 2, 0, 1, 3] \\ \text{RowPointer} &= [1, 2, 3, 4, 6] \end{aligned} \tag{2.6}$$

To access the nonzero elements of row i , $a = \text{RowPointer}[i]$ points to the starting index of i -th row's elements, and $b = \text{RowPointer}[i + 1]$ points to the starting index of $(i + 1)$ -th row's elements in the *Values* and *ColumnIndices* arrays. The nonzero values for i -th row are then found in the closed interval $\text{Values}[a : b - 1]$, and their corresponding column indices are in interval $\text{ColumnIndices}[a : b - 1]$ where indices a and $(b - 1)$ are included. For example, to access the nonzero elements of the second row ($i = 2$) of the sparse matrix given in Equation 2.6 in

CSR format: $a = \text{RowPointer}[2]$ which is 2, $b = \text{RowPointer}[2 + 1]$ which is 3. The nonzero values for second row are located in $\text{Values}[2 : 2]$, and their column indices are in $\text{ColumnIndices}[2 : 2]$. As a result, only one nonzero element exists in second row, which is a_{23} at column index 2 of the sparse matrix.

The *Compressed Sparse Column (CSC)* format is the column-major version of CSR. CSC is optimized for column-based operations, such as column-wise partitioning, especially in column-major memory layouts. It uses three arrays: *Values* stores nonzero elements in column-major order, *RowIndices* stores their corresponding row indices, and *ColumnPointers* stores the starting index of the nonzero element of each column in column-major order.

2.2.2 Communication Matrix

For optimal performance, it is preferable that all data required for a computation be stored on the same processor. If this is not the case, a processor must receive the necessary data from another processor, generating inter-processor communication traffic. The following analysis structures the communication patterns of the sparse matrix example to model processor-to-processor data exchanges.

Equation 2.5 illustrates a naive partitioning in which no objective function (i.e., cost function) is applied. The first partition, shown in red and assigned to p_1 , needs the value x_2 from the second partition, shown in blue and assigned to p_2 , to perform its multiplication. Table 2.1 demonstrates the send (rows) and receive (columns) operations for all partitions of the example given in Equation 2.5. Only p_4 does not send any data, as the other partitions do not require the value x_4 . The vector $\mathbf{x}$ consists of elements x_a , each representing a message with a unit size of 1 word. The subscript a is used here temporarily to denote an arbitrary x value. For illustrative purposes, the sparse matrix and vector are divided into four equal partitions. Consequently, each partition sends a single x value; however, for larger problem sizes, a partition may need to send multiple x values. The word size may vary depending on the underlying computer architecture. On a 64-bit architecture, one word corresponds to 64 bits. In the context of memory access

and data transfer operations, a word is considered the unit size. Throughout the remainder of this thesis, the size of each element of $\mathbf{x}$ is assumed to be 1 word.

Table 2.1: Processor communication pattern after row-wise partitioning

Sender (p_i)	Receiver (p_j)	Data
p_1	p_3	x_1
p_2	p_1	x_2
p_2	p_4	x_2
p_3	p_2	x_3
p_4	-	-

Assuming that the data size is 1 word, the Communication Matrix (M_c) is constructed as shown in Figure 2.3. In this representation, rows correspond to senders and columns correspond to receivers. The row index is denoted by i and the column index by j . A message m_{ij} represents the data transferred from processor p_i to processor p_j , containing the required x values. For example, if p_2 sends data to p_1 , the entry $M_c(2, 1)$ in the communication matrix is incremented by 1. This indicates that, $M_c(2, 1) = 1$ represents that p_2 transmits 1 word of data to p_1 , while p_1 correspondingly receives 1 word from p_2 . More generally, if processor p_2 transmits c number of distinct x_a elements to p_1 , then the corresponding matrix entry becomes $M_c(2, 1) = c \cdot 1$, as each message has a unit size of 1 word.

	p_1	p_2	p_3	p_4
p_1	0	0	1	0
p_2	1	0	0	1
p_3	0	1	0	0
p_4	0	0	0	0

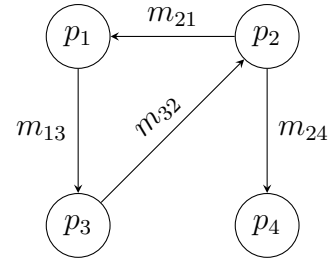


Figure 2.3: Processor-to-Processor Communication Matrix (left), and its graph representation (right) where processors are represented as vertices and communication paths as edges.

As previously stated, a task is defined as a portion of workload assigned to a processor through the mapping method. The designation as tasks reflects that

partitioning has been performed and that the subsequent explanation proceeds in terms of tasks. In the methodology section, the task-to-task communication matrix (M_c) is used instead of the processor-to-processor communication matrix shown in Figure 2.3. This distinction does not indicate a conceptual difference: the matrix M_c is constructed in the same way as in the example in Figure 2.3, by counting the elements of the vector $\mathbf{x}$ assigned to the partitions, but expressed in terms of tasks rather than processors. This designation enables further methods to be explained, as the tasks assigned to processors may change even though processor positions remain fixed. Consequently, the explanation of inter-task communication is expressed in terms of task-to-task (also referred to as process-to-process or inter-process communication) communication, reflecting the interactions between computational units rather than physical processors.

2.2.3 Partitioning Strategies

Figure 2.4 (left) presents the same partitioning as in Equation 2.5 and Figure 2.3, demonstrated in a different visual representation. This representation provides a clearer illustration of the partitioning methods. The matrix from Equation 2.5 is partitioned for visualization: on the left, it is partitioned row-wise, following the original partitioning, while on the right, the same matrix is partitioned column-wise to illustrate an alternative partitioning approach. The matrix elements are enclosed in black boxes solely for visual clarity. Unlike the original representation, the elements of $\mathbf{x}$ are written above their corresponding columns, since, during matrix-vector multiplication, each value is multiplied only with the elements of the corresponding column. For example, x_1 is multiplied with $a_{11}, a_{21}, a_{31}, a_{41}$. For visual simplicity, the elements of $\mathbf{x}$ are not color-coded according to their partitions in this figure; however, as in Equation 2.5, the values x_1, x_2, x_3, x_4 can be colored *red, blue, green, orange* for both row-wise and column-wise partitioning. The y_i partitions are assigned to processors p_i along with their corresponding partition Π_i , indicating that they will be computed by these processors.

In row-wise partitioning, each processor computes its local portion of $\mathbf{y}$, denoted as y_i , by multiplying the corresponding partition Π_i with the elements of the vector $\mathbf{x}$ required by multiple partition (i.e., the nonzero elements corresponding to Π_i). For example, the value x_2 is needed both in the computation $x_2 \cdot a_{12}$ for Π_1 and in $x_2 \cdot a_{42}$ for Π_4 . Since the vector elements are distributed prior to the multiplication, no additional data communication is required during or after the computation. This situation differs in the case of column-wise partitioning.

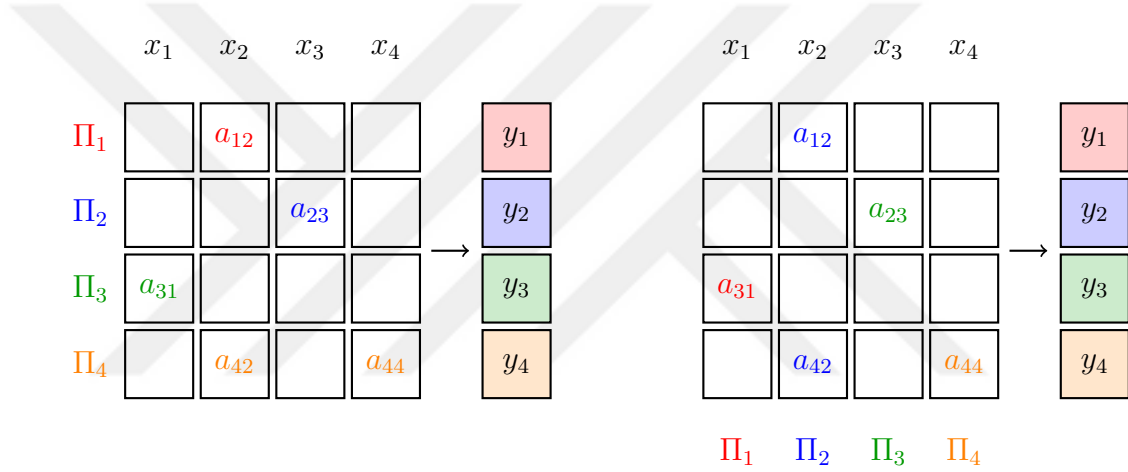


Figure 2.4: Row-wise Partitioning of the Sparse Matrix (left), Column-wise Partitioning of the Sparse Matrix (right).

In the right subfigure of Figure 2.4, the matrix A is partitioned column-wise, dividing it into columns 1, 2, 3, and 4. The matrix elements of each partition are recolored to reflect the column-wise partitioning. Partition Π_1 and x_1 are assigned to processor p_1 , Π_2 and x_2 to p_2 , Π_3 and x_3 to p_3 , and Π_4 and x_4 to p_4 . In this partitioning scheme, each partition already contains all the vector elements required for its local multiplication. Therefore, there is no need to send the vector elements prior to the multiplication. As a result, the total communication for row-wise partitioning, previously 4 in Figure 2.3, is reduced to 0. However, unlike row-wise partitioning, the partial results of the multiplication must be sent to the corresponding processors for calculating the final y_i values. For example, in the column-wise partitioning example $y_4 = a_{42} \cdot x_2 + a_{44} \cdot x_4$. The computation $a_{42} \cdot x_2$ occurs in partition Π_2 on processor p_2 , while $a_{44} \cdot x_4$ occurs in partition

Π_4 on processor p_4 . Therefore, to compute y_4 , the result of $a_{42} \cdot x_2$ calculated on processor p_2 must be sent from processor p_2 to p_4 , where it is added to $a_{44} \cdot x_4$ to obtain the final y_4 . When the same procedure is applied to all partitions, a total of four multiplication results are sent to their respective processors in column-wise partitioning.

As a result, row-wise partitioning requires inter-processor communication before computation, and then the multiplication is completed locally, on the processors. Column-wise partitioning eliminates the need for inter-processor communication prior to computation. However, after the processors complete their local computations, inter-processor communication is required to aggregate the results and finalize the multiplication. Within the scope of this example, neither method provides a clear advantage over the other. Nevertheless, depending on the characteristics of the matrices used in matrix multiplications, a partitioning method may be more suitable than the other.

2.2.4 Load balancing

Inter-processor communication can introduce network latency and bandwidth limitations. To reduce such communication, priority can be given to placing the data required for a multiplication within the same partition. However, reducing communication may lead to an uneven distribution of the computational load on processors. A balanced distribution of computation aims that all processors begin their calculations simultaneously and finish at approximately the same time. Achieving such balance is a primary objective in High Performance Computing, as it minimizes idle resources and enables computations to complete in the shortest possible time. In modern HPC systems, processors have advanced technologically faster than the network communication between processors. Consequently, the latency introduced by communication, including network links, network switches, routing has become a dominant factor contributing to idle time in large-scale computations. Moreover, many HPC applications are memory-bound, meaning that

arithmetic operations are relatively inexpensive while data fetching from memory is slow and can create a performance bottleneck [4]. Compared to arithmetic operations, data movement such as memory operations, communication, and I/O often represent the most energy-consuming component in HPC systems [5, 6].

In Figure 2.5, a sample matrix is partitioned using both row-wise and column-wise approaches. The communication and computation resulting from these two partitioning methods are examined and analyzed. Operations such as multiplication and addition that occur within a processor are considered as computation, whereas communication refers to the data exchange occurring between processors. In the example, each partition is assigned to a processor as demonstrated in Section 2.2.3. The computation corresponding to the workload of the i -th processor is denoted by $Comp(i)$, and the communication is denoted by $Comm(i)$. $Comm(i)$ represents the number of words received by the i -th processor.

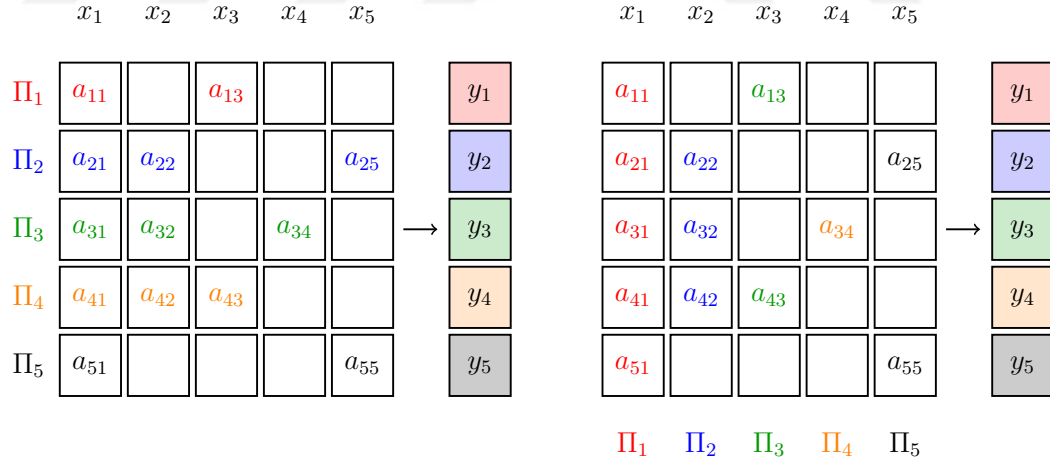


Figure 2.5: Row-wise Partitioning of the Sparse Matrix (left), Column-wise Partitioning of the Sparse Matrix (right).

The computation cost of a partition is defined here based on the number of nonzero (nnz) elements it contains. In other words, the cost of each multiplication operation in the partition is set to 1. Addition operations are neglected for simplicity. In the example presented in Figure 2.5, the row-wise partition Π_1 involves 2 multiplication operations and $(2 - 1)$ addition operations. Therefore,

the computation cost for p_1 is defined as $Comp(1) = 2$. The communication cost is considered here to correspond to the amount of inter-processor data movement, measured in words. Recall that each nonzero element or message has a size of one word. In this case, processor p_1 will receive a total of one element (x_3) from the other partitions. Therefore, the receiving communication cost for p_1 is defined as $Comm_{recv}(1) = 1$. Additionally, p_1 will send the value x_1 to four partitions. Therefore, $Comm_{send}(1) = 4$. As a reminder, each partition is assigned to the processor with the same index (e.g., Π_1 assigned to p_1).

For the row-wise and col-wise partitioning, all computation and communication costs are provided as vectors. Each element corresponds to a processor index: the first element corresponds to p_1 , the second element corresponds to p_2 , and so forth.

$$\begin{aligned} Comp_{row-wise} &= [2, 3, 3, 3, 2] \\ Comm_{recv-row-wise} &= [1, 2, 3, 3, 1] \\ Comm_{send-row-wise} &= [4, 2, 2, 1, 1] \end{aligned} \tag{2.7}$$

In the column-wise partition example, the partitions will send the results of their local multiplications to the corresponding partitions. For instance, Processor p_1 , will send the values $a_{21} \cdot x_1$, $a_{31} \cdot x_1$, $a_{41} \cdot x_1$, and $a_{51} \cdot x_1$. Therefore, the sending communication cost for p_1 is defined as $Comm_{send}(1) = 4$.

$$\begin{aligned} Comp_{col-wise} &= [5, 3, 2, 1, 2] \\ Comm_{recv-col-wise} &= [1, 2, 3, 3, 1] \\ Comm_{send-col-wise} &= [4, 2, 2, 1, 1] \end{aligned} \tag{2.8}$$

In the given example, row-wise partitioning results in a more balanced computation load compared to column-wise partitioning; that is, the workloads are distributed as evenly as possible. The small example matrix is provided to demonstrate the concept; however, for larger matrices involving thousands or millions of operations, significant differences in computation completion times across processors can occur. Message exchanges between processors can cause communication delays depending on the computational environment. Consequently, an effective partitioning strategy must be determined for sparse matrix-vector multiplication, considering the available hardware, software environment, and computational implementation details.

Changing the number of partitions and processors affects the load balance. Dividing the matrix multiplication into fewer partitions can be advantageous. For example, consider the example matrix partitioned column-wise into three partitions and assigned to three processors. For the case where the matrix is partitioned as $\Pi_1, (\Pi_2, \Pi_3), (\Pi_4, \Pi_5)$, the computation and communication costs are as follows:

$$\begin{aligned} Comp_{col-wise} &= [5, 5, 3] \\ Comm_{recv-col-wise} &= [1, 3, 4] \\ Comm_{send-col-wise} &= [4, 2, 3] \end{aligned} \tag{2.9}$$

For the case where the matrix is partitioned as $\Pi_1, (\Pi_2, \Pi_4), (\Pi_3, \Pi_5)$, the computation and communication costs are as follows:

$$\begin{aligned} Comp_{col-wise} &= [5, 4, 4] \\ Comm_{recv-col-wise} &= [1, 4, 4] \\ Comm_{send-col-wise} &= [4, 3, 2] \end{aligned} \tag{2.10}$$

By examining Equations 2.7–2.10, it can be observed that in a one-to-one mapping, as the number of partitions decreases, the computational load assigned to each processor increases. In this example, it can be observed that the total number of messages decreases if a communication graph is constructed or if the values of the $Comm_{send}$ vector are summed. However, the total size of the messages sent by each processor increases. Depending on the partitioning strategy applied to the problem, the number of messages generally decreases due to the reduction in the total number of connections, while the communication volume, that is the total amount of data exchanged across all processors, increases. For instance, consider a 1000×1000 matrix being partitioned. In this partitioning, each row or each nonzero element can be assigned to a processor. In this case, the computation on each processor finishes quickly; however, communication becomes excessive due to the large number of messages. This situation complicates the synchronization of computation and communication. Such a partitioning method is referred to as fine-grain. In another partitioning method, multiple rows can be assigned to each processor. In this case, inter-processor communication decreases,

while the computational load per processor increases. However, achieving a balanced distribution of workloads becomes more challenging, and the achievable level of parallelism decreases. This type of partitioning is referred to as coarse-grain. During partitioning, the balance between computation and communication must be carefully managed by considering both the target hardware architecture and the characteristics of the problem. Furthermore, various metrics can be used to model computational and communication costs.

2.2.5 Symmetric and Nonsymmetric Partitioning

In SpMV multiplication $A\mathbf{x} = \mathbf{y}$ is computed. This computation is distributed across processors as explained in Section X. The partitioning strategy determines which processor stores which parts of the matrix A , the input vector $\mathbf{x}$, and the output vector $\mathbf{y}$. In the symmetric partitioning, the matrix, the input, and the output vectors are partitioned in a matching way. For example, consider a 100×100 SpMV multiplication partitioned across two processors. Processor p_1 stores rows of A , denoted by $A[1..50]$, and computes the corresponding entries $\mathbf{y}[1..50]$; processor p_2 stores rows of A , denoted by $A[51..100]$, and computes $\mathbf{y}[51..100]$. In this symmetric partitioning example, p_1 stores the corresponding input vector entries which are $\mathbf{x}[1..50]$, and p_2 stores $\mathbf{x}[51..100]$. However, in the symmetric partition case, each processor may need nonlocal entries of $\mathbf{x}$ from other processors to compute its part of $\mathbf{y}$. For example, matrix A partitioned symmetrically across two processors. p_1 stores $A[1..50]$ and requires the 3rd entry of $\mathbf{x}$, and the entries from the 70th to the 90th indices which are $\mathbf{x}[70..90]$ to compute $\mathbf{y}[1..50]$. In this case, storing the entire range $\mathbf{x}[1..50]$ on p_1 is inefficient, as many of those values will not be used. Placing the required $\mathbf{x}$ entries on another processor may increase communication volume, especially if other processors do not require those entries. Therefore, nonsymmetric partitioning can potentially improve communication.

In nonsymmetric partitioning, the entries of the input vector $\mathbf{x}$ can be assigned to processors independently of the matrix A partitioning, in a way that

minimizes data transfer. The goal is to assign $\mathbf{x}$ entries based on data dependencies. For the nonsymmetric partitioning case, p_1 stores $A[1..50]$, computes $\mathbf{y}[1..50]$; differing from the symmetric case, p_1 may store the corresponding entries $\mathbf{x}[3]$ and $\mathbf{x}[70..90]$. The possibility expressed by 'may' indicates that, if a $\mathbf{x}$ entry is required by multiple processors, it can be randomly assigned to a processor or copied to processors that require the entry. However, the need to keep track of which processor owns which parts of $\mathbf{x}$ and where to fetch the nonlocal entries may introduce computational overhead. Another advantage of nonsymmetric partitioning is that it enables communication/computation overlap. While waiting for remote $\mathbf{x}$ entries to arrive, a processor can begin computing parts of $\mathbf{y}$ that only depend on local $\mathbf{x}$ entries, improving overall parallel performance. Nonsymmetric partitioning can be more efficient than symmetric partitioning for irregular sparsity patterns [7, p. 21], optimizing metrics such as latency, bandwidth, communication volume, load balance, and overall execution time.

2.3 Topology and Message passing

In the field of HPC, the term topology is used to describe the interconnections among computational units. A computational unit refers to a core, processor, or node that performs computations in an HPC system. A node is a single computing unit that contains one or more processors (CPU), graphics processing unit (GPU), memory (RAM), storage, and network interfaces. In Figure 2.3, the representation on the right illustrates the messages exchanged between processors. The transmission of these messages can occur either through direct connections (links) between the computation units or along a designated route across intermediate connections. Figure 2.6 presents the Full Mesh and Star topologies for a system with six nodes ($K = 6$). These two topologies are presented in contrast due to their distinct structural characteristics. In network topology, a direct connection between two nodes is referred to as a point-to-point (P2P) connection. In practice, this may correspond to either a physical cable link between devices or a virtual link established over the underlying network infrastructure. In a Star topology, the total number of point-to-point (P2P) connections is K , where K

represents the total number of nodes. The central node directs communication among all peripheral nodes and may be either a high-speed network switch or a computational unit. All nodes communicate with each other through this central node. In a Full Mesh topology, each node is directly connected to every other node. This topology has the highest number of connections. Each node has a total of $(K - 1)$ P2P connections, and the total number of P2P connections in a Full Mesh topology is given by $\frac{K \cdot (K-1)}{2}$. In a Star topology, a message sent from one node to another traverses 2 hops, whereas in a Full Mesh topology it traverses only 1 hop. The hop count refers to the number of P2P connections that a data packet goes through to reach its destination. For example, a message sent from p_5 to p_1 follows the route p_5 –switch– p_1 in a Star topology, while in a Full Mesh topology it follows the direct route p_5 – p_1 .

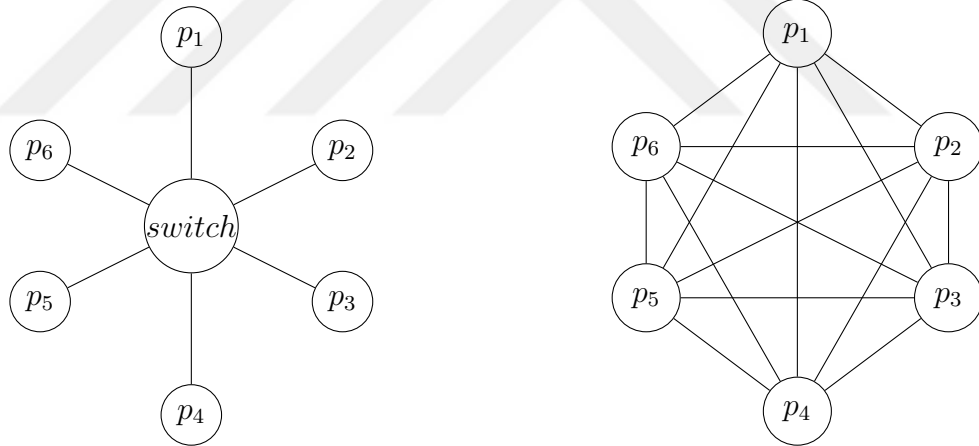


Figure 2.6: Star Topology (left), Full Mesh Topology (right).

To compare the effects of topology on communication volume and the number of messages sent, consider processor p_5 sending a 1-word message to processors $p_1, p_2, p_3,$ and p_6 . These messages are denoted as $m_{51}, m_{52}, m_{53},$ and m_{56} . In a Star topology, processor p_5 must send all four messages to the switch. The switch forwards these messages to the corresponding processors. Therefore, p_5 sends 4 messages to the switch, and the switch forwards 4 messages to the target processors. Synchronization at the switch is not considered in this example, and sufficient bandwidth is assumed to be available. After the communication completes, the total number of messages is 8, and the total communication volume

is 8 words. In a Star topology, all communication between nodes occurs via the switch. This can cause delays when the number of messages is high, or a bottleneck when the communication volume exceeds the available bandwidth. In a Full Mesh topology, every node is directly connected to all others. Processor p_5 can send the four messages directly to the target processors. After the communication completes, the total number of messages is 4 and total communication volume is 4 words. Bandwidth is defined as the amount of data that can be transmitted per unit of time. Communication volume refers to the total amount of data (e.g., 8 words in a star topology or 4 words in a full mesh topology) transferred between processors during message exchanges.

As observed, changing the topology affects both the number of messages and the communication volume. If every processor sends messages to all other processors, a total of $2 \cdot K \cdot (K - 1)$ messages are required in a Star topology, whereas $K \cdot (K - 1)$ messages are required in a Full Mesh topology. In the Star topology, all messages are collected at the central switch and then forwarded, while in the Full Mesh topology, messages are transmitted directly between nodes. In the case of a failure, communication cannot be routed in the Star topology, whereas in the Full Mesh topology, the greatest possible number of alternative communication paths is available. Therefore, a Full Mesh topology provides the best result for HPC systems in terms of communication performance and fault tolerance. However, implementing a Full Mesh topology requires $K \cdot (K - 1)/2$ physical connections in total and $(K - 1)$ ports per node. For a system with $K = 1024$ nodes, this corresponds to 523776 connections. The physical construction of such a system is extremely costly and presents a major engineering challenge. Moreover, since the number of required connections grows quadratically with K , the topology does not scale efficiently.

In many HPC applications, it is not necessary for all nodes to communicate directly with each other. Messages can be efficiently routed through algorithms, and tasks with high communication intensity can be positioned close to one another. Therefore, alternative topologies are used by optimizing the number of connections. This approach enables the scalability of HPC systems and improves efficiency. As an example, a two-dimensional mesh topology of size 4×4 for a

system with 16 nodes is illustrated in Figure 2.7.

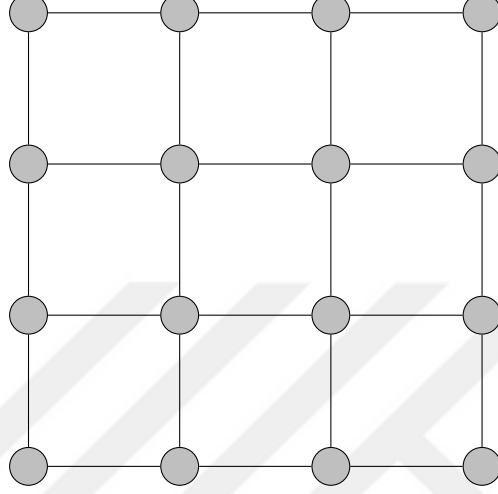


Figure 2.7: 4×4 Mesh Topology

When every node sends a message to all other nodes, a total of $K \cdot (K - 1)$ messages are required. In a 2D grid mesh topology, the total number of physical connections is $2(K - \sqrt{K})$ [8, Table 2.1]. While the number of connections grows asymptotically as $O(K^2)$ for the full mesh, it is only $O(K)$ for the 2D grid mesh. However, this reduction in physical connections comes at the cost of increased message transmission times and higher traffic. The reason is that, in a 2D grid mesh, messages with a destination of non-neighboring nodes must be forwarded through intermediate nodes. If the traffic on a particular point-to-point connection becomes high, it can be redirected to alternative links. Furthermore, while the maximum communication distance in the full mesh is one hop, it increases to $2(\sqrt{K} - 1)$ hops in the 2D grid mesh. The maximum distance between two nodes is referred to as the diameter of a network [8]. The increase in the average hop distance in the 2D grid mesh leads to an increase in the total communication volume across the system. In other words, when messages traverse multiple intermediate nodes, the total amount of data transmitted throughout the network, particularly over certain heavily utilized connections (hotspots), increases. Excessive demand on system resources, especially on hotspots, can lead to network congestion, which occurs when multiple processes attempt to access

the network simultaneously. For instance, in a star topology, the presence of a single central node creates a bottleneck, potentially causing congestion. This congestion increases latency, communication overhead, and reduces scalability and performance in parallel applications. The increase in communication volume also elevates the required bandwidth. Techniques such as network topology design, load balancing, and routing algorithms are employed to reduce or prevent congestion, and enhance the efficiency.

2.4 Communication Overhead

The intensive communication requirements during HPC tasks can lead to various bottlenecks, negatively affecting overall system efficiency. The additional delays and performance losses caused by communication are defined as communication overhead. Overhead represents the additional time and resources an HPC system requires to complete a task. The most common causes of communication overhead are latency and bandwidth costs.

Latency cost: Latency is the time a message takes to travel from its source to its destination. This includes the startup time (t_s), which is the total time required to determine the message route, package the message, establish a connection with the destination, and process the message at the receiving end [8]. The transmission of the message between the source and the destination also contributes to this delay. In problems involving highly intensive communication or requiring fast computation, such latency is undesirable. For example, in fine-grained parallelism, the message size decreases while the number of messages increases. As the number of messages increases, the total communication time increases, potentially leading to greater overhead.

Bandwidth cost: Bandwidth is the amount of data transmitted over a network connection (volume) per unit of time. This theoretical value, which defines the communication capacity of the network, is expressed as the product of the number of messages transmitted per second and the size of each message. The actual

value measured during execution is referred to as throughput. For example, consider a point-to-point (P2P) connection with a bandwidth of 100 GB per second. If the data communication (throughput) during parallel computation approaches or exceeds this value, the P2P connections become saturated, leading to queue formation and consequently delays. This situation increases the overall communication overhead in the system.

In the field of HPC, latency costs are generally more critical than bandwidth limitations [7, p. 3]. Advancements in HPC hardware, network topologies, and network infrastructure have significantly improved both latency and bandwidth. Innovations such as advanced routing algorithms, optimized topologies, and modern network switches have enabled more pronounced improvements particularly in bandwidth and congestion management. Furthermore, the partitioning process should also be carried out by considering these costs, aligning with the characteristics of the hardware used.

2.5 Optimization Strategies for Irregular HPC Workloads

In HPC systems, latency-bound applications/workloads with irregular communication patterns such as those in sparse matrix multiplication cause uneven message traffic that can congest the physical network. For example, consider a sparse matrix-vector multiplication operation partitioned and distributed across nodes in an HPC system. Some tasks may send around 30 messages, while others send more than 2,000 messages. Such imbalanced communication can cause irregular communication patterns when tasks are mapped onto the physical topology. This situation may lead to congestion on certain network links and increase the communication overhead. Consequently, congestion increases communication delays, negatively impacting overall system performance and scalability.

Various methods are employed to prevent or reduce imbalanced communication and congestion. The physical network topology in HPC systems monitors communication at the hardware level and dynamically routes messages to mitigate network congestion. While the physical topology manages data transmission paths, it does not have knowledge of application-level communication. In other words, optimization is performed at the hardware level, without awareness of the communication pattern of the application or the computational workload of tasks. However, the communication pattern can be analyzed and modeled through algorithms, enabling message traffic to be predicted and improved. In HPC systems, while an application is running, run-time profiling collects information on message traffic between nodes, communication times, potential congestion points, the balance between computation and communication, and scheduling. For example, microbenchmarks such as ping-pong tests are used to measure point-to-point latency and bandwidth [9]. The communication pattern of the application may be static (consistent across runs) or dynamic (changing during runs). By using runtime profiling data, a good optimization strategy is selected to improve performance [10]. Profiling can be conducted in various ways. Communication patterns are analyzed through algorithms during different types of application runs, such as warm-up runs (initial test executions), small-scale test runs, or full-scale production runs (also referred to as real-life or operational runs). In the literature, profiling is categorized as pre-runtime, during-runtime, or post-runtime profiling, or alternatively as online, offline profiling.

In this subsection, approaches that can be used to improve full-scale run performance of the application are divided into two groups, in a way that aligns with the thesis and maintains the flow of explanation. This grouping differs from the classifications commonly presented in the literature. The first group is running the application on HPC and making improvements based on profiling data, and the second group is analyzing the application without running it on HPC and making improvements. The second grouping is possible through modeling and simulation of message counts, communication volume, and communication/computation processes on the topology. However, models and simulation results deviate from the

real execution of the application on the HPC system due to non-deterministic delays caused by parameters such as CPU, caching, software, network connections, memory access, network switches, and errors.

In the first group, based on empirical profiling results obtained from HPC run, task placement on the network, message routing strategies, and computation/communication overlap (especially through non blocking communication) can be adjusted dynamically. The process is as follows: the communication pattern of the application is analyzed to extract a communication graph that represents the interactions between tasks (MPI ranks). This communication graph is first mapped onto a virtual topology within the MPI framework, which is then mapped onto the physical network topology to optimize application performance.

In the second group, the performance of an application can be improved by applying various methods prior to its execution on an HPC system. For example, selecting different partitioning methods and tuning their parameters can help improve communication and computation loads at runtime. Matrix re-ordering techniques, such as changing the order of rows or columns, can improve memory access and communication efficiency. Various partitioning and mapping strategies can be combined and tuned in a topology-aware manner by considering the network topology and communication model of the target HPC system. In addition, topology-aware task distribution can enhance the performance of runtime optimization algorithms by enabling more efficient task placement and communication.

However, it is important to note that methods applied prior to execution may not accurately reflect the application’s runtime behavior on an HPC system. This is because non-deterministic factors during execution, as discussed in this section, can lead to unpredictable delays and communication patterns. Consequently, while optimization strategies prior to execution may improve performance in theory, the desired outcome falls short in practice. To address this, the number of factors causing non-deterministic results can be quantitatively reduced. Factors that cascade and create a snowball effect leading to non-deterministic outcomes can be limited.

2.6 Virtual Processor Topology and Store and Forward Scheme

Irregular and sparse point-to-point (P2P) communication patterns create significant challenges for HPC applications. When message sizes are small and message counts are high, the communication tends to become latency-bound. In contrast, when message sizes are large and message counts are fewer, communication tends to become bandwidth-bound. In sparse and irregular communications, latency often has a greater impact on application runtime due to message count imbalances. These imbalances result in significant differences between the average and maximum number of messages transmitted between any two processors. As a result, nodes that finish communication early may need to wait at synchronization points for other nodes that are still sending or receiving more messages, which reduces overall performance and limits the scalability of HPC applications. Therefore, reducing latency overhead can lead to significant improvements in overall application performance and scalability. Selvitopi and Aykanat [2] proposed a virtual processor topology and a communication scheme that impose structure and predictability on latency-bound applications with irregular and sparse point-to-point communication. They introduced a novel virtual topology and a store-and-forward algorithm, which reduces the communication time and the runtime of parallel sparse matrix-vector multiplication by achieving a trade-off between reduced total latency and increased bandwidth.

2.6.1 Virtual Processor Topology

An example of a physical mesh topology and a virtual processor topology for $K = 16$ processors is presented in Figure 2.8. In the physical topology, solid lines represent actual hardware interconnections between processors. In contrast, in the virtual topology, dashed lines represent logical neighbor relationships. In the HPC domain, virtual topologies are non-physical, logical structures that are

defined in software. The dashed lines illustrated between nodes do not correspond to physical P2P connections. In a virtual topology, a pair of processors can be defined as logical neighbors even if they are not physically P2P connected. Furthermore, nodes connected via dashed lines may not necessarily be logical neighbors; this depends entirely on the specific neighborhood definition used in the virtual topology. Logical neighborhood refers to processors being defined to communicate directly with each other in a virtual manner, independent of their physical placement in the HPC system. This communication is realized using a combination of communication models and algorithms, which enable logical neighbors to exchange data as if directly connected. However, the actual data transfer relies on the underlying hardware connections, such as network interconnects, to route messages between processors. In the physical topology shown in Figure 2.8, processors p_i and p_j are definitely neighbors with a P2P connection, and they are definitely not neighbors with p_h . However, in the virtual topology shown in the same figure, depending on the defined neighborhood relationship, p_i and p_j may or may not be neighbors, p_i or p_j may be a neighbor to p_h . The dashed lines are for illustration purposes only; the actual neighborhood relationships depend on the chosen definition.

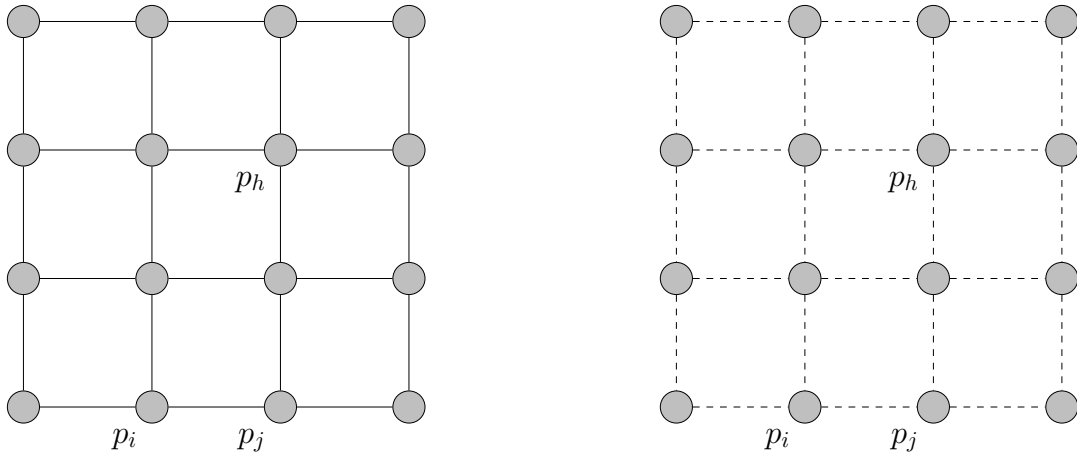


Figure 2.8: Comparison of a 4×4 physical mesh topology (left) and a virtual processor topology (right).

The proposed virtual processor topology (VPT) [2] is inspired by k -ary n -cube (torus, mesh) networks and uses a novel neighborhood definition. In this definition, processors are grouped dimension-wise, and processors within each

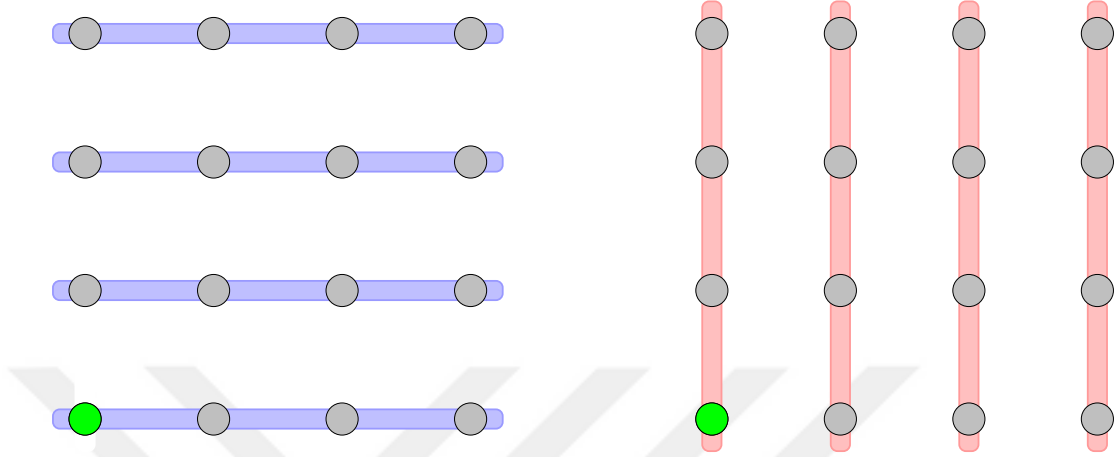


Figure 2.9: Dimension-wise processor groups: row-wise (left) and column-wise (right).

group are defined as neighbors. For example, consider a 4×4 , 2-dimensional k -ary n -cube (4-ary, 2-cube) network as shown on the right in Figure 2.8. In this case, processors are grouped row-wise and column-wise. Processors within each group are neighbors at a virtual 1-hop distance, as if they are connected via point-to-point links. This is illustrated in Figure 2.10, and the lines do not indicate physical connections; they are included only to illustrate virtual neighborhood relationships. In Figure 2.9, the row-wise (left) and column-wise (right) processor groups for a 2-dimensional VPT are illustrated using colored bars. The green node represents a processor p_i . p_i has six virtual neighbors in total, each at a 1-hop distance within their respective dimension-wise groups. In this configuration, each node has the same number of neighbors and the maximum distance between any two processing nodes in the VPT is 2 hops. This structure can be generalized such that, in an n -dimensional VPT, the maximum distance between any two nodes is at most n hops.

The neighborhood relationship is mathematically defined as follows. An n -dimensional VPT with K processors is denoted by $T_n(k_1, k_2, \dots, k_n)$, where the italic T stands for the term topology, and each $k_d \in \mathbb{N}^+$ denotes the size of the d -th dimension, for $1 \leq d \leq n$. The total number of processors is given by $K = k_1 \times k_2 \times \dots \times k_n$. The set of all processor is defined as $\mathcal{P} = \{p_1, p_2, \dots, p_K\}$. Superscripts c and d are used to denote dimensional indices in order to describe

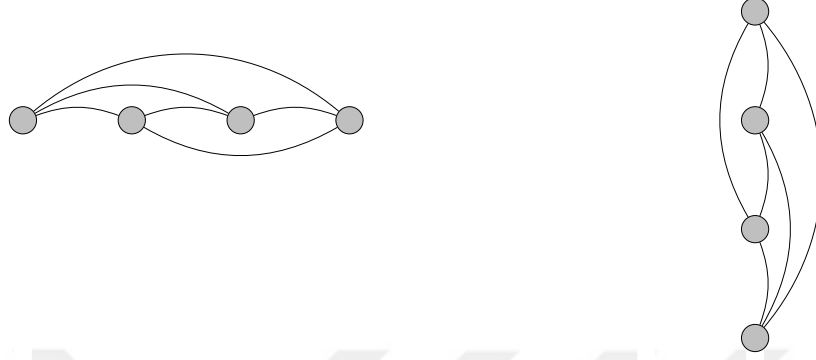


Figure 2.10: Illustration of the neighboring relationships within a processor group: row-wise (left) and column-wise (right).

neighborhood relationships. In the base case, two processors p_i and p_j are identical if and only if $p_i^d = p_j^d$ for all $d \in \{1, \dots, n\}$. In the neighboring case, two processor p_i and p_j are neighbors along the d -th dimension if they differ only in that dimension, while having identical coordinates (i.e., identical scalar values) in all other dimensions c , where $1 \leq c \leq n$ and $c \neq d$. Therefore, the neighborhood relation between processors is defined as follows:

$$p_i^d \neq p_j^d \text{ and } p_i^c = p_j^c \text{ for all } c \in \{1, \dots, n\} \text{ such that } c \neq d \text{ where } d \in \{1, \dots, n\} \quad (2.11)$$

In the thesis, the value of K is chosen as a power of 2. However, a VPT can be constructed for any positive integer K . For example, 64 processors are organized into a 4-dimensional VPT: $T_4(2, 4, 4, 2)$. Consider the coordinates of p_i as $\langle 1, 3, 2, 1 \rangle$ and p_j as $\langle 1, 3, 3, 1 \rangle$. In this case, when the processors are grouped for $d = 2$, p_i and p_j will be neighbors. Their coordinates differ only in the 2nd dimension, while the others are the same. Notice that the 2nd dimension is counted from the rightmost dimension. If the dimensions are considered to follow the order $\langle \text{row}, \text{col}, \text{depth}, \dots \rangle$, the message passing process starts from the last dimension. This is due to the memory layout of the linear indices used for efficient indexing of multidimensional arrays in the VPT, as well as for retrieving the coordinates from those linear indices [11, 12].

Communication in the VPT topology is implemented using the Message Passing Interface (MPI) [13, 14]. MPI is a standardized parallel computing framework

to model and realize communication between computational units. It abstracts the communication layer, making it independent of the underlying hardware infrastructure. Within MPI, each computational unit is assigned an a unique identifier called rank, and communication establish virtually between these ranks. MPI provides a set of communication primitives such as `MPI_Send()`, `MPI_Receive`, and `MPI_Broadcast` to achieve data exchange. To achieve low-latency, high-bandwidth communication in HPC environments, MPI data transfer operations utilize underlying data transfer technology such as RDMA [15], interconnect technologies such as Aries, Slingshot, and InfiniBand, effectively bypasses operating systems mechanisms. In the absence of such advanced technologies, MPI utilizes traditional communication paths provided by operating systems. This is how MPI standardizes communication by providing a software-level interface between application and hardware on various architectures. Communication is carried out between ranks. As mentioned in Section 2.5, MPI’s design enables optimization techniques to monitor and reorganize communication patterns between tasks (ranks) through its abstraction layer. Moreover, these ranks (tasks) can be mapped onto a virtual topology to better align communication with the application’s needs. MPI natively supports only cartesian, graph, and distributed graph virtual topologies [16, 13, 14, 17]. For example, a workload designed for a 2D mesh can be remapped to a 3D torus virtual topology with `MPI_Cart_create` and wrap-around connections. This remapping can enhance performance on physical interconnects in HPC sytems such as Dragonfly or Fat-tree by aligning the communication patterns with the characteristics of the interconnect. Therefore, even if two computation units are defined as neighbors on the virtual topology at 1-hop distance from each other, they can be sent to the destination through more than one network switch and link due to physical interconnects and the routing algorithms of the HPC system. Thus, MPI provides a portable and efficient way to manage inter-process communication, abstracting away the details of the underlying infrastructure across parallel computing environments. Communication models defined with MPI can translate differently onto a parallel computing systems. The actual performance and scalability of a model designed with MPI can be evaluated through the execution of the application on HPC systems.

2.6.2 Store-and-Forward Scheme

As mentioned in Section 2.5, irregular and sparse point-to-point communication patterns often lead to non-deterministic behavior in distributed systems, resulting in unpredictable communication timing, which can lead to increased latency costs. Such non-deterministic behavior is common in small, medium and large scale distributed systems, especially as they scale in size and complexity. Store-and-forward scheme is a communication algorithm that temporarily stores messages and forwards them in bulk, rather than sending them individually. When used with a Virtual Processor Topology, the scheme helps optimize sparse, irregular communications that do not follow a predictable communication pattern.

Various switching methods such as store-and-forward, cut-through, wormhole, and fragment-free switching are used to transmit data across networks. Switching refers to the process of directing data units (or frames) between devices so that they reach their intended destinations. This process occurs within networking hardware such as switches and routers, which control and route these data transfers. The store-and-forward method used in switching stores each entire frame in a buffer until it is fully received. A frame is received sequentially over time, and the store-and-forward method buffers the data until the entire frame is collected. Once the complete frame is assembled, the method validates the frame's integrity and then forwards the entire frame to the next hop or the destination. The proposed store-and-forward scheme used in the VPT borrows the idea of storing data in a buffer and forwarding it towards its destination from this switching method. However, in the store-and-forward scheme, messages do not arrive as fragmented streams, and therefore assembly is not required. Instead, complete messages from one or more sources are accumulated into a contiguous buffer (MPI Send buffer) and forwarded to their next hop. The key feature of the proposed scheme is that communication is carried out in multiple stages using a Dimension-Ordered Routing (DOR) approach, analogous to e-cube routing used in hypercube topologies [ref].

In Dimension-Ordered Routing, each message traverses the topology in a fixed

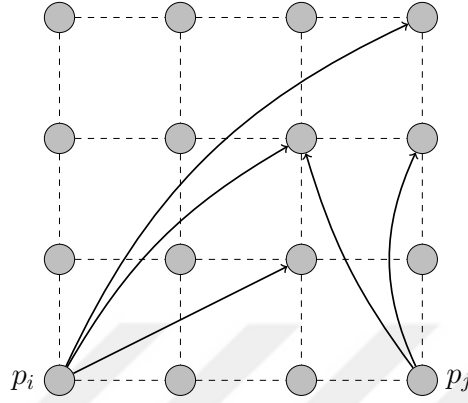
dimension order. All messages sent over the network use the same routing order regardless of their source or destination. For example, in a 2-dimensional mesh topology (with x and y dimensions), if the sender and receiver differs along both axes, the message will first travel in the x direction, then in the y direction. In DOR, messages are sent through the network dimensions without synchronization. This means each message travels independently whenever it can, following its calculated route (e.g., x then y order) without waiting for other messages to complete their transmission. The store-and-forward scheme transmits messages in a manner similar to dimension-ordered routing. In an n -dimensional VPT, communication is carried out in n distinct stages, with each stage corresponding to message transmission along one dimension. The critical difference is that the transmission along the next dimension cannot begin until all message transfers of a processor in the current dimension are completed. In other words, at each stage, communication occurs within the processors of processor group k_d , i.e., among neighbors along the d -th dimension, and the communication stages/phases proceed synchronously, implicitly enforcing a synchronization barrier between stages. It is important to note that there is no global barrier; instead, each processor independently waits for all message transfers in the current stage to complete before proceeding to the next dimension. This local synchronization ensures correct stage ordering without incurring the overhead of global synchronization. However, the requirement that messages must be completely received before being sent out introduces an added delay at each hop for each message in contrast to the store-and-forward switching method.

This implicit synchronization model complicates the methodology for measuring communication latency and bandwidth. Since processors do not start or finish each stage simultaneously, one processor may proceed to the next stage while another is still completing the current one. For example, once a processor p_i finishes its stage 1 communication, it proceeds to stage 2 and may send messages to processor p_j . If p_j has not yet completed its stage 1 communications, it cannot receive messages from p_i until it finishes its stage 1 communication. As a result, the bottleneck processor (that is, the one with the maximum communication latency or bandwidth) may vary from stage to stage. It is also worth

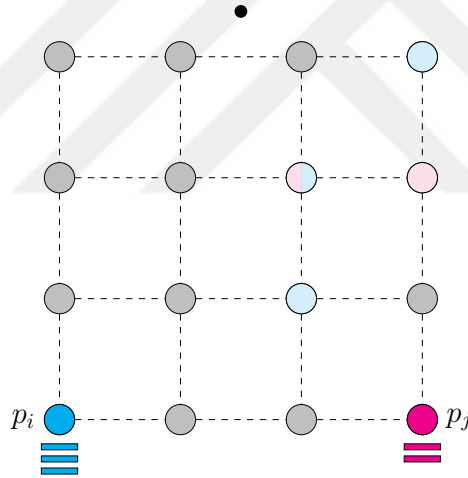
noting that in MPI, a `Send()` operation may complete before the receiver posts a matching `Recv()`, due to internal buffering handled by the MPI library. This occurs as MPI may internally copy the data from the user-provided buffer into its own internal buffer, allowing the sender to continue without waiting for the receiver. Therefore, one possible approach to evaluating latency and bandwidth is to identify the bottleneck processor at each stage, as these processors may influence overall communication performance.

Figure 2.11 illustrates the communication process in a 2-dimensional VPT $T_2(4, 4)$, and demonstrates how the store-and-forward communication scheme operates in stages. Subfigure (a) presents the high-level communication pattern on the VPT. Each node represents a virtual processor, and arrows points messages sent from sender processors p_i and p_j to their respective receiver processors. The subfigure (a) provides an abstract overview of communication pairs without showing routing details. Subfigures (b), (c), and (d) present the communication process in three stages using a color-coded visualization and omitting arrows to reduce visual clutter and improve clarity. The sender processor p_i is colored cyan, and sender p_j is colored magenta. Receiver processors are colored with lighter (more transparent) versions of the corresponding sender's color to visually indicate the source of each message, in order to create a clear visual association between senders and receivers. Below each sender processor, colored rectangular blocks represent messages. The position of each message block under a specific processor indicates the current location of the message (i.e., the processor's buffer) at the end of each stage. The communication starts with the initial state and proceeds through two stages. Subfigure (b) presents the initial state. This is the starting point of the communication, where all messages are located at their original sender processors. Subfigure (c) presents communication stage 1, where messages are forwarded along rows, and subfigure (d) presents communication stage 2 where remaining messages are forwarded along columns.

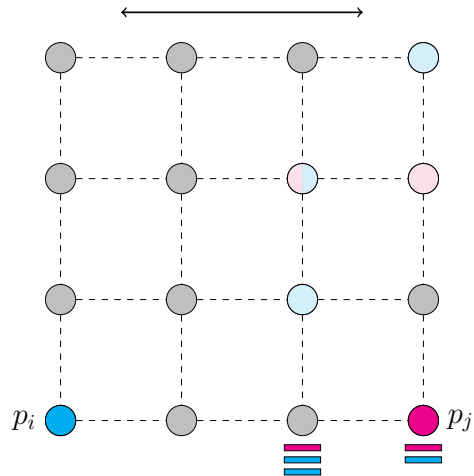
(a) communication in 2-dimensional VPT , $T_2(4, 4)$



(b) initial state of communication



(c) comm. stage 1: row-wise ($d = 1$)



(d) comm. stage 2: col-wise ($d = 2$)

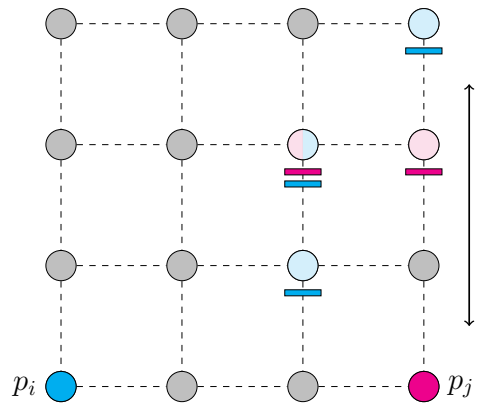


Figure 2.11: Stages of Communication in a VPT $T_2(4, 4)$ Using the Store-and-Forward Scheme.

In the store and forward scheme, messages move step by step along the VPT grid following a dimension ordered routing approach, first along one VPT dimension (e.g., rows) and then along the next (e.g., columns). As mentioned earlier, the intermediate positions of the message boxes are shown below the processors to illustrate their progress through the topology. In Stage 1, if destination of a message lies within the same row (processor group), it is sent directly to the destination processor. Otherwise, the message is forwarded to an intermediate processor within the same row that aligns with the column of the destination processor in the VPT. This intermediate processor buffers the message to be forwarded in the next stage. In Stage 2, the remaining messages are forwarded column-wise (across processor groups) from the intermediate processors to their final destinations, completing the routing process. These stages demonstrate how messages are buffered together at intermediate processors, instead of being sent independently and asynchronously (i.e., without stage synchronization). By introducing structured communication steps with implicit synchronization between stages, the store-and-forward scheme in VPT effectively reduces communication contention and regularizes traffic patterns. In addition, notice that the message sent by p_i is being sent to different processors within the same column. In the first stage, the same message will be written to the buffer twice. This can be avoided by sending a single copy of the message in the first stage. In the second stage, the intermediate processor can then send the copied message to the two processors in the same column. This approach avoids sending the same message multiple times, thereby reducing communication volume. This method is referred to as 'expand-aware' [18, p. 53]; however, it is not included in the thesis. Since the routes of the messages are known in advance, the method can be applied in practice.

The routing algorithm is achieved by sequentially comparing the coordinates of the processors. For example, consider a 3-dimensional VPT $T_3(4, 8, 2)$ represented as $\langle \text{row}, \text{column}, \text{depth} \rangle$. Let the source processor coordinate be $p_s = \langle 3, 1, 2 \rangle$ and the destination processor coordinate be $p_d = \langle 1, 3, 2 \rangle$. The store-and-forward routing proceeds by comparing coordinates dimension by dimension, starting from the rightmost dimension (depth) and moving leftwards through column and row.

This routing order is chosen for implementation reasons discussed in Section 2.6.1 due to the algorithms used in the multidimensional array indexing strategy. Since the depth values of both p_s and p_d are the same (2), the message remains buffered during the first stage. In the second stage, the column coordinates differ, so the message is forwarded to the intermediate processor whose coordinate aligns with the destination in the column dimension, that is, $\langle 3, 3, 2 \rangle$. In the third and last stage, the row coordinates differ, the message is forwarded to the processor aligned with the destination row coordinate, $\langle 1, 3, 2 \rangle$. In this example, the message requires two forwarding steps to reach its destination. This is referred as store-and-forward distance. At each stage d , if the source and destination coordinates differ in dimension d , the message is forwarded to align with the destination's coordinate in that dimension. Thus, the total number of hops is the sum of coordinate differences across all dimensions. This can be expressed as:

$$\text{STFWDistance}(p_i, p_j) = \sum_{d=1}^n \begin{cases} 1 & \text{if } p_i^d \neq p_j^d, \\ 0 & \text{otherwise} \end{cases} \quad (2.12)$$

This sum is known as the Hamming distance [19] between the source and destination processor coordinates, which counts the number of differing dimensions. In the case of Figure 2.11, let the source processor be $p_i = \langle 1, 1 \rangle$, the destination processor be $p_j = \langle 1, 4 \rangle$, and the upper-right processor be $p_h = \langle 4, 4 \rangle$. The store-and-forward message routing from p_i to p_h proceeds in two stages: $\langle 1, 1 \rangle \rightarrow \langle 1, 4 \rangle \rightarrow \langle 4, 4 \rangle$.

In addition, there are two special cases where the VPT with the store-and-forward scheme coincides with existing topologies. Recall that an n -dimensional VPT with K processors is denoted as $T_n(k_1, k_2, \dots, k_n)$, where $n \in \mathbb{N}^+$ and $k_d \in \mathbb{N}^+$ represents the size of the d -th dimension, for $1 \leq d \leq n$. In the first case, where $n = 1$, all K processors belong to the same single processor group, and therefore the VPT structure becomes equivalent to a full mesh topology, as illustrated in Figure 1. As a result, the 1-dimensional VPT serves as the full-mesh model, also referred to as the baseline (BL) model in the study [2]. This VPT T_1 baseline model is used in the experiments as a reference point representing the model without regularization. In the second case, where $n \in \mathbb{N}^+$ and $k_d = 2$ for all $d \in \{1, \dots, n\}$, the n -dimensional VPT serves as a hypercube model with

E-cube routing. Since each dimension has size 2, the total number of processors is $K = 2^n$. Therefore, the model can be expressed as a $\log_2 K$ -dimensional VPT.



Chapter 3

Motivation and Related Works

Latency is a critical bottleneck in high-performance computing, especially for applications with irregular communication patterns such as sparse matrix-vector multiplication (SpMV). While buffering in the store-and-forward scheme increases memory usage, this is typically not a major concern in HPC systems where memory resources are relatively abundant. The more significant overhead arises from the implicit synchronization barriers between communication stages (as discussed in Section 2.6.2), which can increase latency due to imbalances between the average and maximum number of messages transmitted between any two processors. However, the VPT provides a communication structure well suited for the store-and-forward scheme, reducing the number of hops each message must traverse at the cost of increased bandwidth usage on links. The store-and-forward routing strategy significantly reduces the total number of messages transmitted by aggregating data into the buffer of each MPI Rank before forwarding. Together, the VPT and the store-and-forward scheme help balance the load across network links by reducing the difference between the average and maximum (i.e., worst case) number of messages transmitted per link, thereby improving overall communication latency.

The store-and-forward scheme in VPT was tested with sparse matrix-vector multiplication operations. The test matrices were partitioned using PaToH [20]

with a row-wise partitioning scheme for small, medium, and large-scale HPC systems. In addition, this approach is compatible with any partitioning algorithm. It is worth noting that the partitioning tools are used to reduce communication overheads. For the matrix multiplication tests, the row-parallel SpMV algorithm was used, which consists of a communication phase followed by the computation phase (as discussed in Section 2.2.3). And the tests were conducted for VPTs with sizes between $2 \leq n \leq \log_2 K$. The method significantly reduces the maximum and average message counts. It reduces the maximum message count by 3.3-7.4 times, 3.6-10.6 times, 4.6-15.1 times, 4.5-20.9 times, 5.6-45.5 times, 5-53.4 times, and 6.5-75.3 times for $K = 64, 128, 256, 512, 4096, 8192$, and 16384 respectively. In each range, the first value corresponds to a 2-dimensional VPT, the last value corresponds to a $\log_2 K$ -dimensional VPT, and the values in between correspond to intermediate VPT dimensions. And the method reduces average message count 3-8 times. On large-scale systems, the method significantly reduces communication time. On the Cray XK7 (3D Torus, Gemini interconnect), it achieves up to 94% and 95% improvement compared to the baseline (BL) at 8192 and 16384 processors, respectively. On the medium-scale Cray XC40 (Dragonfly, Aries interconnect), it achieves 86% reduction compared to baseline at 4096 processors. On small-scale systems, the method reduces communication time by up to 50%, 55%, 61%, and 72% compared to the BL for $K = 64, 128, 256$, and 512, respectively. Additionally, it reduces parallel SpMV runtime up to 12%, 22%, 44%, and 65% compared to the BL for the same values of K .

It should be noted that these significant improvements are achieved at the expense of increased communication volume. In their study [2], the authors addressed this downside and proposed an approach for further improvement: strategically placing (remapping) processes (tasks, MPI ranks) to the VPT in a way that reduces communication volume by minimizing the Hamming distance [19] between process pairs assigned to processors. There are $K!$ possible permutations for mapping K processes onto the Virtual Processor Topology. Therefore, in order to reduce the size of the solution space, a selection strategy must be applied. Some form of restriction or heuristic can reduce the search space. To

minimize communication volume, one possible approach is to reduce the Hamming distance of processes that have high communication loads with others. For example, as illustrated in Figure 2.11, the computation task assigned to processor p_i communicates with several targets, and all its targets located at a Hamming distance of 2. Reassigning the p_i 's task closer to its communication targets would reduce both total number of messages and the total communication volume (calculated as buffer size multiplied by Hamming distance). In the Figure 2.11, this can be achieved by assigning the p_i 's task to the processor located at the intersection point of its communication targets along X and Y axes in the upper row. This reassignment (remapping) reduces the Hamming distance to one (1 hop), as a result lowering the number of messages and the total communication volume. In such cases, remapping may not significantly reduce the number of messages, as messages are aggregated into a single buffer and transmitted using dimension-order routing. Therefore, the overall maximum and average message count may remain similar. However, total communication volume can still be reduced by better task placement. It is important to note that, in the case of irregular sparse matrices that are partitioned and mapped one-to-one onto a processor topology, many nodes are involved in communication to some extent. The reason is that, partitioners aim to balance computational load across partitions while minimizing communication between partitions to improve parallel execution performance, even when the partitioning is not topology-aware. However, irregular sparsity patterns often result in significant communication between partitions despite the optimization goals of the partitioner. As a result, reassigning a task to reduce its communication cost may lead to a similar increase in the communication cost of the task previously assigned to that location.

Relocating a task is a local modification that may shift communication costs elsewhere, whereas a global solution can improve the overall communication volume. Probabilistically, within the solution space, there exist task-to-processor topology mappings that yield the lowest communication volume, as long as the current solution is not the optimal one (i.e., best possible). This problem is commonly referred as the topology mapping problem, which is known to be NP-hard [21]. Topology mapping can be modeled as a graph embedding problem, where

the application's communication graph (guest graph) is mapped onto the processor topology (host graph). In this context, the goal is to assign application tasks to physical processors in a way that minimizes communication overhead. The communication overhead often modeled as the product of message size and the distance between processors. The modeled function, referred to as cost function, represents the communication cost in HPC systems. This modeling of the topology mapping problem resembles the Quadratic Assignment Problem (QAP) [22, 21], a well-known combinatorial optimization problem. In QAP, the cost defined as the product of a weight (or flow) and the distance between the assigned locations. The communication cost between a pair of tasks i and j given by:

$$cost(i, j) = w(i, j) \times d(map(i), map(j)) \quad (3.1)$$

$w(i, j)$ denotes the message size transmitted from task i to j , and $map(\cdot)$ denotes the processor to which a task is assigned. The total communication cost for a given mapping is the sum over all pairs which corresponds to the element-wise product of the weight and distance interaction matrices:

$$TotalCost = \sum_{i=1}^K \sum_{j=1}^K w(i, j) \cdot d(map(i), map(j)) \quad (3.2)$$

Since there are $K!$ possible task-to-processor assignments (permutations), the mapping that yields the minimum total communication cost is the optimal solution. This can be formulated as:

$$\min_{\pi \in S} \sum_{i=1}^K \sum_{j=1}^K w(i, j) \cdot d(\pi(map(i)), \pi(map(j))) \quad (3.3)$$

In this equation, S denotes the solution space consisting of all $K!$ permutations, the Greek letter pi (π) denotes the initial letter of the word permutation, representing one possible mapping of tasks to processors. And the $\pi(map(i))$ refers to the processor assigned to task i .

Selecting the exact solutions is impractical due to the combinatorial explosion of the solution space. For smaller problem sizes (K values), an exhaustive search can be applicable. However, for larger K values, such an approach becomes computationally infeasible. As a result, an algorithm is required to efficiently select a

good solution among all the possible configurations. Solving the QAP is achievable through methods used to solve NP-hard problems, including but not limited to, approximation algorithms, metaheuristic, heuristic, iterative, and probabilistic methods. In the context of task-to-processor mapping, several state-of-the-art approaches demonstrated significant performance improvements. Hoefler and Snir [23] proposed a heuristic based on graph similarities. Their method reduced network congestion by up to 80%. and improved communication performance by 18%. Glantz et al. [24] proposed two variants that build upon the heuristics introduced by Hoefler and Snir [23], and Brandfass et al. [25]. The limitations of the original methods were addressed through a greedy approach. In their experiments, they observed that their greedy heuristic, referred to as GreedyAllC, a variant of the heuristic proposed by Brandfass et al., outperformed all other state-of-the-art algorithms with respect to maximum congestion. The algorithm is executed with a time complexity of $O(|V_c||E_c|)$, where V_c denotes the vertex set and E_c denotes the edge set of the communication graph. The proposed mapping algorithms demonstrated significantly better performance than random mappings and offered fast, scalable, and performance-enhancing solutions to the NP-hard task-to-processor assignment problem.

The topic of task-to-processor mapping leads to a broader consideration of how tasks are initially partitioned, and whether the partitioning algorithms themselves are topology-aware. In HPC, a wide range of partitioning strategies have been developed and tailored to specific application domains. These strategies often depend on several factors, such as the mathematical formulation of the problem, the structure and sparsity pattern of the matrices, the target hardware architecture, the way data is exchanged between tasks, and the communication model used (e.g., message passing, shared memory). Although a particular partitioning method might be designed with one type of problem in mind, it can sometimes perform well in other contexts. Conversely, a method that achieves good load balancing or minimizes edge-cuts in one application may perform poorly in another, primarily due to differences in communication-to-computation ratios or irregular data dependencies. In addition, there are other determining factors such as hardware topology, memory hierarchy, task granularity, and the dynamic behavior of

workloads. This demonstrates that no universally optimal partitioning strategy exists. Therefore, various partitioning methods and tools have been developed to distribute computations and data efficiently in parallel systems. Graph based methods such as METIS [26], Scotch [27], and KaHIP [28] represent computations as vertices and communication as edges. Hypergraph based methods such as PaToH [20, 29], hMETIS [30], Mondriaan [31], and Zoltan [32] use hyperedges to represent unpredictable patterns in irregular applications like sparse matrix-vector multiplication. Geometric methods such as those in Zoltan use coordinates to efficiently partition structured grids. Hierarchical partitioning, supported by Zoltan and implemented by Scotch, recursively divides the problem to align the workload with multi-level hardware architectures such as clusters of NUMA nodes and heterogeneous CPU-GPU systems. In addition, Zoltan and Scotch provide topology-aware partitioning that maps tasks to the physical system layout to minimize costly communication.

Given the close relationship between partitioning and mapping, their combined impact on performance has been a focus of research. It can be hypothesized that better partitioning improves load balancing and enhances mapping performance. However, studies show that partition quality has little impact on mapping effectiveness in many applications [24, 33, 34, 35]. Moreover, task-to-processor mapping methods can achieve good results even with suboptimal partitioning, unless it is very poor, and can be more effective than high-quality partitioning in improving overall performance, especially in terms of communication efficiency and network congestion. Effective mapping algorithms can compensate for poor partitioning, achieving results close to or better than those with high-quality partitioning.

While both partitioning and mapping significantly impact performance, mapping offers greater opportunities for optimization. Mapping methods vary in their approach to assigning tasks to processors. The most commonly used methods are constructive and iterative algorithms (metaheuristics are moderately used). Iterative algorithms refine an existing solution, whereas constructive algorithms create a solution. Another common strategy is to combine constructive and iterative methods.

A constructive algorithm generates an initial solution, then the solution is refined by an iterative algorithm. Constructive algorithms build a solution from scratch in a single pass, making local decisions without revisiting previous choices. Initially, no solution exist, and the algorithm builds solution step by step based on a decision strategy. A common example is a greedy algorithm that starts with no solution and, at each step, assigns a task to the processor that yields the lowest communication cost, making local decisions without considering past or future consequences. In general, constructive algorithms are computationally efficient and favorable for generating initial solutions, and they can be quicker than iterative methods.

Iterative algorithms refine an existing solution repeatedly until a stopping condition is met, such as reaching an iteration limit or achieving a target cost value. They can start with a random solution or an existing solution (often generated by a constructive algorithm). They aim to optimize an objective function, such as minimizing communication volume or load imbalance. The success of iterative algorithms is highly dependent on the quality of the initial solution. They are preferred when higher solution quality is required or when dealing with complex problems (e.g., irregular topologies or large-scale systems). Generally, their computational cost is higher than the computational cost of constructive algorithms. Graph partitioning algorithms such as Fiduccia-Mattheyses (FM) [36] and Kernighan-Lin (KL) [3] are widely used for this purpose. They achieve partitioning by moving or swapping vertices between processors to reduce edge cuts, which represent the cost function for total communication volume, minimizing communication cost. The FM algorithm refines a bipartition of a graph by moving a single vertex from one partition to another in each step. It calculates the gain which is the reduction in the number of edges crossing partitions (i.e., edge-cut, communication cost) and selects the move with the highest gain in a pass, keeping the best partition found. FM is more efficient than KL with a time complexity of $O(n)$ per pass, where n is the number of vertices, and allows partitions to have slightly unequal sizes within a specified tolerance. In contrast, the KL algorithm refines a bipartition of a graph by swapping a pair of vertices, one from each partition, to reduce the edge cut. It evaluates multiple swaps in a pass,

selecting the swap with the highest gain, and keeps the best partition found. KL is more computationally intensive, with a time complexity of $O(n^2)$ per pass. KL ensures strict balance by maintaining equal partition sizes as swapping one vertex from each partition inherently keeps partition sizes equal.

In this thesis, a KL-based iterative optimization heuristic is proposed to reduce the total communication volume by improving one-to-one task-to-processor mapping onto VPT with store-and-forward scheme. Selvitopi and Aykanat [2] demonstrated that the use of a VPT with a store-and-forward scheme significantly reduced latency at the expense of increased communication volume. To mitigate the increased communication volume, they suggested reducing the Hamming distance between task pairs with high communication requirements. The proposed KL-based heuristic iteratively refines the task-to-processor mapping by performing task swaps on the VPT. At each iteration, among all possible swaps, the swap yielding the highest gain is selected, and this process continues until no feasible task pairs remain. At the end of each pass, a hill-climbing algorithm keeps the best mapping found that yields the maximum gain. The iterative refinement continues, with a greedy selection of the highest-gain task pair, until the total gain becomes zero. Due to this design, the heuristic structurally resembles the KL algorithm. To make the problem computationally feasible, swaps are constrained to processor groups within the VPT, reducing the solution space for task pair swaps in a k -ary n -cube VPT from $C(k^n, 2) = \binom{k^n}{2}$ to $(n \cdot k^n \cdot (k-1))/2$, a reduction by a factor of $(k^{n-1})/(n \cdot (k-1))$.

This work is based on [2] and, as a continuation, investigates the potential for further improving random and smart task-to-processor mappings on the VPT using the iterative refinement algorithm described in Section 4. The process is outlined as follows:

1. As in [2], the selected sparse matrices are partitioned nonsymmetrically (Section 2.2.5) using PaToH with a row-wise, column-net partitioning scheme.

2. The resulting partitions are randomly assigned to the VPT using a one-to-one task-to-processor mapping. This random mapping is then improved using smart mapping methods [23, 24].
3. The proposed KL-based iterative optimization heuristic is applied to further improve both the random and smart mappings.
4. The performance of the KL-based heuristic is evaluated and compared against other mapping methods.

Chapter 4

Iterative Optimization Heuristic for Reducing Total Communication Volume

In this section, the research conducted to improve one-to-one task-to-processor assignments using Kernighan-Lin (KL)-based heuristics is presented, with a focus on reducing the additional communication volume incurred by the increased number of messages routed through the Virtual Processor Topology (VPT) in the store-and-forward method. As previously stated, the one-to-one assignment problem for a parallel system with K processors is defined as the assignment of K computational tasks to K processors. In other words, each processor is responsible for executing exactly one computational task. Since the number of computational tasks equals the number of processors in this context, these tasks are referred to in the literature as coarse-grain computational tasks. For simplicity, they will be referred to as tasks throughout the thesis. KL-based heuristics target the reduction of communication volume by positioning tasks/(processors in report) with high-volume communication closer to each other on the VPT. The proposed heuristic aims to reduce the total volume of store-and-forward communication by swapping tasks positioned on the VPT, utilizing a hill-climbing strategy to iteratively explore solutions.

The KL algorithm was designed and implemented for 2 and 3-dimensional VPT and Hypercube-VPT structures. As a result of this design process, a generalized algorithm was developed for VPT topologies in network structures with two or more dimensions. Based on the findings presented in Section 4.2, an algorithm was developed for n -dimensional VPT topologies that is more efficient in terms of time and memory without compromising the correctness of the heuristic and is presented in Section 4.3.

4.1 Reduction of the Store-and-Forward Communication Volume Through Pairwise Task Swap on VPT

In the store-and-forward method, messages are sent sequentially over the dimensions of the VPT. Swaps are performed between task pairs located along the direction of these dimensions, i.e., within the same processor group. To illustrate the concept, it is assumed that communication in the first stage occurs over rows and in the second stage over columns on a two-dimensional VPT. For ease of explanation, the 2D VPT is represented as $\sqrt{K} \times \sqrt{K}$. As discussed, in a two-dimensional VPT, the distance between two tasks is equal to the number of steps required for communication and is at most two. Swap gains are calculated based on the pairwise combinations of tasks located along the same rows and columns. To ensure consistency in notation throughout the thesis, t_i and t_j are reserved to represent the pair of tasks involved in a swap operation. Additionally, t_h is used to denote the tasks that either send to or receive from t_i or t_j . Consider a message m_{ih} that requires two steps to reach its destination over the VPT. The reason for the two-step requirement of m_{ih} is that tasks t_i and t_h are assigned to different rows and columns in the VPT. To reduce the total volume contribution of this message and achieve a gain, during the swap, the processor currently assigned to task t_i (or t_j) must be reassigned to the row or column in which task t_j (or t_i) is located, ensuring that t_j is in the same row or column as t_h . The swap operation

must be performed in this manner to obtain a gain from the m_{ih} message.

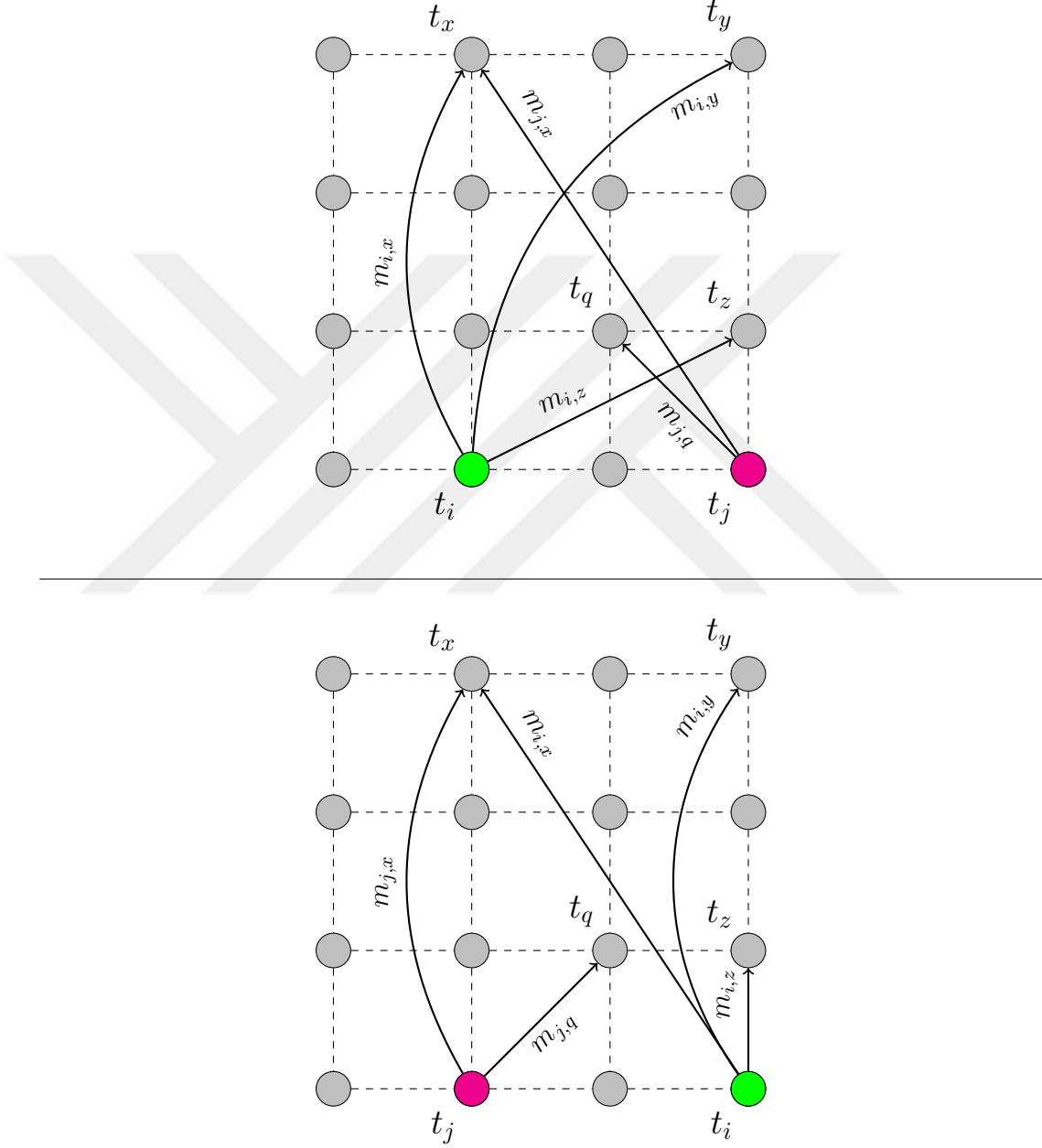


Figure 4.1: The swapping of the assignments of tasks t_i and t_j within the same row in a 2-dimensional 4×4 VPT $T_2(4, 4)$.

Figure 4.1 illustrates the swapping of assignments between tasks t_i and t_j , which are located in the same row and have send sets of $\{m_{i,x}, m_{i,y}, m_{i,z}\}$ and

$\{m_{j,q}, m_{j,x}\}$, respectively. For simplicity, without loss of generality, it is assumed that t_i and t_j only send messages and do not receive any. The initial configuration is shown at the top, and the post-swap configuration is shown at the bottom. The swapped task pair is highlighted in the figure for visibility. It can be observed that the distances to tasks in the send lists of t_i and t_j are either one or two steps. Following the swap, all store-and-forward distances change, except for the distance to t_q . The total communication volumes incurred in the VPT shown in Figure 4.1 before and after the swap can be expressed as follows:

$$\begin{aligned} Vol_{pre-swap} &= 1 \cdot |m_{i,x}| + 2 \cdot |m_{i,y}| + 2 \cdot |m_{i,z}| + 2 \cdot |m_{j,x}| + 2 \cdot |m_{j,q}| \\ Vol_{post-swap} &= 2 \cdot |m_{i,x}| + 1 \cdot |m_{i,y}| + 1 \cdot |m_{i,z}| + 1 \cdot |m_{j,x}| + 2 \cdot |m_{j,q}| \end{aligned} \quad (4.1)$$

Here, $|\cdot|$ denotes the size of the corresponding message. $Vol_{pre-swap}$ and $Vol_{post-swap}$ represent the total communication volumes incurred in the VPT after swapping tasks t_i and t_j . If t_i (or t_j) were to receive messages from another task, the associated contribution to the communication volume would be expressed as a positive term in the formula. The total communication volume corresponds to the sum of the communication costs incurred by all sent and received messages across all stages. The swap gain is the difference in total communication volume between the pre-swap and post-swap states. The swap gain is expressed as follows:

$$SwapGain(t_i, t_j) = Vol_{pre-swap} - Vol_{post-swap} = |m_{i,y}| + |m_{i,z}| + |m_{j,x}| - |m_{i,x}| \quad (4.2)$$

In this expression, the terms $|m_{i,y}| + |m_{i,z}|$ and $|m_{j,x}|$ are positive because the store-and-forward distance of the corresponding messages is reduced from 2 hops to 1 hop. Conversely, the term $|m_{i,x}|$ is negative due to an increase in the store-and-forward distance from 1 hop to 2 hops. This demonstrates that a swap is considered successful when it results in a decrease in the total distance traveled by messages after the swap.

The heuristic is explained through an example and formalized using mathematical equations to demonstrate the process. Building upon this, the same

concept is restructured by dividing the problem into smaller, independent parts. By adopting a methodology inspired by divide-and-conquer principles, the process is organized to address individual components, ultimately leading to the overall solution. In the alternative formulation, the swap gain can be computed based on the individual gains of each task. A difference in communication volume results from swapping the processor assignments of t_i and t_j . This difference corresponds to the gain. The contribution of t_i to the (t_i, t_j) swap gain can be calculated by subtracting the communication volume incurred by t_i in its position before the swap, from the communication volume incurred by t_i in its position after the swap.

The notation $Vol(t_i, map(t_i))$ is used to represent the communication volume incurred by t_i in its original position before the swap, while $Vol(t_i, map(t_j))$ represents the communication volume incurred by t_i in t_j 's position after the swap. The equations for the communication volumes incurred by t_i in these two positions are as follows:

$$\begin{aligned} Vol(t_i, map(t_i)) &= 1 \cdot |m_{i,x}| + 2 \cdot |m_{i,y}| + 2 \cdot |m_{i,z}| \\ Vol(t_i, map(t_j)) &= 2 \cdot |m_{i,x}| + 1 \cdot |m_{i,y}| + 1 \cdot |m_{i,z}| \end{aligned} \quad (4.3)$$

The change in total communication volume resulting from the swap of t_i with t_j is represented by $Gain(t_i, map(t_j))$ and the swap gain calculation for t_i is expressed as follows:

$$\begin{aligned} Gain(t_i, map(t_j)) &= Vol(t_i, map(t_i)) - Vol(t_i, map(t_j)) \\ &= -1 \cdot |m_{i,x}| + 1 \cdot |m_{i,y}| + 1 \cdot |m_{i,z}| \end{aligned} \quad (4.4)$$

The same approach is applied to t_j . The change in total communication volume for t_j following its swap with t_i is denoted as $Gain(t_j, map(t_i))$. The swap gain calculation for t_j is given as follows:

$$\begin{aligned} Vol(t_j, map(t_j)) &= 2 \cdot |m_{j,x}| + 2 \cdot |m_{j,q}| \\ Vol(t_j, map(t_i)) &= 1 \cdot |m_{j,x}| + 2 \cdot |m_{j,q}| \end{aligned} \quad (4.5)$$

$$\begin{aligned}
Gain(t_j, map(t_i)) &= Vol(t_j, map(t_j)) - Vol(t_j, map(t_i)) \\
&= 1 \cdot |m_{j,x}| + 0 \cdot |m_{j,q}|
\end{aligned} \tag{4.6}$$

Thus, the swap gain is computed as the sum of the individual gains of t_i and t_j . The explicit formula for $SwapGain(t_i, t_j)$ as follows:

$$\begin{aligned}
SwapGain(t_i, t_j) &= Vol(t_i, map(t_i)) - Vol(t_i, map(t_j)) \\
&\quad + Vol(t_j, map(t_j)) - Vol(t_j, map(t_i))
\end{aligned} \tag{4.7}$$

Two significant conclusions can be drawn from the gain calculation analysis:

1. The gain resulting from the swap is solely determined by changes in the store-and-forward distances.
2. The computationally intensive components in the calculation of the (t_i, t_j) swap gain, such as $Vol(t_i, map(t_i))$, are inherently separable and can be computed independently.

Building on these two conclusions, separate algorithms can be developed for the heuristic. As a preliminary step, an algorithm is developed to compute the gain associated with changes in store-and-forward distances. This approach provides a clearer understanding of the algorithm's boundary conditions and computational complexity. Building on these observations, the final algorithm is designed by leveraging the second key observation, resulting in a robust and efficient solution designed for modern computer architectures and caching mechanisms.

4.2 Algorithm Design for 2D Topology

As illustrated in Figure 4.1, for row-wise task swaps, gain-contributing communicating tasks are located solely in the same column as the swapped tasks. Depending on the direction of the task, the distance either increases or decreases.

Algorithm 1 presents the steps involved in calculating the gain associated with swapping the processor assignments of the task pair t_i and t_j located on the same row. The swap gain is calculated based on the destinations of the messages sent by t_i and t_j . Specifically, after the tasks are swapped, the gain is determined as follows: messages sent by t_i are evaluated through $SendSet(t_i)$ (lines 3–11), while messages sent by t_j are evaluated through $SendSet(t_j)$ (lines 22–30). The loop in lines 3–11 of Algorithm 1 performs row and column checks for each target task in $SendSet(t_i)$. The conditional check in line 4 evaluates whether the target task is on the same row, in which case it does not contribute to the gain, as the store-and-forward distance remains unchanged. Conversely, the checks in lines 5 and 7 evaluate whether the target task is in the same column as t_j or t_i , respectively. In these cases, the store-and-forward distance changes, and the task can contribute to the gain. The loop in lines 22–30 operates analogously for t_j . Similar calculations are also performed for the $ReceiveSet(t_i)$ and $ReceiveSet(t_j)$ in lines 12–20 and 31–39, and the resulting values are added to the total gain associated with the swap.

Algorithm 1 Calculation of the swap gain for the tasks t_i & t_j on the same row

Require: $t_i, t_j, \text{SendSet}, \text{ReceiveSet}$

```
1: currentRow  $\leftarrow t_i.\text{row} = t_j.\text{row}$ 
2: rgain_ti  $\leftarrow 0$ 
3: for  $t_h \in \text{SendSet}(t_i)$  do
4:   if  $t_h.\text{row} \neq \text{currentRow}$  then
5:     if  $t_h.\text{col} = t_j.\text{col}$  then
6:       rgain_ti  $\leftarrow \text{rgain\_ti} + |m_{i,h}|$ 
7:     else if  $t_h.\text{col} = t_i.\text{col}$  then
8:       rgain_ti  $\leftarrow \text{rgain\_ti} - |m_{i,h}|$ 
9:     end if
10:  end if
11: end for
12: for  $t_m \in \text{ReceiveSet}(t_i)$  do
13:  if  $t_m.\text{row} \neq \text{currentRow}$  then
14:    if  $t_m.\text{col} = t_j.\text{col}$  then
15:      rgain_ti  $\leftarrow \text{rgain\_ti} + |m_{i,m}|$ 
16:    else if  $t_m.\text{col} = t_i.\text{col}$  then
17:      rgain_ti  $\leftarrow \text{rgain\_ti} - |m_{i,m}|$ 
18:    end if
19:  end if
20: end for
21: rgain_tj  $\leftarrow 0$ 
22: for  $t_k \in \text{SendSet}(t_j)$  do
23:  if  $t_k.\text{row} \neq \text{currentRow}$  then
24:    if  $t_k.\text{col} = t_i.\text{col}$  then
25:      rgain_tj  $\leftarrow \text{rgain\_tj} + |m_{j,k}|$ 
26:    else if  $t_k.\text{col} = t_j.\text{col}$  then
27:      rgain_tj  $\leftarrow \text{rgain\_tj} - |m_{j,k}|$ 
28:    end if
29:  end if
30: end for
31: for  $t_n \in \text{ReceiveSet}(t_j)$  do
32:  if  $t_n.\text{row} \neq \text{currentRow}$  then
33:    if  $t_n.\text{col} = t_i.\text{col}$  then
34:      rgain_tj  $\leftarrow \text{rgain\_tj} + |m_{j,n}|$ 
35:    else if  $t_n.\text{col} = t_j.\text{col}$  then
36:      rgain_tj  $\leftarrow \text{rgain\_tj} - |m_{j,n}|$ 
37:    end if
38:  end if
39: end for
40: return rgain_ti + rgain_tj
```

Algorithm 2 outlines the main steps of the KL heuristic. The while loop in lines 2–18 demonstrates a refinement pass of the KL heuristic. In the KL heuristic, among the task pairs present in each dimension, only the pair with the highest gain is selected, and a swap operation is performed for this pair (lines 6–9). The tasks involved in the swap operation are excluded from further consideration for swaps with other tasks. To represent this, the concept of locking is used. Locked tasks are assigned to the processors with which they were swapped, and their processor assignments remain unchanged throughout the refinement pass. After the locking operation, the task pair with the highest gain is recalculated and selected. This process continues until no more pairs are available for swapping.

Algorithm 2 General Structure of the KL Algorithm for 2-dimensional VPT

```

1:  $gain \leftarrow +\infty$ 
2: while  $0 < gain$  do
3:   Initialize an empty list named GainList to store integer triplets, with a
     capacity of  $K/2$ .
4:   Unlock all task locks.
5:   for  $i \leftarrow 1$  to  $K/2$  do
6:     Compute the maximum swap gain for tasks based on rows.
7:     Compute the maximum swap gain for tasks based on columns.
8:     Determine the maximum of the row-based and column-based swap gains.

9:     Swap the processor assignments of the task pair with the maximum gain
     and lock the tasks.
10:    Add the triplet  $\langle gain, t_i, t_j \rangle$  representing the maximum gain and the task
     pair to GainList.
11:  end for
12:  Compute the maximum prefix sum on GainList.
13:  Assign the maximum prefix sum value to the variable gain.
14:  Revert the processor assignments of task pairs not included in the maximum
     prefix sum to their pre-swap state.
15: end while

```

The swap gain with the maximum value and the corresponding task pair are added to an array called *GainList* as a triplet $\langle gain, t_i, t_j \rangle$ in the order they are obtained (line 10). Due to the locking operation being performed on the tasks in the swap pair (t_i and t_j), at most $\frac{K}{2}$ swap operations can be executed. Therefore,

a maximum of $\frac{K}{2}$ triplet of $\langle gain, ti, tj \rangle$ are calculated. Fewer swaps may be performed in cases where a task is unlocked, but no other unlocked task is available in its row or column. The Prefix-Sum algorithm is applied to the gain values in the *GainList*, and the index corresponding to the maximum cumulative gain in the prefix array is determined (lines 12–13). Swap operations corresponding to pairs after this index are reverted to their pre-swap state (line 14). Although the swap pair with the highest gain in a pass may have a negative value, its inclusion in the gain list gives the KL heuristic a hill-climbing optimization capability. The difference between the total communication volume at the beginning of a pass and the total communication volume after the swaps are realized should be equal to the total gain of the realized swaps. The refinement process continues as long as the total gain remains positive (line 2). If a new pass is initiated, all locks are removed (line 4).

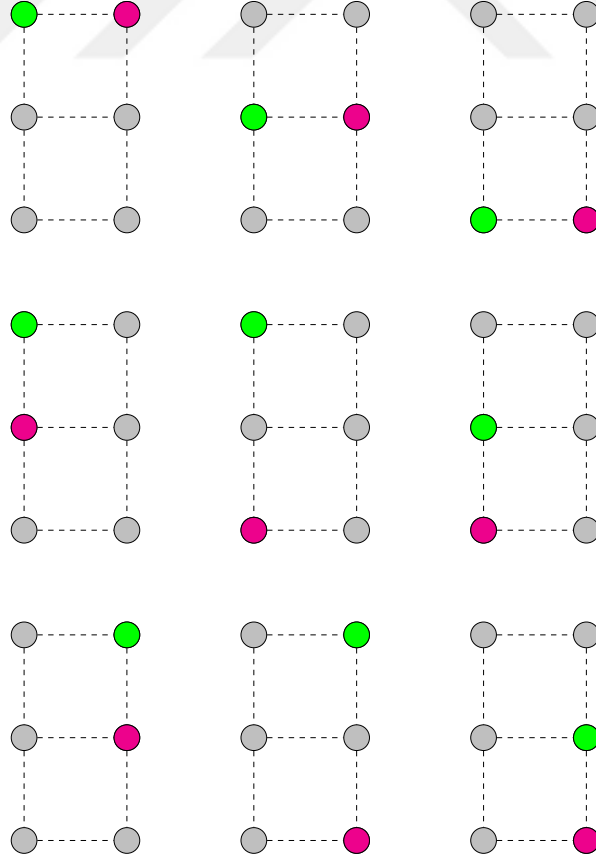


Figure 4.2: Illustration of pairwise task swaps for a 3×2 VPT $T_2(3, 2)$. The first row shows row-wise swaps, and the last two rows show column-wise swaps.

4.3 Generalization to N-Dimensional Topology

The algorithm for n -dimensional VPT builds upon the approach developed for the two-dimensional VPT topology. For n -dimensional topology, the part of Algorithm 1 that requires modification is the conditional checks for detecting gain-inducing neighbors. In the two-dimensional case, gain-inducing positions of communicating tasks are calculated using two specific conditional checks: The first check involved verifying, through an if statement, whether the processors to which the communicating tasks are assigned belong to the same processor group as the swap pair (as shown in Algorithm 1, lines 4, 13, 23, and 32). The condition of being in the same processor group is mathematically expressed for the n -dimensional VPT in Section 2.6.1 as $p_i^d \neq p_j^d$ and $p_i^c = p_j^c$ for all $c \in \{1, \dots, n\}$ such that $c \neq d$ where $d \in \{1, \dots, n\}$. This equation was implemented in the algorithm through an equality check for each dimension d , where d corresponds to the current message passing stage used for pair gain calculation. And the control is performed using the function 'IsInGroupKd' in Algorithm 3. The second check involved verifying, for row-wise swaps, whether the communicating task is in the same column as the processors to which the swapped tasks are assigned, and for column-wise swaps, whether the communicating task is in the same row as the processors to which the swapped tasks are assigned. In the case of a 2-dimensional topology, the gain-inducing communicating task positions lie along these lines. As the topology size increases to 3, these positions shift to lie on the planes. For n -dimensional topology, the gain-inducing communicating task positions lie on the $(n - 1)$ -dimensional hyper-surfaces.

This generalization from 2-dimensional to n -dimensional topology involves an extrapolation of the underlying principles, extending the structure and relationships identified in lower dimensions to higher-dimensional spaces. The computation of hyper-surfaces for an n -dimensional topology requires solving linear equations for each neighbor listed in the Send and Receive sets. Therefore the nodes positioned on these hyper-surfaces can be pre-computed and stored separately for each processor. In this case, it is only required to check whether the tasks in $SendSet(t_i)$ and $ReceiveSet(t_i)$ are assigned to these positions. This eliminates

the need to check the position of tasks that do not contribute to any gain, as they are not part of the hyper-surface. However, for an increasing n -dimension, solving a linear equation representing the $(n - 1)$ -dimensional hyper-surface introduces additional computational overhead. As a result, this approach does change the asymptotic complexity but modifies the constant factor. Alternatively, without loss of generality, a different strategy is used to achieve an optimization-driven simplification, which further enhances the understanding of the improved algorithm given in Section 4.4. As previously discussed, the condition for no gain requires the store-and-forward distance to remain unchanged. The exclusion of these two cases (being within the same processor group and computing the gain only for those where the store-and-forward distance changes) from the swap-gain computation does not compromise the correctness of the algorithm (Figure 4.1 and Equation 4.1). Therefore, the mathematical rigor can be relaxed. This relaxation enables a focused demonstration of how the heuristic and algorithm work across dimensions, even with the reduced rigor.

Algorithm 3 Swap gain calculation for t_i and t_j along dimension d in n -dimensional VPT for the *SendSet*.

Require: $t_i, t_j, \text{SendSet}, \text{curStage}$

```

1:  $\text{rgain} \leftarrow 0$ 
2: for  $t_h \in \text{SendSet}(t_i)$  do
3:   if not IsInGroupKd( $t_h, t_i, \text{curStage}$ ) then
4:      $\Delta D \leftarrow D(\text{map}(t_h), \text{map}(t_i)) - D(\text{map}(t_h), \text{map}(t_j))$ 
5:      $\text{rgain} \leftarrow \text{rgain} + \Delta D * |m_{i,h}|$ 
6:   end if
7: end for
8: for  $t_h \in \text{SendSet}(t_j)$  do
9:   if not IsInGroupKd( $t_h, t_j, \text{curStage}$ ) then
10:     $\Delta D \leftarrow D(\text{map}(t_h), \text{map}(t_j)) - D(\text{map}(t_h), \text{map}(t_i))$ 
11:     $\text{rgain} \leftarrow \text{rgain} + \Delta D * |m_{j,h}|$ 
12:   end if
13: end for
14: return  $\text{rgain}$ 

```

In Algorithm 3, the communication volume gain is computed at each iteration based on changes in the store-and-forward distance of the communicating tasks of t_i or t_j . To simplify notation, all tasks in the communication set of t_i and t_j are

denoted by t_h . This simplification does not change the algorithmic representation or behavior since the gains for t_i and t_j are calculated independently. For each communicating task $t_h \in \text{SendSet}(t_i)$, a conditional check is performed to determine whether it belongs to the same processor group (a scenario where no gain is obtained) (line 3). The D function calculates the store-and-forward distance between two tasks on the VPT. The change in this distance due to the swap between tasks t_h and t_i is assigned to the variable ΔD (line 4). The gain value is determined by multiplying ΔD by the size of $m_{i,h}$, and the running gain is subsequently updated (line 5). The variable curStage indicates the communication dimension by specifying the stage at which the message is sent along dimension d , which is essential for defining neighborhood relationships. The same process is applied to t_j for $t_h \in \text{SendSet}(t_j)$. It should be noted that, the algorithm designed for gain computation on n -dimensional VPT is presented in terms of the SendSet . Similar calculations must also be performed for the ReceiveSet , and the resulting values should be added to the total running gain (rgain) associated with the swap.

Algorithm 4 General Structure of the KL Algorithm for n -dimensional VPT

```

1:  $gain \leftarrow +\infty$ 
2: while  $0 < gain$  do
3:   Initialize an empty list named  $\text{GainList}$  to store integer triplets, with a
     capacity of  $K/2$ .
4:   Unlock all task locks.
5:   for  $i \leftarrow 1$  to  $K/2$  do
6:     for  $d \leftarrow 1$  to  $N$  do
7:       Compute the maximum swap gain for dimension  $d$ .
8:     end for
9:     Compute the maximum swap gain pair obtained across all dimensions.
10:    Swap the processor assignments of the task pair with the maximum gain
      and lock the tasks.
11:    Add the triplet  $\langle gain, t_i, t_j \rangle$  representing the maximum gain and the task
      pair to  $\text{GainList}$ .
12:   end for
13:   Compute the maximum prefix sum on  $\text{GainList}$ .
14:   Assign the maximum prefix sum value to the variable  $gain$ .
15:   Revert the processor assignments of task pairs not included in the maximum
     prefix sum to their pre-swap state.
16: end while

```

The main steps of the KL heuristic for n -dimensional VPT are presented in Algorithm 4. Generalizing Algorithm 2, which limits gain calculations to task pairs in two dimensions (rows and columns), Algorithm 4 extends this computation to all n dimensions using a for loop (lines 6–8). From the computed gains, the pair yielding the maximum swap gain is selected (line 9).

4.4 Improved Algorithm Design

The initial algorithm, which will be referred to as the base algorithm throughout the remainder of this thesis, was designed to calculate the swap gain for each pair and to incorporate conditional checks in order to avoid swaps that would yield no gain in communication volume. As stated in Conclusion 2 at the end of Section 4.1, the computationally intensive components in the calculation of the (t_i, t_j) swap gain are separable and can be computed independently. This feature results in a more efficient algorithm in terms of both time and memory usage, without compromising the correctness of the heuristic, as the number of computations is reduced.

4.4.1 Generalized Formulation for Communication Volumes Incurred by a Task

In Section 4.1, the general formulation for calculating pair gain was derived (Equation 4.7). As mentioned in the second conclusion, the total communication volume incurred by tasks in any given position can be computed independently of the swap pair being considered. The generalized formula for the arithmetic volume calculation, demonstrated previously using the 4×4 mesh example and expressed in Equation 4.3, is as follows:

$$\begin{aligned}
Vol(t_i, map(t_i)) = & \sum_{t_h \in SendSet(t_i)} |m_{i,h}| \cdot D(map(t_i), map(t_h)) \\
& + \sum_{t_h \in ReceiveSet(t_i)} |m_{h,i}| \cdot D(map(t_h), map(t_i))
\end{aligned} \tag{4.8}$$

$$\begin{aligned}
Vol(t_i, map(t_j)) = & \sum_{t_h \in SendSet(t_i)} |m_{i,h}| \cdot D(map(t_j), map(t_h)) \\
& + \sum_{t_h \in ReceiveSet(t_i)} |m_{h,i}| \cdot D(map(t_h), map(t_j))
\end{aligned} \tag{4.9}$$

The communication volume incurred by each message associated with t_i on the VPT is determined by the product of the store-and-forward distance between the tasks involved in sending or receiving the message and the size of the message. To calculate the communication volume incurred by t_i from the position mapped to $map(t_i)$, the communication volumes for all communicating tasks in $SendSet(t_i)$ and $ReceiveSet(t_i)$ must be computed and summed. This value represents the communication volume that would be incurred by t_i from the processor position to which it is mapped.

4.4.2 Optimizing Task Volume Calculations through Post-Processing of the Communication Matrix

Equation 4.8 and 4.9 can be further simplified by manipulating the task-to-task communication matrix. As a reminder, the task-to-task communication matrix M_c is constructed using one-to-one mapping based on 1D non-symmetric (Section 2.2.5) row-wise partitioning of sparse matrices using the column-net hypergraph model [20]. $m_{i,j}$ represents the total message size (in words) sent from t_i (row) to t_j (column) in task-to-task communication matrix, with each individual message having a size of 1 word. By analyzing the M_c , the send and receive sets for each task can be extracted. In this matrix, each row corresponds to the tasks sending messages, while each column represents the tasks receiving messages. As

seen in Equation 4.8, the communication volume incurred by a task is expressed as the summation of the communication volumes for the send and receive sets. The equation can be simplified by utilizing the property of task-to-task communication matrix M_c . Let $SendSet(t_i) \cap ReceiveSet(t_i) = \emptyset$. For the index sets in the two summation functions in Equation 4.8, $CommSet(t_i)$ can be defined as $SendSet(t_i) \cup ReceiveSet(t_i)$. In this case, messages that are present in $SendSet(t_i)$ but not in $ReceiveSet(t_i)$ produce a value of 0 in the summation function for $ReceiveSet(t_i)$, and the same behavior holds for the $ReceiveSet(t_i)$. By utilizing the commutativity and associativity properties of the total summation function, the summand corresponding to the send and receive sets can be combined as follows:

$$Vol(t_i, map(t_i)) = \sum_{t_h \in CommSet(t_i)} \left(|m_{i,h}| \cdot D(map(t_i), map(t_h)) + |m_{h,i}| \cdot D(map(t_h), map(t_i)) \right) \quad (4.10)$$

Due to distributive property of multiplication over addition, the total communication volume of a task, $Vol(t_i, map(t_i))$, can be expressed in an alternative form through equation manipulation, which reduces computational cost and simplifies the implementation as follows:

$$Vol(t_i, map(t_i)) = \sum_{t_h \in CommSet(t_i)} D(map(t_i), map(t_h)) \cdot (|m_{i,h}| + |m_{h,i}|) \quad (4.11)$$

It can be observed that in Equation 4.11, $m_{i,h}$ and $m_{h,i}$ are transposes of each other in the communication matrix. Leveraging this property allows for further simplification of the equation through post-processing of the communication matrix. Since the proposed KL heuristic evaluates gain based on the combined send and receive volumes, maintaining the communication matrix in a form that directly represents the total message sizes (send + receive) facilitates a more efficient and computationally cost-effective calculation. The combined (send + receive) task-to-task communication matrix is defined as $M_{sr} = M_c + M_c^T$. Consequently, $m_{i,h}$ value in M_{sr} represents the message size for both send and receive operations. In other words, M_{sr} has the property of being symmetric. Consequently, $|m_{i,h}| = |m_{h,i}|$. Due to the store-and-forward communication scheme, the hop

distance between t_i and t_j is the same as the hop distance between t_j and t_i , even if their routes differ in some cases. Therefore, the communication volume incurred by t_i based on M_{sr} can be calculated as $\sum_{t_h \in \text{SendSet}(t_i)} |m_{i,h}| \cdot D(\text{map}(t_i), \text{map}(t_h))$.

4.4.3 Extended Notation and Terminology for the Improved Algorithm

Processors within the same processor group are referred to as neighboring processors. Tasks assigned to neighboring processors can also be described as neighboring tasks. However, the improved algorithm is formulated based on the complete set of neighboring processors associated with each task. The transitions between processors and tasks add complexity to both the explanation and implementation of the algorithm. To maintain clarity in the explanation, the term “adjacent” will be used specifically to refer to neighboring processors in the context of the improved algorithm. In other words, operations related to the set of neighboring processors in the improved algorithm will be denoted using the term “adjacent” (abbreviated as “Adj”). Processors within the same processor group are defined as adjacent processors. The notation $\text{Adj}(\text{map}(t_i))$ will be used to represent the set of processors adjacent to the processor to which task t_i is mapped in the VPT.

In Figure 4.3, the adjacent processors of processor 1 (p_1) are illustrated on a 3-dimensional VPT, $T_3(3,3,3)$. The adjacent processors are highlighted with different colors red, blue, and orange to represent different communication stages. And the processors are highlighted with a black gradient applied in the depth-wise direction to improve visualization. For the example given in Figure 4.3, let task t_i be mapped to p_1 . In this case, $\text{Adj}(\text{map}(t_i)) = \{2, 3, 10, 19, 4, 7\}$.

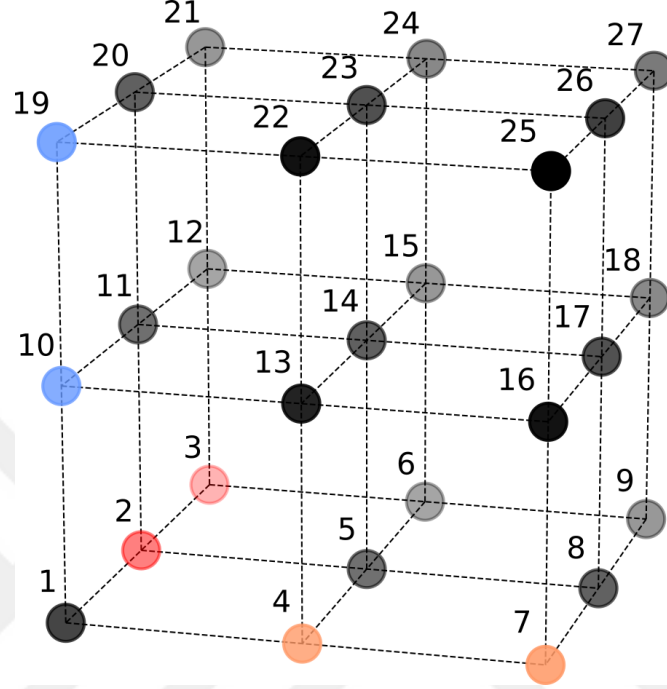


Figure 4.3: 3-dimensional VPT $T_3(3,3,3)$. Task t_i is mapped to p_1 , and its adjacent processors $Adj(map(t_i))$, are highlighted using red, blue, and orange colors.

4.4.4 Pre-computation of Communication Volumes for Potential Swap Positions

In the improved algorithm, the volume components defined in Equation 4.7 are computed and stored at the beginning of each pass for every swap pair. In this way, components that are repeatedly recomputed after each lock in the base algorithm are now computed only once. These components were previously expressed using the notation $Vol(t_i, map(t_j))$, representing the communication volume incurred by task t_i from the processor position to which task t_j is mapped in the VPT. During a pass, as pairwise tasks (t_i, t_j) are locked as part of swap operations, the precomputed volumes are updated for all tasks in $CommSet(t_i)$ and $CommSet(t_j)$. The precomputation step is described as follows: Before each pass, the processor mapped to a given task t_i in VPT is identified. Based on this identified processor, the set of tasks with which t_i can be swapped is determined. The communication volume incurred by t_i from the positions of its swap

candidates is then calculated. This process is repeated for all tasks.

Algorithm 5 presents the steps involved in calculating and storing the communication volumes incurred at potential swap positions for each task assigned to the VPT through a one-to-one mapping. The algorithm takes as input the array *AdjCommVolumes* and the set *CommSet*. The set *CommSet* represents the communication matrix M_{sr} , which is obtained through a post-processing step. This matrix is stored in Compressed Sparse Row (CSR) format. The messages sizes are denoted by $m_{i,h}$. The array *AdjCommVolumes* is defined as a one-dimensional array of size $K \cdot (|Adj(map(t_i))| + 1)$, and is initialized with zero at the beginning of each pass. The term $|Adj(map(t_i))|$ denotes the number of adjacent processors to a given processor, and the corresponding entries in the array are used to store the values of $Vol(t_i, map(t_j))$. The term '+1' in the size expression accounts for the value $Vol(t_i, map(t_i))$, which represents the communication volume when t_i remains at its current position.

Algorithm 5 Pre-computation of communication volumes incurred by each task from all potential swap positions in the VPT.

Require: *CommSet*, *AdjCommVolumes*

```

1: for  $t_i \in \{t_1, t_2, \dots, t_K\}$  do
2:    $p_k \leftarrow \text{map}(t_i)$ 
3:    $offset \leftarrow 1$ 
4:   for  $p_l \in Adj(p_k) \cup \{p_k\}$  do
5:      $t_j \leftarrow \text{imap}(p_l)$ 
6:      $\text{SWAP}(t_i, t_j)$ 
7:      $totalCommVol \leftarrow 0$ 
8:     for  $t_h \in \text{CommSet}(t_i)$  do
9:        $p_m \leftarrow \text{map}(t_h)$ 
10:       $commVolume \leftarrow |m_{i,h}| * D(p_l, p_m)$ 
11:       $totalCommVol \leftarrow totalCommVol + commVolume$ 
12:    end for
13:     $AdjCommVolumes[p_k * |Adj(p_k) \cup \{p_k\}| + offset] \leftarrow totalCommVol$ 
14:     $offset \leftarrow offset + 1$ 
15:     $\text{SWAP}(t_i, t_j)$ 
16:  end for
17: end for
```

Algorithm 5 consists of three nested loops. The outer loop iterates over all the tasks $\mathcal{T} = \{t_1, t_2, \dots, t_K\}$ mapped on VPT. The processor to which task t_i is assigned on the VPT is assigned to the variable p_k (line 2). The middle loop iterates over the set of adjacent processors p_l of a given task t_i which is mapped to p_k . To calculate the communication volume incurred by task t_i from its assigned position, denoted as $Vol(t_i, map(t_i))$, the processor p_k is also included in this set. The task assigned to processor p_l is assigned to the variable t_j (line 5). The term inverse mapping (imap) is used to denote the mapping from processor to task. Task t_j corresponds to an adjacent task of $map(t_i)$ that can be swapped. The swap operation is temporarily performed for the calculation of the communication volume incurred by t_i from the position of t_j (line 6). Subsequently, the (t_i, t_j) swap is rolled back to its original state (line 15). In the case where p_l is equal to p_k (i.e., when they are the same processor), t_i will be swapped with itself. The necessary control statements for this situation are not included due to the vectorization and parallelization optimizations. The variable that will store the value of $Vol(t_i, map(t_j))$ is initialized to 0 in Line 7.

The innermost loop iterates over the communicating tasks of t_i . The processor to which the communicating task t_h is assigned is stored in the variable p_m for the store-and-forward distance calculation on the VPT (line 9). In other words, each iteration in the innermost loop corresponds to the case where t_i (p_k) sends a message to t_h (p_m) from the position of t_j (p_l). The total message size $|m_{i,h}|$ sent and received between t_i and t_h is multiplied by the store-and-forward distance $D(map(t_i), map(t_h))$ to calculate the communication volume $Vol(t_i, map(t_h))$ (line 10). This value is then added to the variable *totalCommVol*, which temporarily stores the communication volume that t_i will incur from the position $map(t_j)$ (line 11). The value of *totalCommVol* is assigned to its position in the array that stores all communication volumes (line 13). It corresponds to the value returned by $Vol(t_i, map(t_j))$, and its position in the array is calculated with $AdjCommVolumes[p_k \cdot |Adj(p_k) \cup \{p_k\}| + offset]$ as follows: The number of adjacent processors is the same for all processors. Therefore, the index value is computed using pointer arithmetic, as applied when flattening a two-dimensional array into a one-dimensional array. Each inner array has a

size of $|Adj(p_k) \cup p_k|$. The index corresponds to the element at the offset position in the p_k -th row. Therefore, the offset variable is used to count the number of adjacent processors for array indexing (line 3, 14).

4.4.5 Updating Communication Volumes After Task Swaps

The swap operation is performed after selecting the swap pair that yields the maximum gain. Therefore, after the swap, the communication volumes incurred by tasks other than the swap pair can only change if they communicate with t_i or t_j . In other words, the pre-calculated communication volumes that need to be updated will only be performed for the tasks in $CommSet(t_i)$ and $CommSet(t_j)$. Algorithm 6 represents this update process.

The task t_i communicates with each $t_h \in CommSet(t_i)$. As t_i displace, the store-and-forward distance $D(map(t_i), map(t_h))$ may change. In this case, the communication volumes incurred by each task $t_h \in CommSet(t_i)$ from the processor $map(t_h)$ and its adjacent processors $Adj(map(t_h))$ are expected to change as a result of changes to the store-and-forward distance between t_i and t_h . Notice that the communication volumes incurred by each processor from its assigned position and adjacent positions were calculated. As a result, updates are required to be performed for all potential swap positions. The update is performed before realizing the swap operation, although it may also be carried out afterward. In such a case, the signs of the terms in the equations on Line 9 are reversed. However, performing the update before the swap simplifies the construction of the algorithm. The update operation follows the approach of first subtracting the communication volume incurred for $|m_{i,h}|$ at the previous position of the swap, and then adding the communication volume incurred for $|m_{i,h}|$ at the new position. As a result, the change in communication volume, denoted as ΔVol , will be equal to the product of the change in the store-and-forward distance ΔD , and $|m_{i,h}|$ (lines 9, 10). t_i and t_j are initially mapped to the processor positions prior

to the swap (line 1, 2). The variable s stores the total number of adjacent processors including a processor itself (line 3) and corresponds to $|Adj(p_k) \cup \{p_k\}|$, which is discussed in detail for line 13 in Algorithm 5. The calculated ΔVol value will be added to the communication volume incurred from the positions of processor $p_n \in Adj(map(t_h)) \cup \{map(t_h)\}$ (line 13). All updates are performed for the tasks in the communication sets of t_i and t_j (lines 4–16, 17–29). The operations performed for t_i are repeated for t_j .

Algorithm 6 Updates the communication volumes incurred by the tasks in $CommSet(t_i)$ and $CommSet(t_j)$.

Require: $CommSet$, $AdjCommVolumes$, t_i , t_j

```

1:  $p_k \leftarrow map(t_i)$ 
2:  $p_l \leftarrow map(t_j)$ 
3:  $s \leftarrow |Adj(p_k) \cup \{p_k\}| = |Adj(p_l) \cup \{p_l\}|$ 
4: for  $t_h \in CommSet(t_i)$  do
5:    $offset \leftarrow 1$ 
6:    $p_m \leftarrow map(t_h)$ 
7:   for  $p_n \in Adj(p_m) \cup \{p_m\}$  do
8:     if  $p_n \neq p_l$  then
9:        $\Delta D \leftarrow D(p_k, p_n) - D(p_l, p_n)$ 
10:       $\Delta Vol \leftarrow |m_{i,h}| * \Delta D$ 
11:    end if
12:     $idx \leftarrow p_n * s + offset$ 
13:     $AdjCommVolumes[idx] \leftarrow AdjCommVolumes[idx] + \Delta Vol$ 
14:     $offset \leftarrow offset + 1$ 
15:  end for
16: end for
17: for  $t_h \in CommSet(t_j)$  do
18:    $offset \leftarrow 1$ 
19:    $p_m \leftarrow map(t_h)$ 
20:   for  $p_n \in Adj(p_m) \cup \{p_m\}$  do
21:     if  $p_n \neq p_k$  then
22:        $\Delta D \leftarrow D(p_l, p_n) - D(p_k, p_n)$ 
23:        $\Delta Vol \leftarrow |m_{j,h}| * \Delta D$ 
24:     end if
25:      $idx \leftarrow p_n * s + offset$ 
26:      $AdjCommVolumes[idx] \leftarrow AdjCommVolumes[idx] + \Delta Vol$ 
27:      $offset \leftarrow offset + 1$ 
28:   end for
29: end for

```

4.4.6 General Structure of the Improved Algorithm

Algorithm 7 General Structure of the KL Algorithm for n -dimensional VPT

```

1:  $gain \leftarrow +\infty$ 
2: Initialize an empty communication volumes array named  $AdjCommVolumes$  of
   size  $K \cdot \left( \sum_{d=1}^K (k_d) - N + 1 \right)$ .
3: while  $0 < gain$  do
4:   Unlock all task locks.
5:   Initialize an empty list named  $GainList$  to store integer triplets, with a
     capacity of  $K/2$ .
6:   Set all elements of  $AdjCommVolumes$  to zero.
7:   Calculate and store adjacent comm. volumes in the  $AdjCommVolumes$ .
8:   for  $i \leftarrow 1$  to  $K/2$  do
9:     Compute the maximum swap gain pair obtained across all dimensions.
10:    Update adjacent comm. values in the  $AdjCommVolumes$  for  $t_i$  and  $t_j$ .
11:    Swap the processor assignments of the task pair with the maximum gain
      and lock the tasks.
12:    Add the triplet  $\langle gain, t_i, t_j \rangle$  representing the maximum gain and the task
      pair to  $GainList$ .
13:   end for
14:   Compute the maximum prefix sum on  $GainList$ .
15:   Assign the maximum prefix sum value to the variable  $gain$ .
16:   Revert the processor assignments of task pairs not included in the maximum
     prefix sum to their pre-swap state.
17: end while

```

The main steps of the improved KL heuristic algorithm for the n -dimensional VPT are presented in Algorithm 7. Algorithm 7 maintains the same overall structure as Algorithm 4. In Algorithm 4, the swap gains are recalculated in each iteration. However, Algorithm 7 includes additional steps related to pre-computation and update. These additional steps are as follows: An empty array is initialized to store the communication volume incurred by each task from its assigned positions (line 2). At the beginning of each pass, all elements of this array are set to zero (line 6). Then, the communication volumes are calculated using Algorithm 5 and stored in the $AdjCommVolumes$ array (line 7). Inside the inner loop, due to the position changes of swap pair (t_i, t_j) , the communication volume values incurred by other tasks that communicate with t_i or t_j are affected.

These values are updated for the new positions of t_i and t_j using Algorithm 6, before the swap is applied (line 10).

4.4.7 Complexity Analysis

The running time of a single pass of the KL heuristic is analyzed for Algorithm 3 and Algorithm 4.

4.4.7.1 Computation of the total number of swap pairs

In n -dimensional VPT, at most $K/2$ swap operations are performed per pass (Algorithm 4, line 5). For the initial pass where all tasks are unlocked, the total number of pairwise combinations that need to be calculated for a k -ary n -cube like VPT (Algorithm 4, line 6–8) can be defined as:

$$n \cdot \frac{K}{k} \cdot \binom{k}{2} = \frac{n \cdot k^n \cdot (k-1)}{2} \quad (4.12)$$

It should be noted that K is assumed to be a power of 2. The binomial term represents the number of pairwise combinations within a processor group. Here, $k = K^{\frac{1}{n}}$ indicates the processor group size (k_d), $\frac{K}{k}$ denotes the total number of processor groups for k -ary n -cube like VPT. And n is the number of dimensions.

For the n -dimensional VPT, where the sizes of individual dimensions $k_1, k_2, \dots, k_n$ (denoted as k_d for each dimension d) may vary, the total number of processors K is given by $\prod_{d=1}^n k_d = K = 2^m$, where $m \in \mathbb{N}^+$. In such a VPT with non-equal dimension sizes, the total number of pairwise combinations across all dimensions is computed by the following expression, where the summation is taken over all dimensions.

$$\sum_{d=1}^n \left(\frac{K}{k_d} \cdot \binom{k_d}{2} \right) \quad (4.13)$$

In the equation, $\frac{K}{k_d}$ represents the number of processor groups in each dimension, and $\binom{k_d}{2}$ represents the number of pairwise combinations within each processor group.

The initial total number of swap pairs can be determined based on the number of adjacent processors in the VPT. Each task's assigned processor has a number of adjacent processors equivalent to the difference between the sum of the sizes of the VPT dimensions and the total number of dimensions (see Figure 4.3 for a detailed illustration). Each task is involved in a total of $\sum_{d=1}^K (k_d) - N$ swaps. The total number of pairwise combinations can be expressed as follows, and this alternative formulation will be used in the complexity analysis.

$$K \cdot \left(\sum_{d=1}^K (k_d) - N \right) \cdot \frac{1}{2} \quad (4.14)$$

When counting task pairs based on the number of neighbors, both (t_i, t_j) pair from the perspective of t_i and (t_j, t_i) pair from the perspective of t_j are included. As a result, each pair (t_i, t_j) is counted twice. Therefore, the equation must be multiplied by $\frac{1}{2}$.

4.4.7.2 Complexity analysis of the initial swap in a pass

In order to analyze Algorithm 3, the cardinalities of $SendSet(t_i)$, $SendSet(t_j)$, $ReceiveSet(t_i)$, and $ReceiveSet(t_j)$ must be considered. The task-to-task communication matrix used varies in density or sparsity depending on the sparse matrix partitioning applied and the partitioning process itself. In this case, the number of non-zero entries in the task-to-task communication matrix, specifically through Send or Receive communication, provides a more comprehensive analysis. For the initial pass where all tasks are unlocked, the total number of pairwise combinations is given as shown in Equation 4.14. For each processor, the number of swaps equals the number of adjacent processors. In this initial pass, the swap gain is calculated for all tasks by iterating over the entirety of $SendSet(t_i)$ for each task. This iteration is performed a total of $\sum_{d=1}^K (k_d) - N$ times. As a result, the entire communication matrix is traversed. This traversal corresponds to each non-zero (nnz) entry within the communication matrix. Therefore, the overall complexity of the process until the initial locking operation is expressed as $O((\sum_{d=1}^K (k_d) - N) \cdot nnz)$.

4.4.7.3 Estimation of the total number of swap pairs over a pass

During a single pass, the number of swap pairs is algorithmically counted at each locking step, rather than being computed using a mathematical formula. Figure 4.4 illustrates the number of remaining swap pairs after each locking operation for a 2-dimensional $\sqrt{K} \times \sqrt{K}$ VPT. Each subplot corresponds to a different value of $K \in \{16, 64, 256, 1024\}$. The x-axis in each subplot represents the index of the locking operation, while the y-axis indicates the number of swap pairs that remain unlocked at each step.

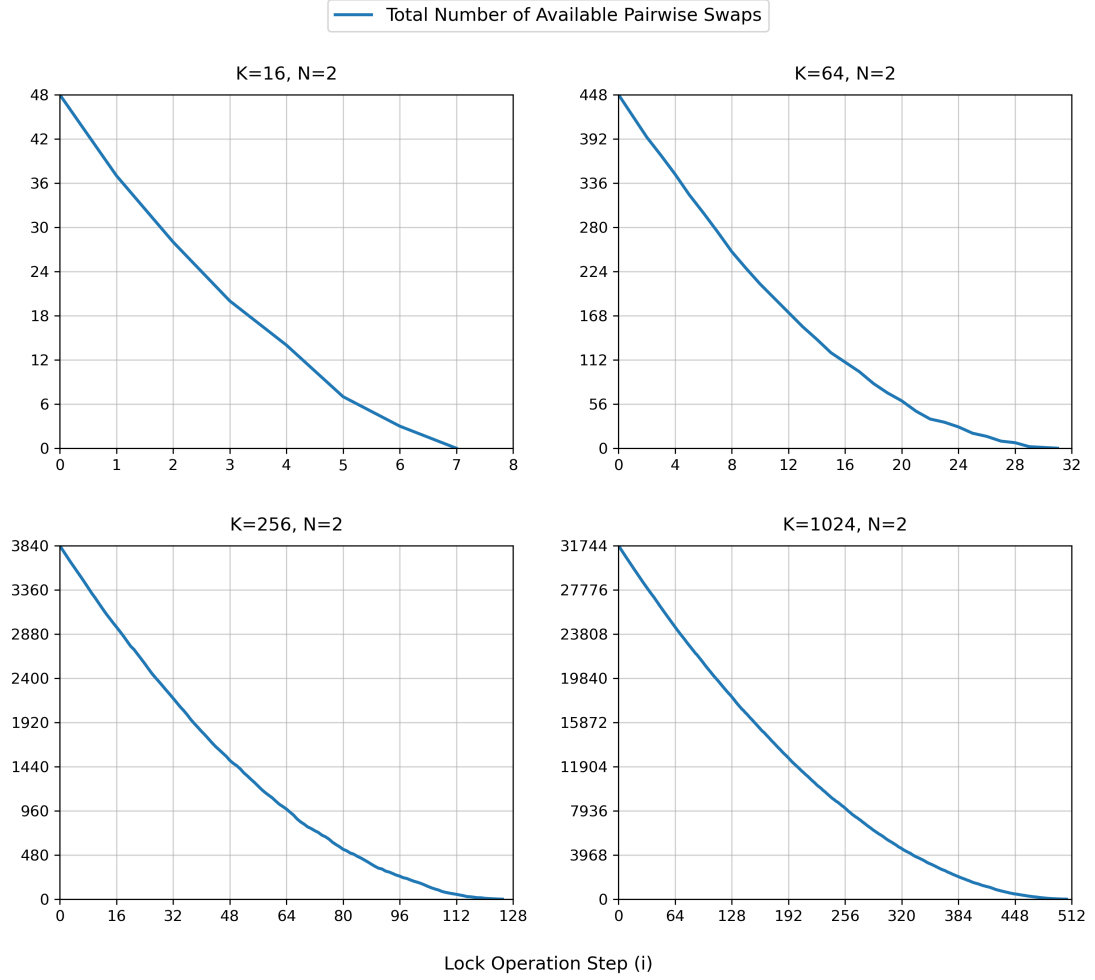


Figure 4.4: Plot of the remaining number of available pairwise swaps after each locking operation throughout a single pass for different values of K in 2-Dimensional VPTs.

The step i_0 represents the initial state, where all locks are open and the total number of swap pairs is at its maximum. The rightmost position on the x-axis corresponds to the theoretical upper bound on the number of possible locking operations, which is at most $K/2$. The step $i_{K/2}$ represents the final state, where the count is taken after the last locking operation. Additionally, if the remaining tasks are not positioned on the same processor group, no swap operation can be performed. As a result, in some cases such as $K = 16$, $N = 2$ in Figure 4.4, the iteration process terminates before reaching the maximum step count of $K/2$.

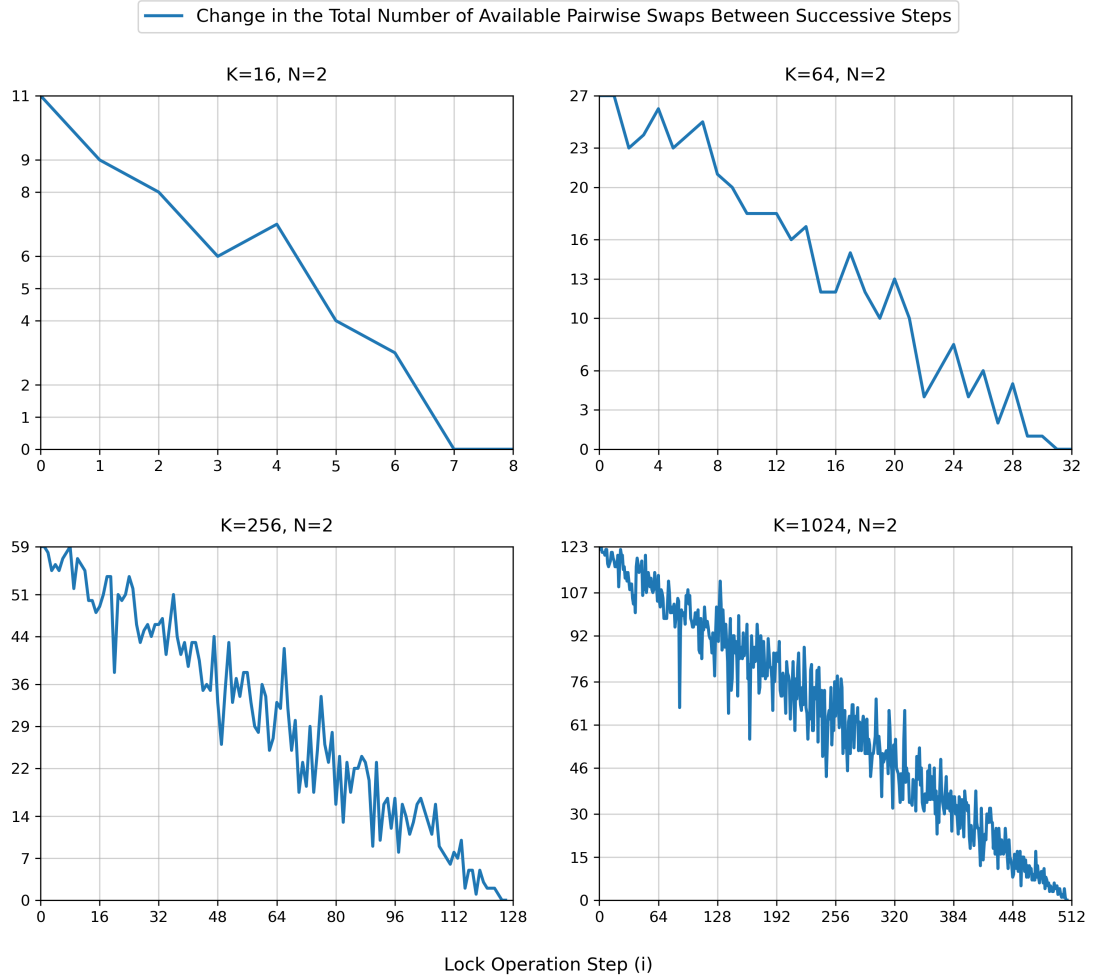


Figure 4.5: Plot of the change in the total number of available pairwise swaps after each locking operation throughout a single pass for different values of K in $\sqrt{K} \times \sqrt{K}$ 2-dimensional VPTs.

The subplots of the total number of swap pairs fit a quadratic function. At each step, the number of swap pairs decreases, with the rate of decrease remaining stable. This reduction follows a pattern fits a linear function, $y = mx + b$ where $m < 0$, since the derivative of a quadratic function would yield a linear decrease. However, the decrease occurs with oscillations around this linear function as shown in Figure 4.5. The plots show how the total number of available pairwise swaps changes between each locking step.

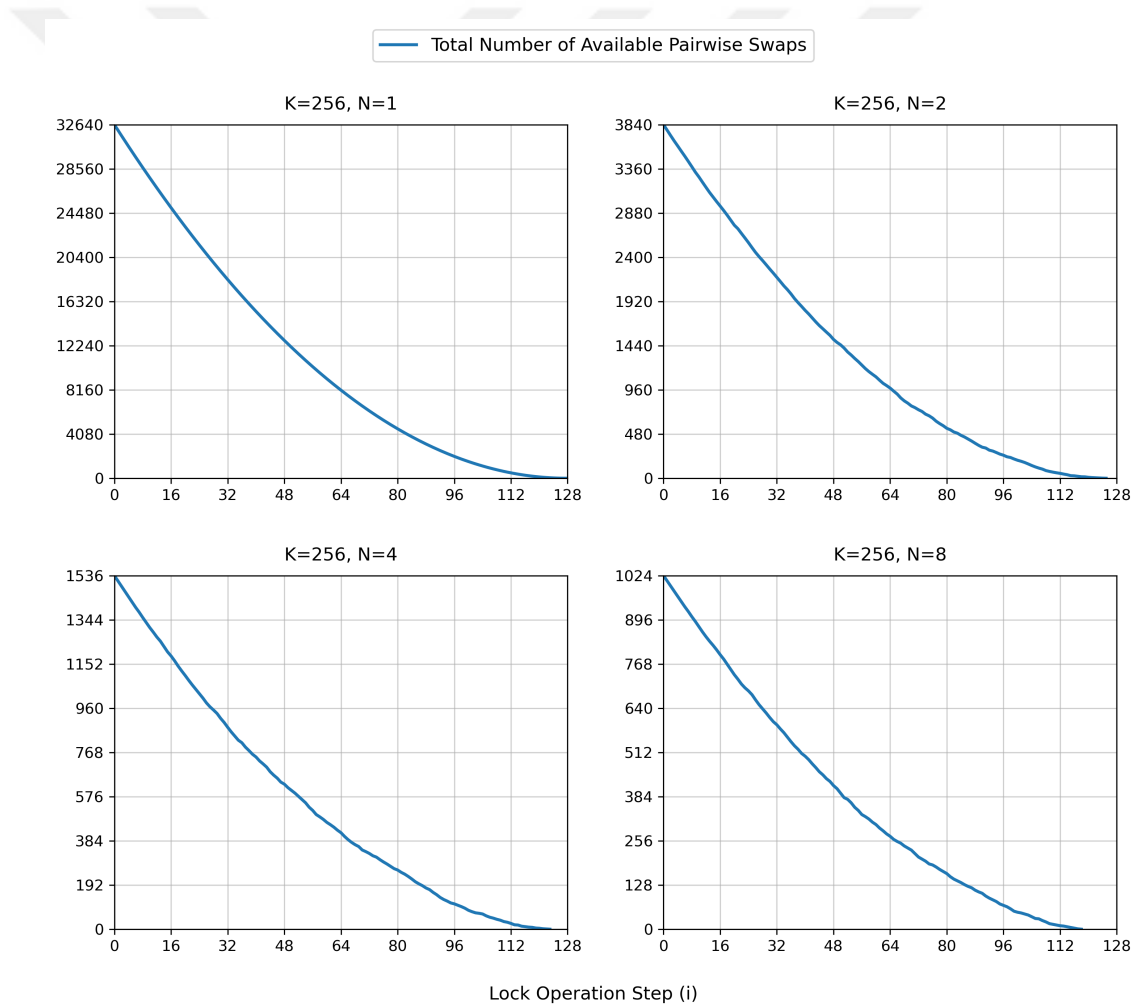


Figure 4.6: Plot of the remaining number of available pairwise swaps after each locking operation throughout a single pass for different N-dimensional VPTs with fixed K .

Furthermore, the area under the quadratic curve and the area under the linear function defined by the maximum values on the x-axes and y-axes appear to be relatively close. Based on this observation, rather than deriving a separate quadratic formula for each value of K , a simplified general formula for calculating the total number of swap pairs throughout a single pass can be approximated by calculating the area under the linear function, which is obtained by multiplying the initial number of swap pairs at i_0 by $K/2$ and dividing by 2. This approximation closely matches the observed behavior across all values of K .

In Figure 4.6, the total number of available pairwise swaps is plotted for $K = 256$ across a range of dimensionalities of VPTs: 1D (base case), 2D (low), 4D (medium), and $\log_2 K$ (i.e., 8D, maximum), to demonstrate its behavior. As shown in the figure, the swap count fits a quadratic function in all four cases. In the 1D VPT (base case), exactly $K/2$ swaps can be performed, since all tasks are placed on adjacent processors. When all locks are open, the total number of available pairwise swaps corresponds to the binomial coefficient $\binom{n}{2}$, where n is equal to K . After each locking operation, the total number of tasks reduces by 2, resulting in a new problem size of $(n - 2)$, and thus the swap count becomes $\binom{n-2}{2}$. For this reason, no oscillation is observed in the 1D VPT case.

4.4.7.4 Estimation of the number of conditional checks per swap pair

The developed KL heuristic aims to reduce communication volume by positioning tasks with high communication volume closer to each other on the VPT through task swapping. This swapping operation occurs within the same processor group (i.e., adjacent processors). The relocation of tasks t_i and t_j on the VPT affects the store-and-forward distances to their communicating tasks t_h , thereby potentially changing their contributions to the total communication volume. Consequently, the communication volume may increase, decrease, or remain unchanged. The swap gain for tasks t_i and t_j is determined by the accumulation of the communication volume differences resulting from the swap of these tasks.

Recall that if t_h belongs to the same processor group as the swap pair (t_i, t_j) ,

the store-and-forward distance to t_h will not change, and as a result, no gain will be achieved. In the design of the Algorithm 1, this was controlled using a conditional check. The cost of performing these checks is a crucial factor in the overall complexity of the algorithm, especially when considering the number of possible swaps. In an n -dimensional VPT, $n - 1$ conditional checks are required for each communicating task t_h . The total number of potential swap pairs in a k -ary n -cube like VPT topology is given by $(n \cdot k^n \cdot (k - 1))/2$, and for each swap pair, the number of conditional checks required is the sum of the total number of communicating tasks of t_i and t_j , multiplied by $(n - 1)$. As mentioned previously, SpMV and SpMM operations are partitioned using PaToH, and a one-to-one mapping is applied either randomly or by a smart mapping heuristic. The probability that a communicating task (t_h) is assigned to the same processor group as the swap pair (t_i, t_j) reaches its maximum value, $P_{\text{best}} = 1$, only in the base case, where a one-dimensional VPT is used. In this case, all tasks are neighbors on the VPT, so every neighboring tasks are in the same processor group. For VPTs with more than one dimension, this probability becomes lower. The likelihood that task t_h is in the same processor group as the swap pair (t_i, t_j) depends on the number of other tasks in the group. In this case, the worst-case probability is given by $P_{\text{worst}} = \frac{k_d - 2}{K}$. The probability that t_h is not in the same group is then $\frac{K - (k_d - 2)}{K}$. Since $\frac{k_d - 2}{K} \ll \frac{K - (k_d - 2)}{K} < 1$, in practice, the probability of executing the $(n - 1)$ conditional checks for task t_h becomes close to one.

4.5 Vectorization and Parallelization of the KL-based Heuristic

Let (t_i, t_j) represent a swap pair, and let t_h denote a communicating task that belongs to the communication set of t_i or t_j , such as $t_h \in \text{SendSet}(t_i) \cup \text{ReceiveSet}(t_i)$ or $t_h \in \text{SendSet}(t_j) \cup \text{ReceiveSet}(t_j)$. Recall that in Section 4.2, the gain calculation is performed for communicating pairs (t_i, t_h) and (t_j, t_h) , and this calculation is performed only if the store-and-forward distance between (t_i, t_h) or (t_j, t_h) changes as a result of the swap between (t_i, t_j) . In Section

4.3, the algorithm from Section 4.2 is generalized to n -dimensions by removing this conditional check without compromising the correctness of the algorithm, and modifying the condition to check whether tasks are in the same processor group. As a result, communicating pairs with a gain of zero are included in the calculations, regardless of whether they contribute to the gain. This modification fundamentally changes the algorithm, and it corresponds to the difference in communication volume incurred by the tasks t_i and t_j . In Section 4.4, the communication volume incurred by each task from their potential swap positions is computed and stored. This computation occurs only once at the beginning of each pass. The swap gain is calculated based on the difference between these volumes. When tasks are swapped, the communication volumes affected by the swap are updated to reflect the changes in position.

In both Sections 4.3 and 4.4, the swap gain is computed based on differences in communication volume. In Section 4.3, communication volumes are recalculated for each swap pair gain calculation, whereas in Section 4.4, they are computed only once at the beginning of each pass. This approach, used in the improved algorithm, reduces the asymptotic time complexity. The complexity of this calculation in Algorithm 5 is the product of the following factors: the total number of processors, the number of neighboring processors plus one, and the number of nonzero elements in the sparse matrix.

In Algorithm 5, the communication volume calculation approach enables vectorization, and overall design enables a high degree of parallelization. The communication volume incurred by a task in the improved algorithm is calculated as described in Equation 4.11. This equation can be expressed as a vectorized operation using an element-wise product $w * d$, or alternatively, using the dot product $w^T d$, where w is a vector of weights (i.e., message sizes, $m_{i,h}$), and d is a vector representing the store-and-forward distances between t_i and t_h . The communication volume calculation for each task, which constitutes a significant portion of the computation, can be accelerated using SIMD instructions on CPUs. These calculations can be performed independently for each task’s potential swap positions, as there are no data dependencies between them, enabling an embarrassingly parallel design. Consequently, the combination of an embarrassingly

parallel design and SIMD-accelerated communication volume calculation leads to significant performance improvements. Furthermore, the dot product $w^T d$, which involves element-wise multiplication followed by reduction, is ideal for GPU computation, particularly in large-scale workloads.



Chapter 5

Experiments

5.1 Data Set

The proposed method for reducing the increased communication volume incurred by store-and-forward scheme was evaluated on 12 latency-bound, square sparse symmetric matrices with irregular communication patterns. In the experiments, the same matrices selected in the study by Selvitopi and Aykanat [2] were used. These matrices were selected from the SuiteSparse Matrix Collection [1] and have diverse structural characteristics. Their properties are presented in Table 5.1.

In Table 5.1, the column labeled 'nrows' represents the total number of rows, 'nnz' represents the number of nonzero elements, and 'dens' represents the density, i.e., fill ratio or the proportion of nonzero entries in the sparse matrix. The maximum degree (maxd) is the number of nonzero elements in the row with the highest number of nonzero counts. This value defines the degree of that row. The maximum degree ratio (maxdr) is the ratio of maxd to the total number of rows and represents the possibility of observing high density row in the matrix. The coefficient of variation (cv) is the ratio of the standard deviation to the mean, also known as the normalized standard deviation. This metric captures the variability of the number of nonzero elements per row across the matrix. A high cv value

indicates that the number of nonzero elements varies significantly between rows. Therefore, the coefficient of variation is used as an indicator of how irregular the matrix is. For example, consider a 5 by 5 matrix where four rows are very sparse (each containing one nonzero element) and one row is extremely dense (containing five nonzero elements). In this case, the maximum degree is 5, the maximum degree ratio is $5/5 = 1$. And the mean value of nonzeros per row is 1.8, the standard deviation is 1.6, and the coefficient of variation is 0.89.

Table 5.1: Properties of the square sparse matrices used in the experiments.

Matrix Name	Kind	nrows	nnz	dens	max	cv	maxdr
F1	structural eng.	343 791	26 873 113	0.023	435	0.52	0.001
mip1	optimization	66 463	10 352 819	0.234	66395	2.25	0.999
fe_rotor	undirected graph	99 617	1 324 862	0.013	125	0.29	0.001
TSOPF_FS_b300_c3	power network	84 414	13 135 930	0.184	41542	7.59	0.492
cbuckle	structural mech.	13 681	676 515	0.361	600	0.16	0.044
bundle_adj	comp. vision prob.	512 351	20 208 051	0.008	12588	6.37	0.025
pattern1	optimization	19 242	9 323 432	2.518	6028	0.78	0.313
GaAsH6	chemistry problem	61 349	3 381 809	0.090	1646	2.44	0.027
SiO2	chemistry problem	155 331	11 283 503	0.047	2749	4.05	0.018
coAuthorsDBLP	co-author network	299 067	1 955 352	0.002	336	1.50	0.001
nd24k	2D/3D problem	72 000	28 715 634	0.554	520	0.19	0.007
human_gene2	gene network	14 340	18 068 388	8.787	7229	1.09	0.504

5.2 Experimental Setup

The experiments were conducted to evaluate the proposed iterative refinement heuristic under a one-to-one task-to-processor mapping scheme, where each task is assigned to a unique processor. The sparse matrices were partitioned using a K-way partitioning approach based on the column-net hypergraph model. The

PaToH hypergraph partitioning tool [20] was used for this purpose. Communication between one-to-one mapped tasks was performed using the row-parallel algorithm for both SpMV and SpMM operations. In this algorithm, processors first exchange the required input vector elements via point-to-point (P2P) messages. After receiving the necessary data, each processor performs its local computations independently.

The communication model, in which all messages are sent P2P and each message reaches its destination in a single hop, is referred to as the Baseline (BL) scheme. The Baseline corresponds to the case where the number of dimensions of VPT, denoted as n , is 1. When n is greater than 1, communication is carried out using a store-and-forward scheme. And, the performance of the proposed heuristic on communication was evaluated using two different metrics: Total Communication Volume (TCV) and Maximum Weighted Communication Volume (MWCV). The Total Communication Volume corresponds to the sum of all words sent by all processors across each dimension (i.e., stage) of the n -dimensional VPT. More generally, it is defined as the total amount of data sent between processors during the communication phase, and it is expressed as follows:

$$\text{TCV} = \sum_{t_i \in T} \left(\sum_{t_h \in \text{SendSet}(t_i)} |m_{i,h}| \cdot D(\text{map}(t_i), \text{map}(t_h)) \right) \quad (5.1)$$

The Weighted Communication Volume (WCV) is defined as the total number of words sent and received via P2P communication by any single processor in the store-and-forward scheme on an n -dimensional VPT. In other words, the Weighted Communication Volume of a processor corresponds to the total number of words it sends and receives in each stage of the n -dimensional VPT. The WCV for a processor is calculated as follows:

$$\text{WCV}(t_i) = \sum_{d=1}^N (\text{StageSendSize}(t_i, d) + \text{StageReceiveSize}(t_i, d)) \cdot 1 \quad (5.2)$$

In Equation 5.2, $\text{StageSendSize}(t_i, d)$ ve $\text{StageReceiveSize}(t_i, d)$ denote the total message size sent and received, respectively, by task t_i at stage d . Since the store-and-forward distance is 1 for each message sent or received at every stage,

the multiplication factor is explicitly given as 1 in the equation. In the calculation of Total Communication Volume, the specific route a message follows is not important since the total number of hops the message must travel remains the same. However, the route taken by the message is significant for the calculation of Weighted Communication Volume as the ordering of communication stages in the store-and-forward scheme determines which intermediate processors will store the messages. The KL-based heuristic algorithm does not consider the direction of message transmission or the message routes. While refining the mapping based on Total Communication Volume, the algorithm may exhibit a bias toward one of the two possible stage directions (i.e., dimension order from left to right or right to left). However, this direction influences the calculation of Weighted Communication Volume. Therefore, when reporting experimental improvements, the dimension order direction is selected to favor the lowest Weighted Communication Volume yielded by the initial mapping algorithm. The Maximum Weighted Communication Volume is defined as the highest Weighted Communication Volume among all tasks. It is expressed as follows:

$$\text{MWCV} = \max_{t_i \in T} (\text{WCV}(t_i)) \quad (5.3)$$

Overall, TCV is used for estimating overall network load. MWCV is used for estimating communication bottleneck caused by the most loaded processor, which can increase latency and limit scalability.

5.3 Experimental Results and Analysis

The figures show bar charts comparing the performance of the KL-based heuristic method in terms of Total Communication Volume and Maximum Weighted Communication Volume across 12 different test matrices, for processor counts of $K = 1024$, $K = 512$, and $K = 128$. Figures with a white background correspond to Total Communication Volume results, whereas those with a gray background correspond to Maximum Weighted Communication Volume. Each bar in the figures represents the geometric mean [37] of five independent runs. In each

bar chart, the first bar, colored in black, represents the communication volume caused by the P2P communication scheme for the corresponding processor count. Following the first bar, there are three groups, each containing six bars, corresponding respectively to store-and-forward schemes with low ($n = 2$), medium ($n = \lceil (\max + \text{low})/2 \rceil$), and max ($n = \log_2 K$) VPT dimensions. Each group is composed of two consecutive pairs of bars, with a small gap between the pairs. Each group consists of two pairs of bars, separated by a small gap. Each pair of bars is shown in the same color: the first bar of the pair is filled with diagonal lines, while the second bar is left unfilled. In each pair, the first bar represents the communication volume resulting from the corresponding mapping scheme (Random, MapSI [23], MapMD [25]), while the second bar shows the communication volume achieved after applying the KL-based refinement heuristic to the initial mapping produced by that scheme.

As shown in all figures presenting experimental results, the KL-based heuristic achieves significant improvements in reducing the Total Communication Volume for low, mid, and max VPT dimensions. The amount of improvement becomes more significant with the increase in the number of processors due to the higher communication volume introduced by increased communication. The Total Communication Volume increases as the VPT dimensions grows due to the higher communication volume incurred by store-and-forward scheme. And, the percentage of improvement achieved also increases with as the VPT dimensions increase. The algorithm significantly reduces the total communication volume across all dimensions for all mapping methods (Random, MapSI, and MapMD). In some matrices, the improvement exceeds 50% on the random mapping (e.g., F1, fe_motorbuckle, bundle_Adj), and approaches 50% on the MapSI and MapMD mappings (e.g., pattern1, mip1, TSOPF_FS_b300_c3).

The algorithm successfully improves the Maximum Weighted Communication Volume for MapSI and MapMD mappings, with the exception of the human_gene2 matrix. However, the improvement percentage and absolute improvement in this metric are generally lower than those observed in Total Communication Volume. The reason is that the Maximum Weighted Communication Volume only considers the processor with the highest communication load. While the algorithm

improves MapSI and MapMD for Maximum Weighted Communication Volume metric, decreases in this metric are observed in some matrices with random mapping, such as `bundle_adj`, `GaAsH6`, `SiO2`, and `TSOPF_FS_b300_c3`. This can be explained by the fact that iterative refinement algorithms are sensitive to the quality of the initial solution; however, for most of the test matrices, the algorithm consistently reduces the communication volumes to similar levels, regardless of the VPT size, the number of processors, or the initial mapping method (Random, MapSI, or MapMD). This may be due to the effect of the store-and-forward scheme, which helps to regularize communication over the VPT. And, the parallelism between the store-and-forward communication pattern on the VPT and the structure of the search space used for swap selection likely contributes to the algorithm’s success in finding a good solution, on already reduced and regularized irregular communication.

Consistent improvement in Maximum Weighted Communication Volume was not achieved for `mip1`, `human_gene2`, and `TSOPF_FS_b300_c3` matrices under MapSI and MapMD mappings. In these cases, performance drops are observed across various VPT sizes, processor counts, and mappings. When examining the coefficient of variation (`cv`) and maximum degree ratio (`maxdr`) of these matrices in Table 5.1, it can be seen that they have high `cv` and `maxdr` values. Although `human_gene2` has relatively lower `maxdr` and `cv` values compared to `mip1`, no improvement was achieved on `human_gene2` in either Total Communication Volume and Maximum Weighted Communication Volume, except in the case of a 2-dimensional VPT with MapMD mapping. This may be due to the high matrix density in `human_gene2`, which results in a more balanced communication distribution after partitioning, and with the application of regularization, the optimization potential may have already reached saturation. However, when looking at the properties of the test matrices, a clear grouping between matrices with high and low performance improvements cannot be made. Matrices that show high performance gains appear to share characteristics with those that do not. A higher-dimensional analysis of matrix properties may be required for such meaningful classification.

In conclusion, the KL-based heuristic achieved significant improvements in

both Total Communication Volume and Maximum Weighted Communication Volume metrics across all 9 combinations of 3 different VPT dimensions and 3 various mapping methods tested. The most prominent outcome of these results is that the heuristic reduces communication volumes to similar levels, regardless of the number of dimensions of VPT, number of processors, or initial mapping methods. Based on this observed behavior, the KL-based heuristic can be considered as a new evaluation criterion for assessing the performance of constructive mapping methods on VPTs with the store-and-forward scheme, in terms of Total Communication Volume and Maximum Weighted Communication Volume.

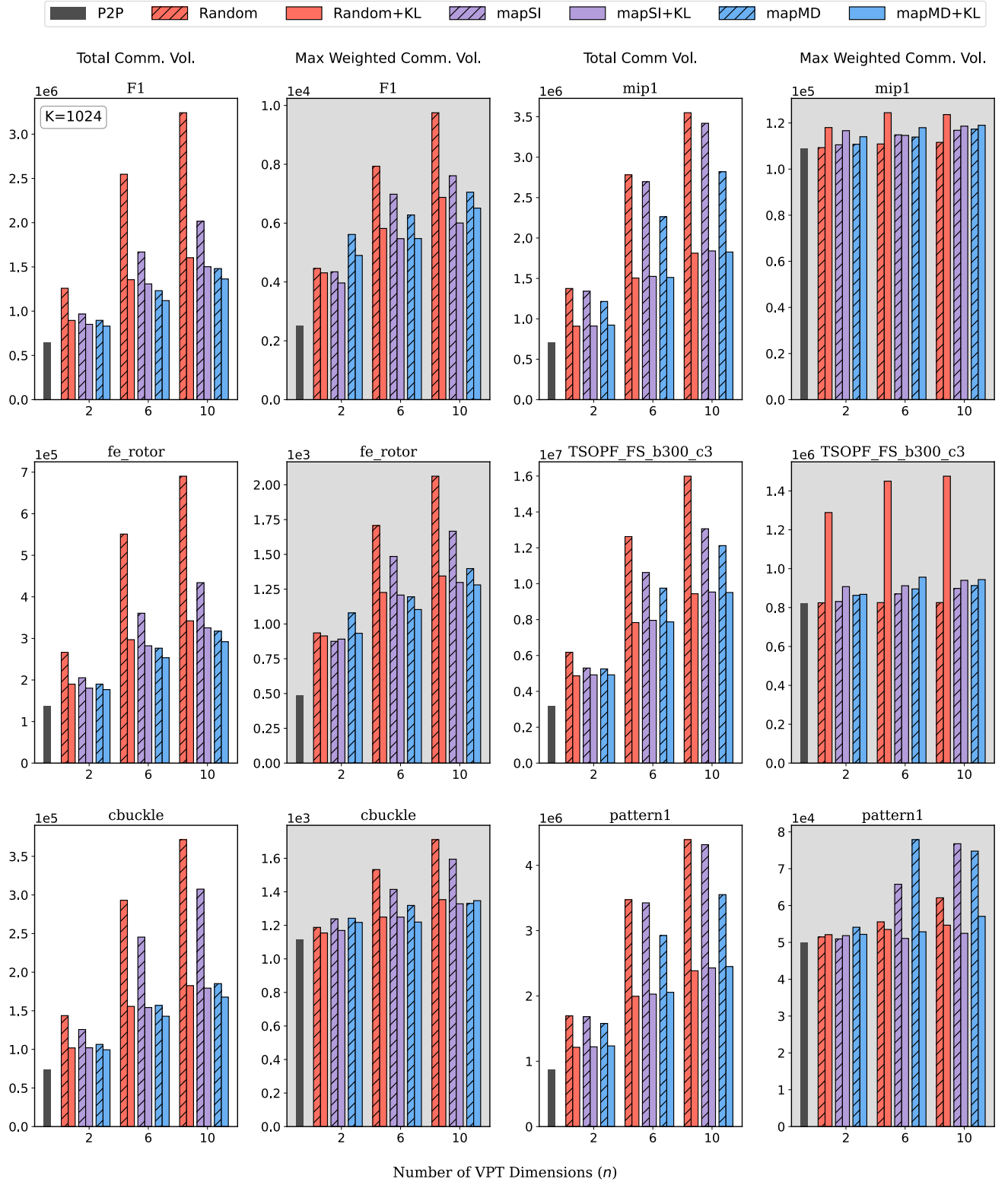


Figure 5.1: Total communication volume and maximum weighted communication volume for $K=1024$ processors in VPT dimensions of 2, 6, and 10.

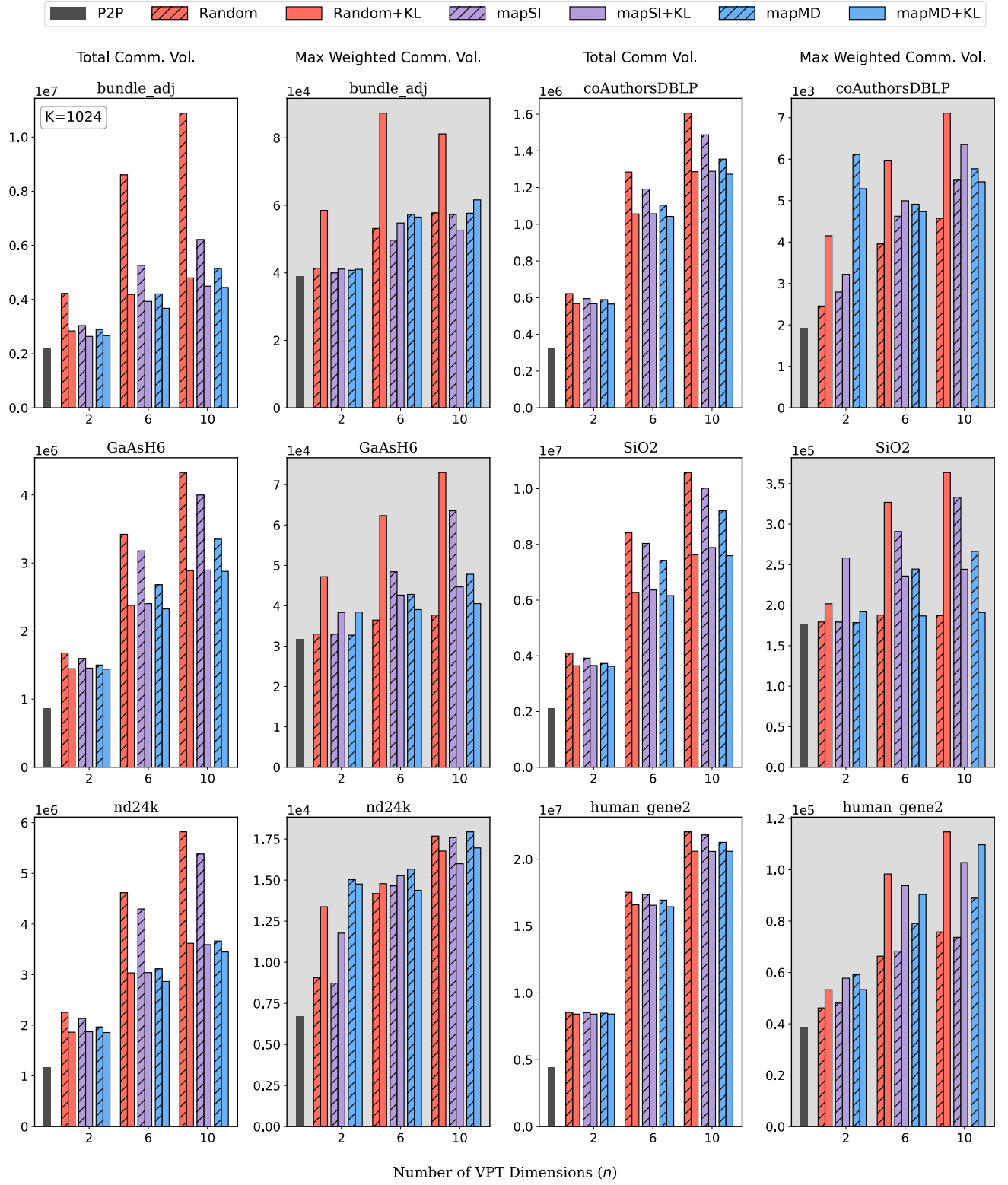


Figure 5.2: Total communication volume and maximum weighted communication volume for $K=1024$ processors in VPT dimensions of 2, 6, and 10.

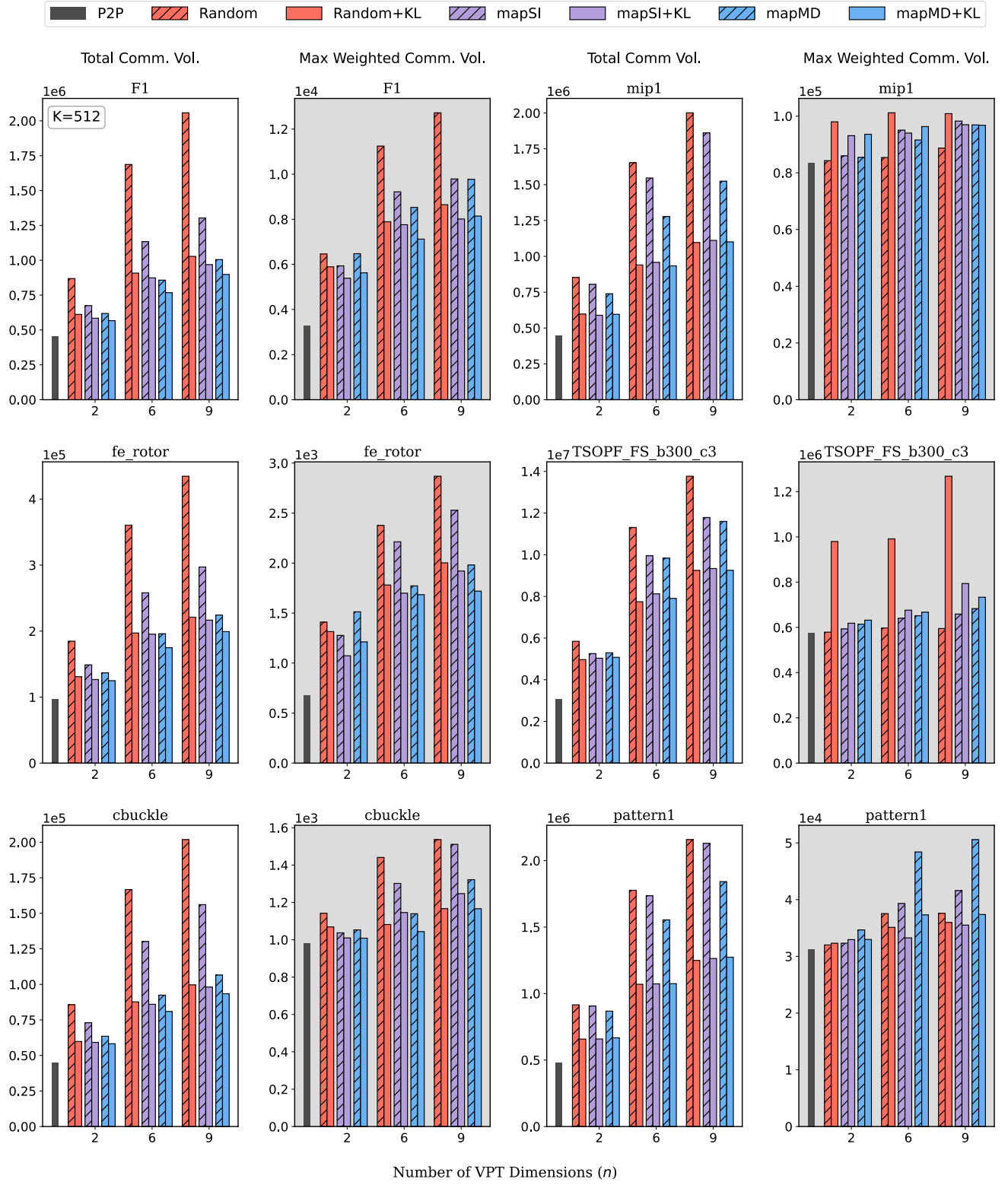


Figure 5.3: Total communication volume and maximum weighted communication volume for $K=512$ processors in VPT dimensions of 2, 6, and 9.

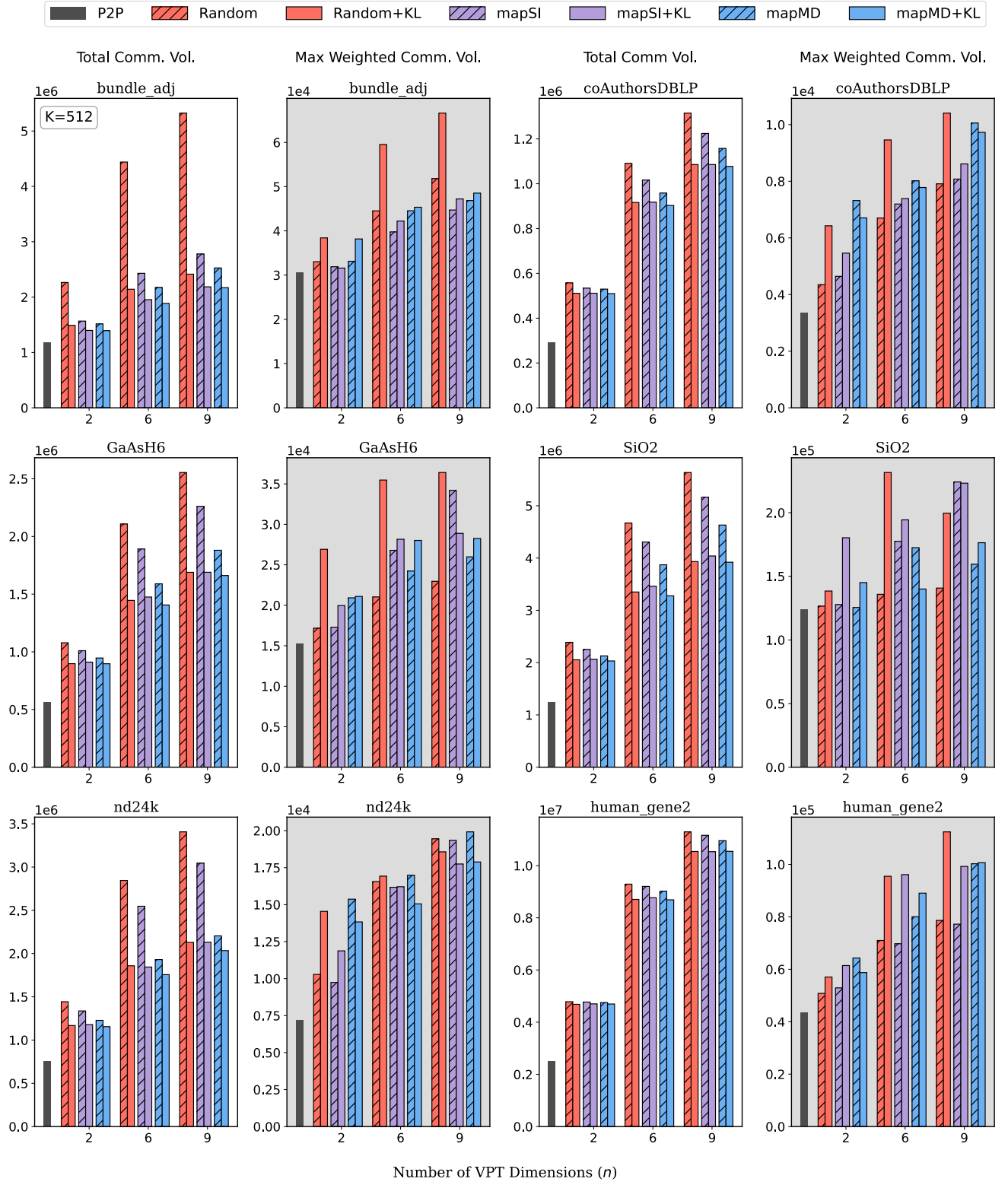


Figure 5.4: Total communication volume and maximum weighted communication volume for $K=512$ processors in VPT dimensions of 2, 6, and 9.

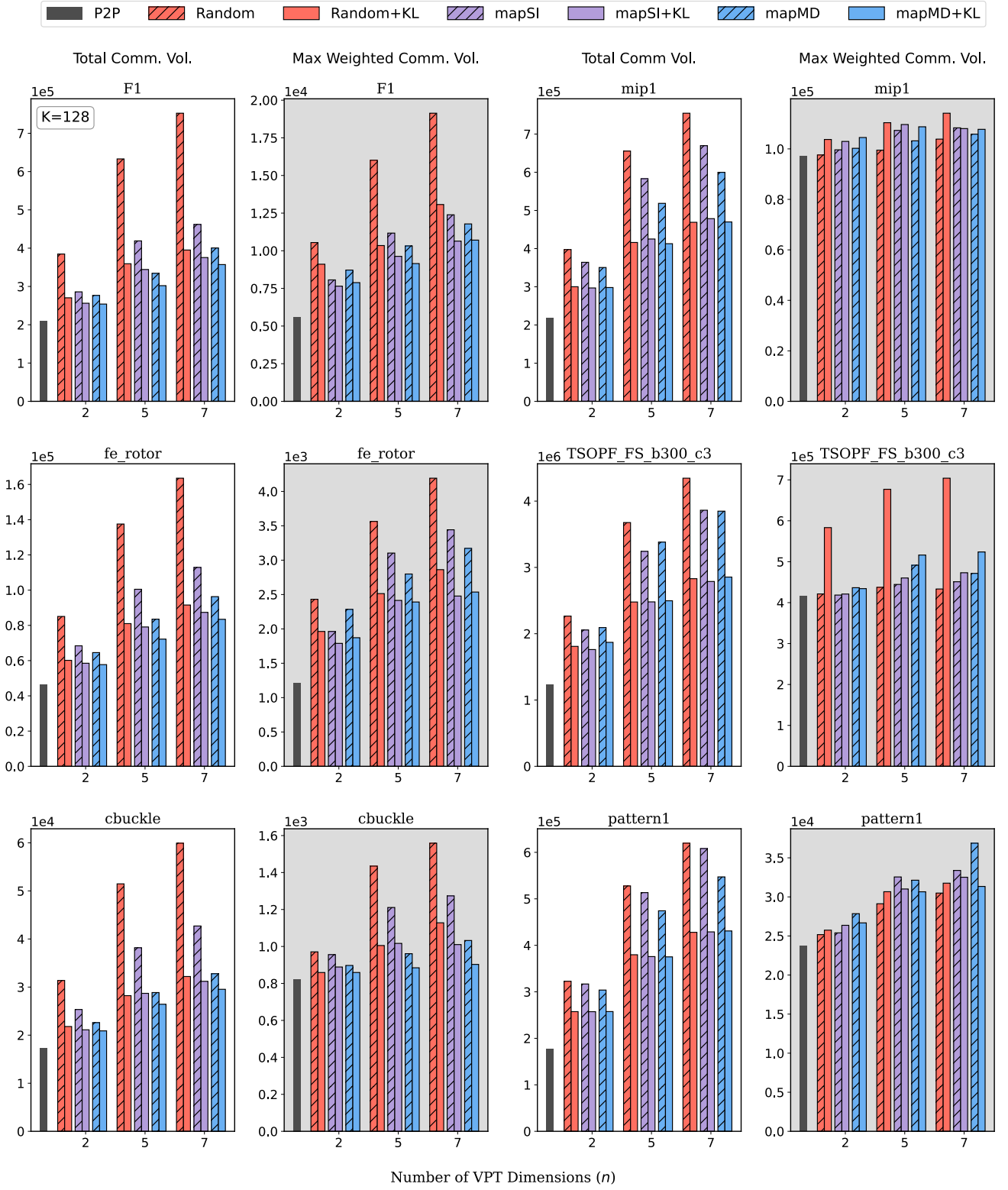


Figure 5.5: Total communication volume and maximum weighted communication volume for $K=128$ processors in VPT dimensions of 2, 5, and 7.

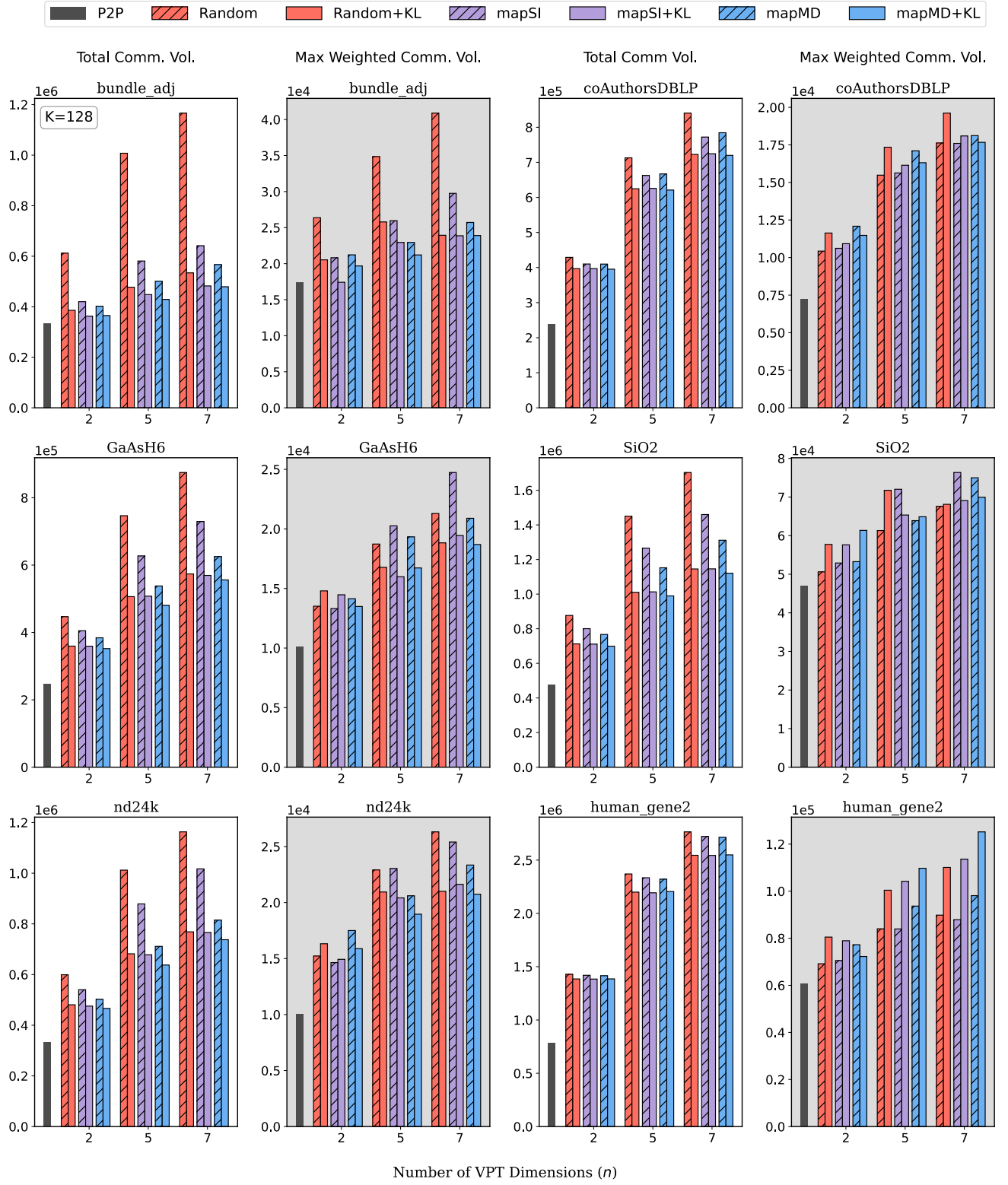


Figure 5.6: Total communication volume and maximum weighted communication volume for $K=128$ processors in VPT dimensions of 2, 5, and 7.

Chapter 6

Conclusion and Future Work

In this thesis, a KL-based iterative refinement heuristic was proposed to reduce communication volume increases in latency-efficient store-and-forward schemes. The method aims to reduce the total communication volume of SpMV and SpMM operations by placing tasks with high communication volume between them close to each other in a one-to-one task-to-processor mapping. The success of the proposed method was tested on square symmetric sparse matrices with irregular communication patterns partitioned using PaToH [20]. The KL-based heuristic significantly reduced the total communication volume of one-to-one task-to-processor mappings on VPTs generated by random and constructive methods. A key outcome of the algorithm is its ability to reduce communication volumes to similar levels regardless of VPT size, number of processors, or initial mapping method. This behavior suggests that the algorithm can be used as a benchmark for evaluating the performance of other mapping methods.

The KL-based algorithm operates using only the task-to-task communication matrix as input, which stores the sizes of messages sent between tasks. It does not require tracking messages through communication stages; instead, it performs refinements based solely on changes in total communication volume. For future work, the communication matrix could be modified to better reflect the characteristics or requirements of the target application, which may help expand the

search space for finding better mappings. Additionally, the gain calculation used during swap pair selection could be adapted to target the specific communication metric being optimized.



Bibliography

- [1] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection,” *ACM Transactions on Mathematical Software*, vol. 38, no. 1, Dec. 2011, sparse matrix collection website: <https://sparse.tamu.edu/>. [Online]. Available: <https://doi.org/10.1145/2049662.2049663>
- [2] O. Selvitopi and C. Aykanat, “Regularizing irregularly sparse point-to-point communications,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: <https://doi.org/10.1145/3295500.3356187>
- [3] B. Kernighan and S. Lin, “An efficient heuristic procedure for partitioning graphs,” *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, feb 1970.
- [4] J. Li, G. Michelogiannakis, B. Cook, D. Cooray, and Y. Chen, “Analyzing resource utilization in an hpc system: A case study of nersc’s Perlmutter,” in *High Performance Computing*, A. Bhatele, J. Hammond, M. Baboulin, and C. Kruse, Eds. Cham: Springer Nature Switzerland, 2023, pp. 297–316.
- [5] A. Shehabi, A. Newkirk, S. J. Smith, A. Hubbard, N. Lei, and e. a. Siddik, Md Abu Bakar, “2024 united states data center energy usage report,” Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-2001637, 2024. [Online]. Available: <https://escholarship.org/uc/item/32d6m0d1>
- [6] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, and et al., “The international exascale software project roadmap,” *Int. J.*

- High Perform. Comput. Appl.*, vol. 25, no. 1, p. 3?60, Feb. 2011. [Online]. Available: <https://doi.org/10.1177/1094342010391989>
- [7] R. O. Selvitopi, “Latency-centric models and methods for scaling sparse operations,” Ph.D. Thesis, Department of Computer Engineering, Bilkent University, 2016, thesis no: 434226.
 - [8] V. Kumar, *Introduction to Parallel Computing*, 2nd ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2002.
 - [9] “OSU Micro-Benchmarks,” <https://mvapich.cse.ohio-state.edu/benchmarks/>, accessed: 2025-07-19.
 - [10] T. Hoeﬂer, E. Jeannot, and G. Mercier, *An Overview of Topology Mapping Algorithms and Techniques in High-Performance Computing*, 2014, pp. 73–94. [Online]. Available: <https://ieeexplore.ieee.org/document/9005067>
 - [11] “numpy.ravel; NumPy v2.3 Manual — numpy.org,” <https://numpy.org/doc/stable/reference/generated/numpy.ravel.html>, [Accessed 29-07-2025].
 - [12] “numpy.ndarray.strides; NumPy v2.3 Manual — numpy.org,” <https://numpy.org/doc/stable/reference/generated/numpy.ndarray.strides.html>, [Accessed 29-07-2025].
 - [13] MPI Forum, “MPI forum,” <https://www.mpi-forum.org/>, accessed: 2025-07-20.
 - [14] Open MPI Project, “Open mpi documentation,” <https://www.open-mpi.org/doc/>, accessed: 2025-07-20.
 - [15] T. S. Woodall, G. M. Shipman, G. Bosilca, R. L. Graham, e. B. Maccabe, Arthur B.”, J. L. Träff, J. Worringer, and J. Dongarra, “High performance rdma protocols in hpc,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 76–85.

- [16] T. Hoefler, R. Rabenseifner, H. Ritzdorf, B. R. de Supinski, R. Thakur, and J. Larsson, “The Scalable Process Topology Interface of MPI 2.2,” *Concurrency and Computation: Practice and Experience*, vol. 23, no. 4, pp. 293–310, Aug. 2010.
- [17] Lawrence Livermore National Laboratory, “MPI virtual topologies tutorial,” https://hpc-tutorials.llnl.gov/mpi/virtual_topologies/, accessed: 2025-07-19.
- [18] N. F. T. Abubaker, “Novel algorithms and models for scaling parallel sparse tensor and matrix factorizations,” Ph.D. Thesis, Department of Computer Engineering, Bilkent University, 2022, thesis no: 752083.
- [19] R. W. Hamming, “Error detecting and error correcting codes,” *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.
- [20] U. Catalyurek and C. Aykanat, “Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, 1999.
- [21] S. Sahni and T. Gonzalez, “P-complete approximation problems,” *Journal of the ACM*, vol. 23, no. 3, pp. 555–565, jul 1976. [Online]. Available: <https://doi.org/10.1145/321958.321975>
- [22] M. Beckmann and T. C. Koopmans, “Assignment problems and the location of economic activities,” *Econometrica*, vol. 25, no. 1, pp. 53–76, 1957. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.2307/195701>
- [23] T. Hoefler and M. Snir, “Generic topology mapping strategies for large-scale parallel architectures,” in *Proceedings of the International Conference on Supercomputing*, ser. ICS ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 75–84, https://hpc-tutorials.llnl.gov/mpi/virtual_topologies/. [Online]. Available: <https://doi.org/10.1145/1995896.1995909>

- [24] R. Glantz, H. Meyerhenke, and A. Noe, “Algorithms for mapping parallel processes onto grid and torus architectures,” in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2015, pp. 236–243.
- [25] B. Brandfass, T. Alrutz, and T. Gerhold, “Rank reordering for mpi communication optimization,” *Computers and Fluids*, vol. 80, pp. 372–380, 2013, selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S004579301200028X>
- [26] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998. [Online]. Available: <https://doi.org/10.1137/S1064827595287997>
- [27] F. Pellegrini and J. Roman, “Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs,” in *High-Performance Computing and Networking*, H. Liddell, A. Colbrook, B. Hertzberger, and P. Sloot, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 493–498.
- [28] P. Sanders and C. Schulz, “Think Locally, Act Globally: Highly Balanced Graph Partitioning,” in *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA’13)*, ser. LNCS, vol. 7933. Springer, 2013, pp. 164–175.
- [29] D. Padua, Ed., *Encyclopedia of Parallel Computing*. Boston, MA: Springer, 2011.
- [30] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, “Multilevel hypergraph partitioning: applications in vlsi domain,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 1, pp. 69–79, March 1999.
- [31] B. Vastenhouw and R. H. Bisseling, “A two-dimensional data distribution method for parallel sparse matrix-vector multiplication,” *SIAM Review*,

- vol. 47, no. 1, pp. 67–95, 2005. [Online]. Available: <https://doi.org/10.1137/S0036144502409019>
- [32] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen, “Hypergraph-based dynamic load balancing for adaptive scientific computations,” in *Proceedings of 21st International Parallel and Distributed Processing Symposium (IPDPS’07)*. IEEE, 2007, best Algorithms Paper Award.
- [33] E. Jeannot, “Process mapping on any topology with topomatch,” *Journal of Parallel and Distributed Computing*, vol. 170, pp. 39–52, 2022. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0743731522001794>
- [34] U. Çatalyürek, K. Devine, M. Faraj, L. Gottesbüren, T. Heuer, H. Meyerhenke, P. Sanders, S. Schlag, C. Schulz, D. Seemaier, and D. Wagner, “More recent advances in (hyper)graph partitioning,” *ACM Comput. Surv.*, vol. 55, no. 12, Mar. 2023. [Online]. Available: <https://doi.org/10.1145/3571808>
- [35] N. Jain, A. Bhatele, S. White, T. Gamblin, and L. V. Kale, “Evaluating hpc networks via simulation of parallel workloads,” in *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 154–165.
- [36] C. Fiduccia and R. Mattheyses, “A linear-time heuristic for improving network partitions,” in *19th Design Automation Conference*, 1982, pp. 175–181.
- [37] P. J. Fleming and J. J. Wallace, “How not to lie with statistics: the correct way to summarize benchmark results,” *Communication of the ACM*, vol. 29, no. 3, p. 218–221, Mar. 1986. [Online]. Available: <https://doi.org/10.1145/5666.5673>