Computational Science and Engineering

Technische Universität München

Master´s Thesis

Performance Evaluation of The Reshuffle Data-Redistribution Library

Chang-Gen Lai

Computational Science and Engineering

Technische Universität München

Master's Thesis

Performance Evaluation of The Reshuffle Data-Redistribution Library

Author:	Chang-Gen Lai
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Santiago Narváez, M.Sc.
Submission Date:	December 16th, 2024

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

December 16th, 2024

Chang-Gen Lai

16. 12. 2024.

Chang-Gen Lai

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Santiago Narvaez, for his invaluable guidance in both the execution of this project and the writing of this thesis. His mentorship has taught me how to cultivate good practices in conducting research and writing code systematically.

I am also profoundly thankful to my parents and friends in Munich for their unwavering support throughout my years of study at TUM. Their mental support and encouragement have been crucial in helping me complete this thesis.

Abstract

In high-performance computing (HPC), efficient data distribution across multi-node systems is crucial for optimizing parallel processing. Traditional static data distribution strategies often fall short in dynamic workloads, leading to suboptimal performance. This thesis investigates the Reshuffle library, which introduces elastic data distribution capabilities at runtime, leveraging the Message Passing Interface (MPI) to dynamically adjust data layouts across nodes. The study focuses on integrating the Reshuffle library within a distributed heat transfer simulation framework, using the Jacobi method as the core algorithm due to its simplicity and parallel efficiency.

The research employs a modular system architecture to facilitate the integration of the Reshuffle library, enabling dynamic data redistribution and enhancing load balancing and resource utilization. A series of benchmarking experiments were conducted to evaluate the library's performance across varying grid sizes and MPI rank configurations. The study also highlights the impact of different layout configurations on reshuffling efficiency.

This thesis contributes to the field of elastic HPC frameworks by providing a practical implementation and assessment of the Reshuffle library, offering insights into its potential applications beyond heat transfer simulations. Future work could explore the library's integration with other computational frameworks and optimize its algorithms for broader applicability.

Contents

1 Introduction

In high-performance computing (HPC), managing data distribution across multi-node systems is essential for efficient parallel processing. Traditionally, data distribution strategies are predefined and static, which can limit flexibility and performance in dynamic workloads or when resources fluctuate. Reshuffle library [1] introduces a novel approach to this problem by allowing for elastic data distribution at runtime. By leveraging Message Passing Interface (MPI), Reshuffle can dynamically adjust data layouts across nodes, enabling efficient resource usage and scalability in diverse computational environments.

This thesis focuses on evaluating the Reshuffle library within a distributed heat transfer simulation framework. Heat transfer simulation, a fundamental process in computational engineering, serves as an ideal testbed for evaluating data distribution methods due to its regular grid structure and heavy computational demands. The simulator uses the Jacobi method as its core algorithm, chosen for its simplicity and efficiency in handling iterative solutions over a distributed grid.

The Reshuffle library is still under development, and the intention of building this application is not only to benchmark its performance but also to demonstrate the usage of the library. This dual purpose aims to provide insights into the library's capabilities and potential areas for further enhancement.

The primary objectives of this research are to integrate the Reshuffle library within the heat transfer simulator and to benchmark its performance. The simulator's input parameters, including grid size and iteration count, are managed using Abseil [2], a command-line interface tool that allows for flexible configuration. To validate the results visually, the simulation outputs are stored in VTK [3] format, making them compatible with visualization tools that render the distribution and evolution of heat over time. Additionally, during development, the Google Test (GTest) [4] framework is employed to ensure that the application functions correctly, thereby facilitating a robust testing phase.

In summary, this research contributes to the growing field of elastic and adaptable HPC frameworks by demonstrating a practical implementation and assessment of the Reshuffle library. Through performance benchmarking and analysis, this study seeks to highlight the potential benefits and limitations of adaptive data distribution within distributed computing environments, offering insights that can be applied beyond heat transfer simulations to a broader range of computational applications.

This thesis is structured as follows:

- **Chapter 2** explores elastic data distribution in high-performance computing (HPC). It also provides the theoretical background of heat transfer, detailing the conversion of the heat equation into a numerical method using the Jacobi method.
- **Chapter 3** presents a comprehensive examination of the system's design and implementation, covering the methodology for setting up the testbed, integrating the Reshuffle library, managing data distribution, and detailing the core algorithm implementation and workflow.
- **Chapter 4** provides a detailed performance evaluation of the Reshuffle library. It includes validation of the application by comparing results with a sequential simulator, and presents benchmarking results through experiments that analyze time efficiency across varying grid sizes, scalability with different numbers of processors, and efficiency across different layout configurations.
- **Chapter 5** concludes the thesis by summarizing the key findings, discussing the implications for elastic HPC applications, and suggesting directions for future research.

2 Background Theory

2.1 Data Distribution in High-Performance Computing

2.1.1 Traditional MPI in High-Performance Computing

The MPI [5] is a standardized and portable message-passing system designed to function on a wide variety of parallel computing architectures. It is widely used in HPC for its ability to efficiently manage communication between processes running on distributed systems. Traditional MPI provides a set of communication protocols that allow processes to exchange data through point-to-point and collective communication operations. This capability is crucial for parallel applications that require synchronization and data sharing across multiple nodes. Despite its effectiveness, traditional MPI assumes a static data distribution model, which can lead to inefficiencies in applications with dynamic workloads.

2.1.2 Elastic Data Distribution in High-Performance Computing

Elastic data distribution seeks to address the limitations of static partitioning by allowing data to be dynamically reallocated across nodes based on runtime conditions. Adaptive data layouts can provide better load balancing, enhanced resource utilization, and potentially reduced communication overhead, all of which contribute to improved overall performance. Several studies [6, 7] have explored elastic data management in HPC, focusing on frameworks that modify data layouts at runtime to better distribute computational load. For instance, dynamic load-balancing algorithms in HPC applications have demonstrated significant performance improvements, particularly in applications involving irregular workloads. The Reshuffle library extends this concept by offering a general-purpose solution that allows for on-the-fly adjustments to data partitioning within MPI-based distributed applications.

2.1.3 Reshuffle Library and its Applications

The Reshuffle library is designed specifically to address the need for runtime elasticity in data distribution across multi-node environments. By leveraging MPI, Reshuffle provides a framework for redistributing data without interrupting the ongoing computational processes. This capability is particularly beneficial in applications where computational load may vary significantly over time, such as simulations involving dynamic physical processes. Reshuffle achieves this adaptability by enabling users to specify data layouts that can be altered at runtime, thus optimizing data locality and minimizing communication delays between nodes. While previous studies have proposed dynamic load-balancing solutions, Reshuffle's unique contribution lies in its flexible, MPI-compatible approach that is applicable across a range of HPC applications.

2.2 The Jacobi Method in Heat Transfer Simulations

The Jacobi method is a widely used iterative algorithm for solving partial differential equations, particularly those involved in steady-state and time-dependent heat transfer problems. In this thesis, the Jacobi method is selected as the core algorithm for the heat transfer simulator due to its simplicity and effectiveness in handling large grid-based calculations. The Jacobi method operates by iteratively updating grid points based on neighboring values, a process that lends itself well to parallel computing. This section demonstrates the transformation of the heat equation from its continuous PDE form to a discretized form, suitable for application within a Jacobi stencil.

2.2.1 Heat Equation PDE

The heat equation is a fundamental PDE [8] in physics that models the distribution of temperature $u(x, y, t)$ in a domain over time. In two dimensions, it is represented as:

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{2.1}$$

where:

- $u(x, y, t)$ is the temperature at a point (x, y) at time t,
- α is the thermal diffusivity of the material, which depends on its thermal conductivity, density, and specific heat.

The goal is to find a stable solution $u(x, y, t)$ that satisfies the heat equation under certain initial and boundary conditions.

2.2.2 Discretizing the Heat Equation for the Jacobi Method

To solve this equation using the Jacobi method, we discretize the spatial domain into a grid of points and approximate the second-order derivatives using finite difference methods. Let's assume a uniform grid with spacing Δx and Δy in the x and y directions, respectively, and a time step Δt.

The spatial derivatives can be approximated as [9]:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2} \tag{2.2}$$

$$\frac{\partial^2 u}{\partial y^2} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\Delta y)^2} \tag{2.3}$$

Substituting these finite differences into the heat equation, we obtain an approximation of the time derivative:

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} = \alpha \left(\frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{(\Delta x)^2} + \frac{u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}}{(\Delta y)^2} \right) \tag{2.4}$$

Rearranging to solve for the next time step $u_{i,j}^{n+1}$, we get:

$$u_{i,j}^{n+1} = u_{i,j}^{n} + \Delta t \cdot \alpha \left(\frac{u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}}{(\Delta x)^2} + \frac{u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n}}{(\Delta y)^2} \right) \tag{2.5}$$

In the Jacobi method, however, we seek the steady-state solution, so we consider only the spatial terms without the time derivative. For the steady-state heat equation, the above equation simplifies to a weighted average:

$$u_{i,j}^{\text{new}} = \frac{1}{4} \left(u_{i+1,j}^{\text{old}} + u_{i-1,j}^{\text{old}} + u_{i,j+1}^{\text{old}} + u_{i,j-1}^{\text{old}} \right) \tag{2.6}$$

This represents the Jacobi stencil, where the temperature at each point $u_{i,j}$ is updated as the average of its four nearest neighbors. Iteratively applying this update across all grid points leads to a solution that approximates the temperature distribution in the domain.

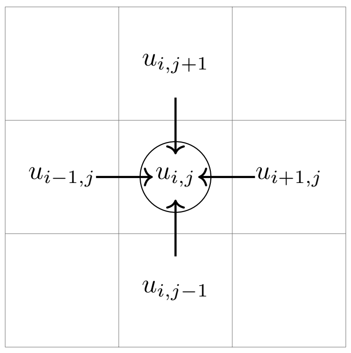


Figure 2.1: Jacobi stencil used in numerical heat transfer methods.

As shown in Figure 2.1, the central point is updated based on its neighbors.

2.2.3 Application in Parallel Computing

The Jacobi method's simplicity and reliance on neighboring points make it highly parallelizable, as updates to each grid point depend solely on the values from the previous iteration. This locality of reference allows for efficient parallelization across a multi-node environment, as each node can handle a subset of the grid with minimal communication overhead. However, in a distributed environment, data distribution becomes critical for performance. The Reshuffle library enables dynamic redistribution of these data blocks during runtime, potentially optimizing load balancing and reducing communication overhead, which can lead to significant performance gains in simulations.

In this study, we implement the Jacobi method on a multi-node system using MPI for parallel execution, with Reshuffle handling data redistribution dynamically. By benchmarking this approach, we aim to demonstrate the potential performance benefits of elastic data distribution in solving the heat equation.

2.3 Visualization of Simulation Data with VTK

Data visualization is essential in computational simulations, as it enables researchers to observe and interpret complex patterns within simulated processes. In this thesis, the VTK (Visualization Toolkit) file format is used to store output data from the heat transfer simulator. VTK is widely supported by various visualization tools and enables users to graphically represent the distribution and flow of heat across grid points. This graphical representation provides a means to validate the simulation results and observe the effects of different data distribution layouts enabled by Reshuffle, thus offering valuable insights into how elastic data distribution impacts computational performance.

To illustrate, a simple VTK file for a 2D grid of temperature data might look like the following:

```
# vtk DataFile Version 4.1
vtk output
ASCII
DATASET STRUCTURED_POINTS
DIMENSIONS 5 5 1
SPACING 1 1 1
ORIGIN 0 0 0
POINT_DATA 25
SCALARS ScalarField double
LOOKUP_TABLE default
0.0 1.0 2.0 3.0 4.0
1.0 2.0 3.0 4.0 5.0
2.0 3.0 4.0 5.0 6.0
3.0 4.0 5.0 6.0 7.0
4.0 5.0 6.0 7.0 8.0
```

In this example, the VTK file describes a 5x5 grid of temperature values. The `DIMENSIONS` keyword specifies the size of the grid, while `POINT_DATA` indicates the number of data points. The `SCALARS` keyword defines the data type and name of the scalar field, in this case, `temperature`. The actual temperature values are listed in a row-major order, representing the temperature at each grid point.

This simple VTK file can be visualized using tools like ParaView, allowing researchers to analyze the temperature distribution across the grid. By using VTK, the simulation results can be easily shared and interpreted, facilitating collaboration and further analysis.

2.4 Development Tools: Abseil and GTest

For practical and flexible command-line interface management, this thesis employs Abseil, a library that simplifies the handling of input flags. This library allows users to easily modify simulation parameters such as grid size, iteration count, and distribution layout, ensuring that the application is adaptable to varied testing conditions. Additionally, during the development phase, Google Test (GTest) is integrated to facilitate rigorous testing of the heat transfer simulator. GTest provides a straightforward framework for writing unit tests, which helps in identifying issues and verifying the correctness of each implementation phase.

The following snippet shows how MPI is initialized and finalized within a GTest environment, along with a test case for the 'isRoot' function:

```
#include <gtest/gtest.h>
#include "mpi_utils.hpp"

// Test for isRoot function
TEST(MPIUtilsTest, IsRoot) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        EXPECT_TRUE(isRoot());
    } else {
        EXPECT_FALSE(isRoot());
    }
}

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);

    ::testing::InitGoogleTest(&argc, argv);
    int result = RUN_ALL_TESTS();

    MPI_Finalize();
    return result;
}
```

This setup ensures that MPI is properly initialized and finalized around the execution of all tests, allowing for parallel testing of MPI-based functionalities. The 'isRoot' test case checks whether the current process is the root process (rank 0) in the MPI environment.

By combining Abseil for input flexibility and GTest for testing, this project ensures both adaptability and reliability in the application's design and functionality.

3 Application Design and Implementation

This chapter presents a comprehensive examination of the parallel heat transfer simulation system's design and implementation. The system employs the Jacobi iteration method to model two-dimensional heat distribution, incorporating sophisticated parallel computing techniques through efficient domain decomposition and process communication strategies. The implementation focuses on achieving both computational accuracy and parallel efficiency while maintaining code maintainability.

3.1 System Architecture Overview

The heat transfer simulator adopts a modular architecture that separates core computational logic from parallel distribution mechanisms. This architectural decision facilitates the integration of parallel computing capabilities while ensuring system maintainability and extensibility. The system comprises three primary components: the core computational engine implementing the Jacobi iteration method, the parallel distribution framework managing data partitioning, and the visualization subsystem generating standardized output for scientific analysis.

The computational engine operates on a two-dimensional domain where temperature values are computed iteratively across a square grid. This core component implements the fundamental heat transfer calculations while maintaining boundary conditions and ensuring numerical stability. The parallel distribution framework handles the complexities of data decomposition and inter-process communication, enabling efficient parallel execution across multiple processors. The visualization subsystem processes computational results and generates standardized output formats, facilitating detailed analysis of the simulation results.

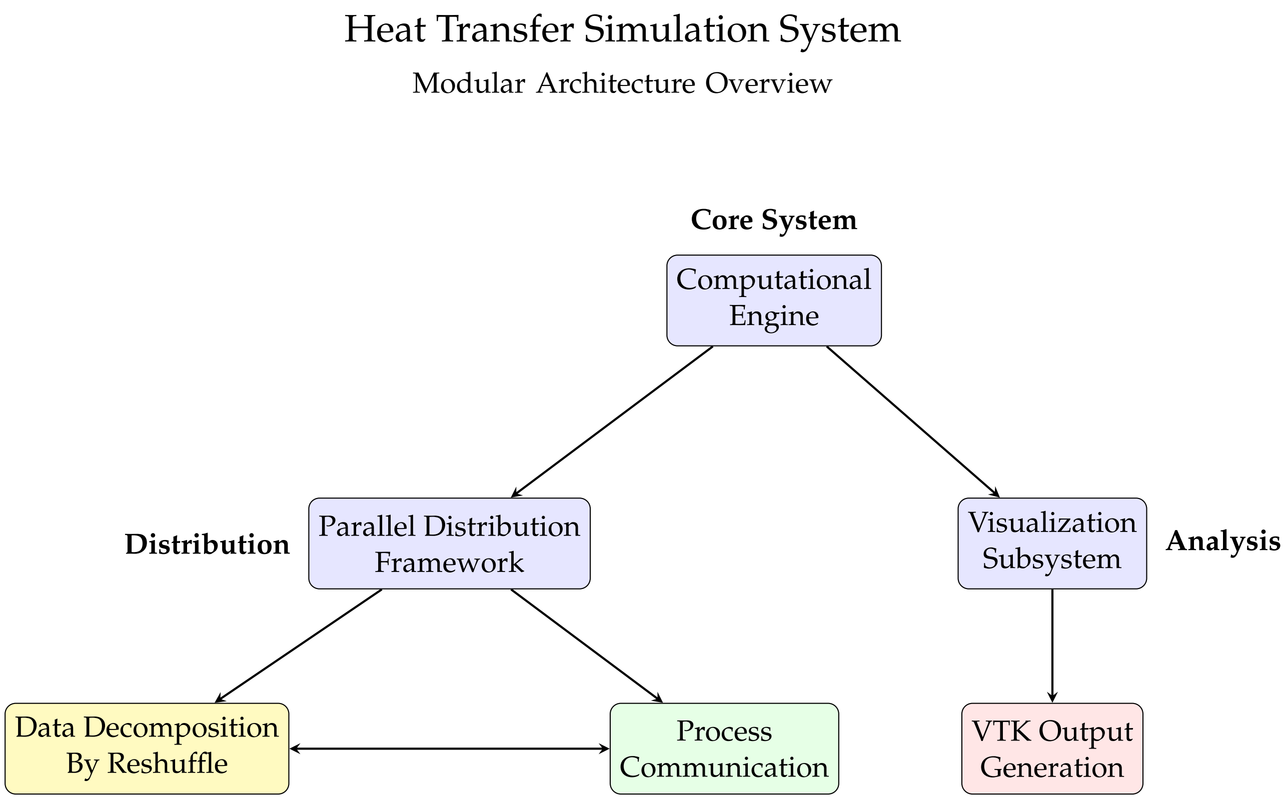

Heat Transfer Simulation System
Modular Architecture Overview
Core System
Computational Engine
Distribution
Parallel Distribution Framework
Visualization Subsystem
Analysis
Data Decomposition By Reshuffle
Process Communication
VTK Output Generation

Figure 3.1: System architecture showing the three main components and their interactions.

3.2 Computation Engine

3.2.1 Heat Transfer Model

The simulation employs a finite difference method to solve the heat equation in a two-dimensional domain. The computational space is discretized into a square grid, where each cell represents a discrete point at which temperature is calculated. The grid implementation utilizes a two-dimensional vector structure, providing a balance between efficient memory access and code maintainability. This data structure enables straightforward manipulation of temperature values while facilitating the implementation of parallel computing strategies.

The system establishes well-defined thermal gradients through fixed boundary conditions, setting edge cells to $100\,^{\circ}\mathrm{C}$ and initializing interior cells to $0\,^{\circ}\mathrm{C}$. This initialization creates a controlled environment for studying heat propagation patterns. This setup creates a stable thermal gradient that aids in the system's steady convergence throughout the computation process, ensuring both stability and accuracy of the simulation.

3.2.2 Jacobi Iteration Method

The core computational algorithm utilizes the Jacobi iteration technique, a well-established method for solving partial differential equations. This technique updates the temperature of each interior cell by computing the average of its four neighboring cells. The mathematical formulation of this process is detailed in the Background Theory section, where the Jacobi method is described using the stencil depicted in Equation (2.6).

This mathematical model is implemented in the `updateGrid` function, which systematically processes each interior point of the grid:

```
Matrix updateGrid(const Matrix &grid) {
    Matrix new_grid = grid;
    int rows = grid.size();
    int cols = grid[0].size();
    for (int i = 1; i < rows - 1; i++) {
        for (int j = 1; j < cols - 1; j++) {
            new_grid[i][j] = 0.25 * (grid[i - 1][j] +
                                    grid[i + 1][j] +
                                    grid[i][j - 1] +
                                    grid[i][j + 1]);
        }
    }
    return new_grid;
}
```

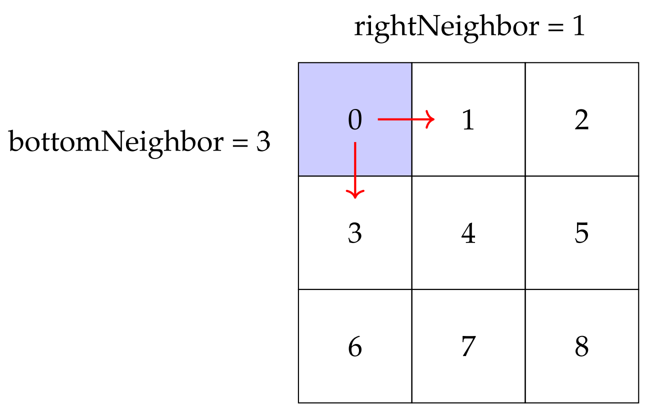
The implementation carefully manages the iteration process to ensure numerical stability. By maintaining consistent boundary conditions and employing proper memory management techniques, the system achieves reliable and accurate temperature calculations across the entire computational domain.

3.3 Domain Decomposition and Communication

3.3.1 Process Information Management

The simulation implements a sophisticated domain decomposition strategy that leverages both MPI communication primitives and the Reshuffle library's data redistribution capabilities. The system maintains detailed process topology information through a comprehensive ProcessInfo structure that tracks each process's position and its neighboring processes within the computational domain, as illustrated in Figure 3.2a:

This structure facilitates efficient communication patterns and enables proper management of boundary conditions across process boundaries, serving as the foundation for the distributed computation system. Figure 3.2b provides a detailed example of how the ProcessInfo structure maintains neighbor relationships for a specific process within the grid layout.



(a) 3x3 process grid layout with process 0 highlighted in blue. Red arrows indicate the neighboring processes.

ProcessInfo for rank 0:

rank = 0
layout_rows = 3
layout_cols = 3

hasTopNeighbor = false
hasBottomNeighbor = true
hasLeftNeighbor = false
hasRightNeighbor = true

topNeighbor = -1
bottomNeighbor = 3
leftNeighbor = -1
rightNeighbor = 1

(b) ProcessInfo structure details for process 0, showing its position and neighbor relationships in the process grid. When there is no neighbor in a particular direction (e.g., top and left for rank 0), the corresponding neighborIndex is set to MPI_PROC_NULL.

Figure 3.2: Illustration of process grid layout and ProcessInfo structure.

3.3.2 Distribution Pattern Management

The system implements a flexible distribution scheme that dynamically adapts to varying process counts and computational requirements. This scheme utilizes Reshuffle's blockwise distribution capabilities to generate different data partition patterns based on the available processes. The distribution pattern management consists of two key components:

Factor Pair Generation

The pattern acquisition process leverages the factorization of the MPI size to determine data distribution layouts. The `getFactorPairs` function implements this logic:

```cpp
Pairs getFactorPairs(int number) {
    Pairs allPairs{};
    // Generate all possible factor pairs
    for (int i = 1; i <= number; i++) {
        if (number % i == 0) {
            allPairs.emplace_back(i, number / i);
        }
    }

    Pairs result{};
    int allPairsLength = allPairs.size();
    int midPoint = (allPairsLength) / 2;

    // Select optimal subset based on process count characteristics
    if(allPairsLength == 2) {
        // Prime number case
        result.push_back(allPairs[0]);
        result.push_back(allPairs[1]);
    }
    else if(allPairsLength % 2 == 1) {
        // Perfect square case
        result.push_back(allPairs[0]);
        result.push_back(allPairs[midPoint]);
        result.push_back(allPairs[allPairsLength-1]);
    }
    else {
        // Other cases
        result.push_back(allPairs[0]);
        result.push_back(allPairs[midPoint-1]);
        result.push_back(allPairs[midPoint]);
        result.push_back(allPairs[allPairsLength-1]);
    }
    return result;
}
```

This implementation considers various process count scenarios:

- For prime numbers of processors, it selects two layouts. For example, with an MPI size of 5, the layouts are 1×5 and 5×1.
- For perfect square numbers, it generates three configurations including the square root layout. For example, with an MPI size of 16, the layouts are 1×16, 4×4, and 16×1.
- For other cases, it selects four configurations including balanced middle pairs. For example, with an MPI size of 20, the layouts are 1×20, 4×5, 5×4, and 20×1.

Distribution Generation

The system generates distribution patterns using Reshuffle's block-wise distribution capabilities:

```
std::vector<DataDistribution2D> generateDistributions(
    Pairs layout, int cols, int rows) {
    std::vector<DataDistribution2D> distributions;
    distributions.push_back({
        reshuffle::make_block_wise(cols, 1),
        reshuffle::make_block_wise(rows, 1)
    });
    for (const auto& pair : layout) {
        distributions.push_back({
            reshuffle::make_block_wise(cols, pair.first),
            reshuffle::make_block_wise(rows, pair.second)
        });
    }
    return distributions;
}
```

3.3.3 Ghost Layer Management

The system implements a ghost layer mechanism to handle distributed computations effectively. Each process maintains additional cells (ghost layers) around its local domain that contain copies of neighboring processes' boundary data. This approach ensures accurate computation of temperature values at subdomain boundaries while minimizing communication overhead, as illustrated in Figure 3.3.

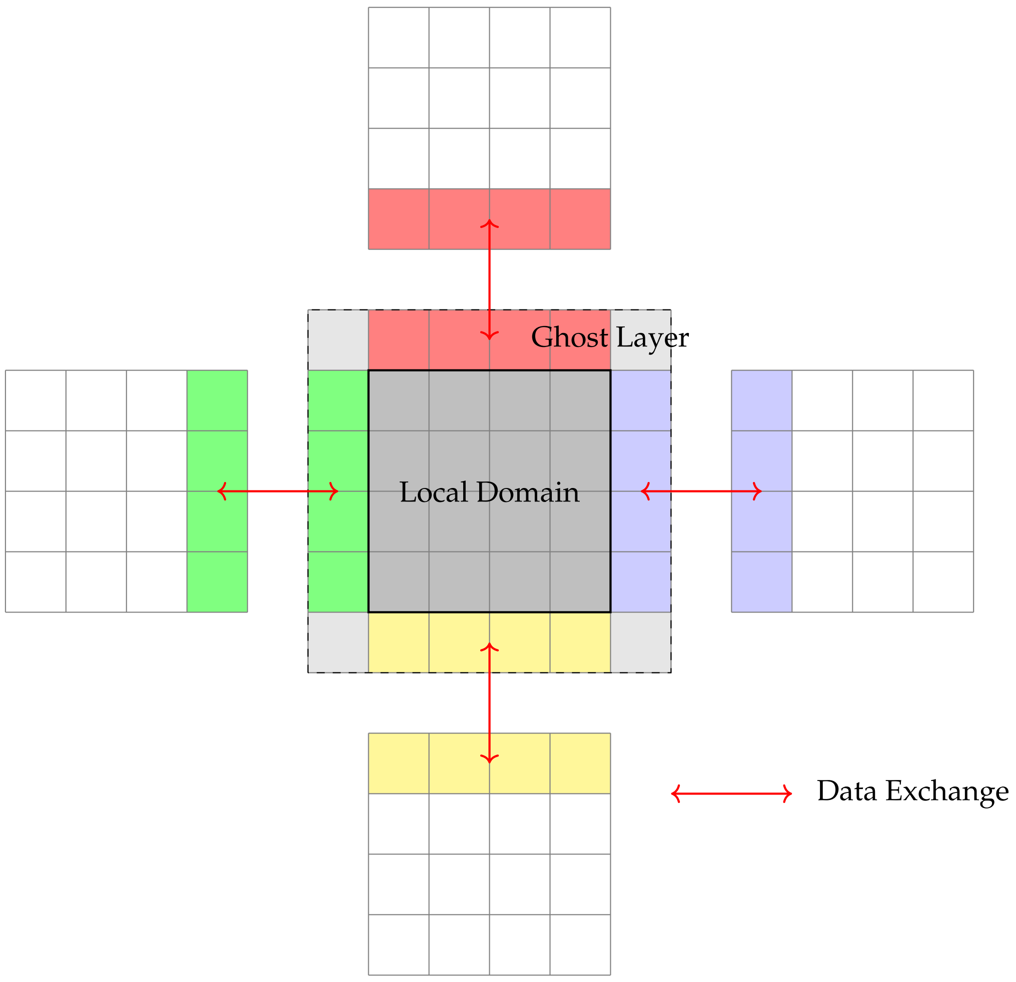


Figure 3.3: Illustration of the ghost layer mechanism. The local domain is surrounded by ghost layers that store boundary data from neighboring processes.

The system implements a ghost layer mechanism for handling boundary communications between processes. This is managed through four key functions:

Main Coordination Function

The `updateGhostLayers` function serves as the main coordinator for ghost layer operations:

```
void updateGhostLayers(Matrix& gridWithGhost, const ProcessInfo& procInfo) {
    if(procInfo.layoutRows == 1 && procInfo.layoutCols == 1) {
        return;
    }
    GhostBuffers buffers;
    fillGhostBuffers(gridWithGhost, procInfo, buffers);
    exchangeGhostBuffers(procInfo, buffers);
    updateGridWithGhostData(gridWithGhost, procInfo, buffers);
}
```

This function orchestrates the three-step process of ghost layer exchange:

1. Buffer preparation
2. Data exchange
3. Grid update

Buffer Preparation

The `fillGhostBuffers` function prepares the data for exchange:

```
void fillGhostBuffers(const Matrix& gridWithGhost,
                      const ProcessInfo& procInfo,
                      GhostBuffers& buffers) {
    int rows = gridWithGhost.size() -
               (procInfo.hasTopNeighbor ? 1 : 0) -
               (procInfo.hasBottomNeighbor ? 1 : 0);
    int cols = gridWithGhost[0].size() -
```

```
                (procInfo.hasLeftNeighbor ? 1 : 0) -
                (procInfo.hasRightNeighbor ? 1 : 0);

    // Prepare buffers for each direction
    if (procInfo.hasLeftNeighbor) {
        buffers.toLeft.resize(rows);
        buffers.fromLeft.resize(rows);
        for (int i = 0; i < rows; i++) {
            buffers.toLeft[i] = gridWithGhost[i +
                (procInfo.hasTopNeighbor ? 1 : 0)][1];
        }
    }
    // Similar operations for right, top, and bottom neighbors
}
```

Data Exchange

The `exchangeGhostBuffers` function handles the actual MPI communication:

```
void exchangeGhostBuffers(const ProcessInfo& procInfo,
                        GhostBuffers& buffers) {
    if (procInfo.hasLeftNeighbor) {
        MPI_Sendrecv(buffers.toLeft.data(), buffers.toLeft.size(),
                     MPI_DOUBLE, procInfo.leftNeighbor, 0,
                     buffers.fromLeft.data(), buffers.fromLeft.size(),
                     MPI_DOUBLE, procInfo.leftNeighbor, 0,
                     procInfo.comm, MPI_STATUS_IGNORE);
    }
    // Similar operations for right, top, and bottom neighbors
}
```

Grid Update

Finally, `updateGridWithGhostData` applies the received data to the grid:

```
void updateGridWithGhostData(Matrix& gridWithGhost,
                             const ProcessInfo& procInfo,
                             const GhostBuffers& buffers) {
    int rows = gridWithGhost.size() -
               (procInfo.hasTopNeighbor ? 1 : 0) -
               (procInfo.hasBottomNeighbor ? 1 : 0);
    int cols = gridWithGhost[0].size() -
               (procInfo.hasLeftNeighbor ? 1 : 0) -
               (procInfo.hasRightNeighbor ? 1 : 0);

    if (procInfo.hasLeftNeighbor) {
        for (int i = 0; i < rows; i++) {
            gridWithGhost[i + (procInfo.hasTopNeighbor ? 1 : 0)][0] =
                buffers.fromLeft[i];
        }
    }
    // Similar operations for right, top, and bottom neighbors
}
```

This ghost layer management system ensures proper communication between neighboring processes while maintaining the efficiency of the parallel computation. The system is particularly important during the Jacobi iteration process, where each cell needs access to its neighbors' values for accurate temperature calculations.

3.3.4 Communication Pattern Implementation

The system employs a hybrid communication approach that combines MPI communication primitives with Reshuffle's data redistribution capabilities. This design enables:

- Efficient nearest-neighbor communication during computation steps
- Flexible data redistribution when process counts or distribution patterns change
- Optimized ghost layer exchanges using `MPI_Sendrecv` operations
- Consistent boundary data maintenance across process boundaries

The communication pattern is particularly crucial during the Jacobi iteration process, where each cell requires access to its neighbors' values for accurate temperature calculations. The system implements explicit MPI communication for ghost layer exchanges while utilizing Reshuffle for broader data redistribution operations, creating an efficient and flexible parallel computation framework.

3.4 Visualization and Output Generation

3.4.1 VTK File Generation

The visualization system generates VTK (Visualization Toolkit) files that enable detailed analysis of simulation results. The output generation process follows a structured three-part format:

1. **Header Generation:** Defines the dataset type and geometry

VTK header structure
```
# vtk DataFile Version 4.1
vtk output
ASCII
DATASET STRUCTURED_POINTS
DIMENSIONS nx ny 1
SPACING 1 1 1
ORIGIN 0 0 0
```

2. **Data Description:** Specifies the type and format of the data

Data description
```
POINT_DATA nx*ny
SCALARS ScalarField double
LOOKUP_TABLE default
```

3. **Data Values:** Contains the temperature values for each grid point

The system implements the following specifications:

- **Dataset Type:** STRUCTURED_POINTS for regular grid data
- **Dimensions:** 2D grid represented as nx × ny × 1
- **Data Format:** ASCII format for maximum compatibility
- **Scalar Field:** Temperature values stored as double-precision numbers

3.4.2 Example Output

Here's an example of the temperature distribution visualization in VTK format:

Example VTK file content
```
# vtk DataFile Version 4.1
vtk output
ASCII
DATASET STRUCTURED_POINTS
DIMENSIONS 6 6 1
SPACING 1 1 1
ORIGIN 0 0 0
POINT_DATA 36
SCALARS ScalarField double
LOOKUP_TABLE default
value1 value2 value3 value4 value5 value6
value7 value8 value9 value10 value11 value12
value13 value14 value15 value16 value17 value18
value19 value20 value21 value22 value23 value24
value25 value26 value27 value28 value29 value30
value31 value32 value33 value34 value35 value36
```

The generated VTK files can be visualized using standard scientific visualization tools such as ParaView, which provides advanced capabilities for analyzing the simulation results. The system supports multiple visualization approaches, including:

- Contour plotting for temperature distribution analysis
- Heat map generation for spatial pattern examination
- Time-series animation for studying temporal evolution

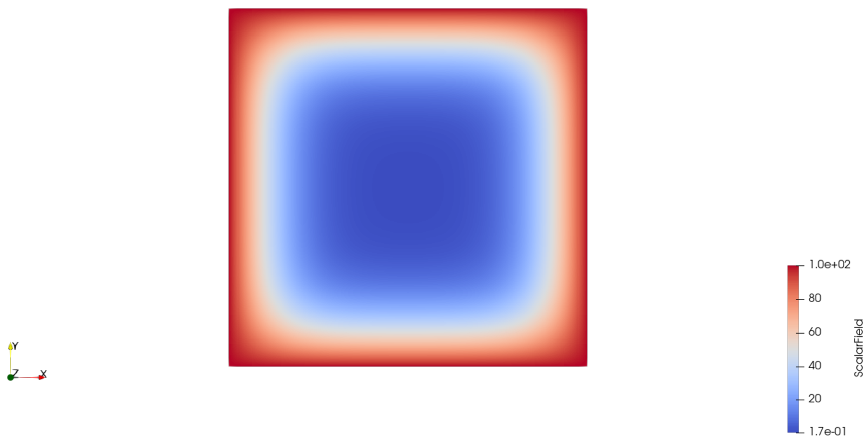


Figure 3.4: Visualization of temperature distribution from VTK output.

3.5 Program Workflow

The main simulation workflow combines all components into a cohesive system:

```cpp
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);

    // Parse command line arguments
    absl::ParseCommandLine(argc, argv);
    const int N = absl::GetFlag(FLAGS_N);
    const int iterNum = absl::GetFlag(FLAGS_i);

    // Initialize grid and generate distributions
    auto grid = isRoot() ? initializeGrid(N, N) : Matrix{};
    const Pairs factorPairs = getFactorPairs(size);
    const Pairs finalLayouts = getIterationPairs(factorPairs, iterNum);
    const std::vector<DataDistribution2D> distributions =
        generateDistributions(finalLayouts, N, N);

    // Main simulation loop
    for (int iter = 0; iter < iterNum; ++iter) {
        redistributeData(grid, distributions);
        updateGhostBuffers(grid, procInfo);
        jacobiIteration(grid, procInfo);
        if (isRoot()) writeVTKFile(grid, getOutputFilename());
    }

    MPI_Finalize();
    return 0;
}
```

This implementation ensures efficient parallel computation while maintaining solution accuracy through proper domain decomposition and communication patterns.

3.6 Testing and Validation

3.6.1 Testing Framework Integration

The system implements a comprehensive testing framework built upon Google Test (GTest), fully integrated with CMake for automated test execution. The CMake configuration enables seamless test discovery and execution through the following configuration:

```
# Find dependencies
find_package(MPI REQUIRED)
find_package(GTest REQUIRED)

# Create library with core functionality
add_library(heat_transfer_lib ${LIB_SOURCES})
target_link_libraries(heat_transfer_lib PUBLIC
    MPI::MPI_CXX
    absl::flags
    absl::flags_parse
    reshuffle::reshuffle
)

# Add tests directory
add_subdirectory(tests)
```

The test configuration is managed in a separate CMakeLists.txt within the tests directory, which handles the test-specific build requirements and dependencies. This modular approach maintains clean separation between the main application and test code while ensuring proper integration of all required components.

3.6.2 Test Implementation

The testing suite comprises two main components: MPI utilities testing and grid operations testing.

MPI Utilities Testing

The MPI utilities test suite (@test_mpi_utils.cpp) validates the parallel computing infrastructure:

```
// Test for process identification
TEST(MPIUtilsTest, IsRoot) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
```

```cpp
        EXPECT_TRUE(isRoot());
    } else {
        EXPECT_FALSE(isRoot());
    }
}

// Test for distribution pattern generation
TEST(MPIUtilsTest, GetFactorPairs) {
    // Test prime numbers
    auto pairs = getFactorPairs(3);
    EXPECT_EQ(pairs.size(), 2);
    EXPECT_EQ(pairs[0], std::make_pair(1, 3));
    EXPECT_EQ(pairs[1], std::make_pair(3, 1));

    // Test perfect square
    pairs = getFactorPairs(16);
    EXPECT_EQ(pairs.size(), 3);
    EXPECT_EQ(pairs[1], std::make_pair(4, 4));
}
```

Grid Operations Testing

The grid operations test suite (@test_grid_operations.cpp) validates the computational core:

```cpp
TEST(GridOperationsTest, InitializeGrid) {
    Matrix grid = initializeGrid(3, 3);
    EXPECT_EQ(grid.size(), 3);
    EXPECT_EQ(grid[0][0], 100.0); // Boundary condition
    EXPECT_EQ(grid[1][1], 0.0);   // Interior point
}

TEST(GridOperationsTest, UpdateGrid) {
    Matrix grid = {
        {100.0, 100.0, 100.0},
        {100.0, 0.0, 100.0},
        {100.0, 100.0, 100.0}
    };
    Matrix updated = updateGrid(grid);
    EXPECT_NEAR(updated[1][1], 50.0, 1e-5);
}
```

3.6.3 Multi-node Test Execution

The testing framework supports execution across multiple compute nodes using MPI. The tests are compiled into a GTest binary named `runTests`. Tests can be run using the following command:

```
mpirun -np <num_processes> ./runTests
```

For example, to run tests across 4 processes:

```
mpirun -np 4 ./runTests
```

The test suite automatically adapts to the available number of processes, validating:

- Process rank identification and root process determination
- Distribution pattern generation for various process counts
- Grid operations across process boundaries
- Inter-process communication patterns

Test execution in a multi-node environment ensures that parallel operations function correctly under realistic deployment conditions. The framework validates both the computational accuracy and the communication patterns essential for distributed execution.

3.7 User Interface and Configuration

3.7.1 Command-Line Interface

The system employs Abseil's command-line interface framework to provide a flexible and robust configuration mechanism. The primary configuration parameter, Grid Dimension (-N), controls the fundamental characteristics of the simulation domain. This parameter determines the dimensions of the square grid (N × N), directly impacting both the memory requirements and computational complexity of the simulation.

The Iteration Count (-i) parameter serves as the primary control for simulation duration and convergence characteristics. This parameter determines the number of Jacobi iterations performed during the simulation, allowing users to balance computation time against solution accuracy. The relationship between iteration count and solution convergence is particularly important for achieving accurate results while maintaining reasonable computation times.

For example, to execute the simulation with:

- 4 processors
- 100 × 100 grid size
- 20 iterations

The following command would be used:

```
mpirun -np 4 heat_reshuffle.out -N 100 -i 20
```

This command-line interface design enables users to conduct various experimental configurations without requiring source code modifications. The interface supports both development testing scenarios and production deployments, providing a consistent mechanism for controlling simulation behavior across different use cases.

3.8 Performance Considerations

3.8.1 Numerical Stability

The implementation incorporates several critical mechanisms to ensure numerical stability throughout the simulation process:

- **Boundary Condition Management:** The system maintains fixed boundary conditions of 100 °C through the `initializeGrid` function, ensuring consistent thermal gradients throughout the simulation:

```
Matrix initializeGrid(int rows, int cols){
    Matrix grid(rows, std::vector<Real>(cols, 0.0));
    for (int i = 0; i < rows; i++) {
        grid[i][0] = 100.0;
        grid[i][cols-1] = 100;
    }
    for (int j = 0; j < cols; j++) {
        grid[0][j] = 100.0;
        grid[rows - 1][j] = 100.0;
    }
    return grid;
}
```

- **Ghost Layer Synchronization:** The system implements rigorous ghost layer management through the `updateGhostLayers` function, ensuring accurate computation at subdomain boundaries while maintaining numerical consistency across process boundaries.

- **Jacobi Iteration Stability:** The implementation uses a fixed weight factor of 0.25 in the Jacobi iteration, which is standard for averaging the four neighboring grid points in a discretized 2D domain. This setup ensures convergence to a steady-state solution given the fixed boundary conditions of 100 °C and initial interior values of 0 °C. The fixed boundary conditions provide a stable thermal gradient, which aids in convergence. However, the Jacobi method is known for its slow convergence rate, especially for large grids.

```
new_grid[i][j] = 0.25 * (grid[i - 1][j] +
                          grid[i + 1][j] +
                          grid[i][j - 1] +
                          grid[i][j + 1]);
```

3.8.2 Communication Efficiency

The parallel implementation incorporates several optimizations to minimize communication overhead:

- **Efficient Ghost Layer Exchange:** The system uses `MPI_Sendrecv` operations for simultaneous bidirectional communication, reducing latency in ghost layer updates:

```
MPI_Sendrecv(buffers.toLeft.data(), buffers.toLeft.size(),
             MPI_DOUBLE, procInfo.leftNeighbor, 0,
             buffers.fromLeft.data(), buffers.fromLeft.size(),
             MPI_DOUBLE, procInfo.leftNeighbor, 0,
             procInfo.comm, MPI_STATUS_IGNORE);
```

- **Selective Communication:** The system performs ghost layer exchanges only when necessary, avoiding unnecessary communication in single-process scenarios:

```
if(procInfo.layoutRows == 1 && procInfo.layoutCols == 1) {
    return;
}
```

These performance optimizations work together to create an efficient parallel implementation while maintaining solution accuracy. The careful balance of numerical stability, memory efficiency, and communication optimization enables the system to handle large-scale heat transfer simulations effectively while producing reliable results.

4 Performance Evaluation

4.1 Validation of Application with Reshuffle Library

In this subsection, we validate the application using the reshuffle library by comparing its results against those of the Serial Heat Simulator. The Serial Heat Simulator serves as the benchmark, while the Distributed Heat Simulator is validated to ensure consistent results and verify the application's functionality.

Both simulators were executed on a 500x500 grid over 30,000 iterations. The Distributed Heat Simulator was executed in different layouts: the first 10,000 iterations were run after reshuffling the grid into a 1x4 layout, the second 10,000 iterations under a 2x2 layout (reshuffled from 1x4), and the last 10,000 iterations under a 4x1 layout. The results were extracted in VTK file format at 10,000, 20,000, and 30,000 iterations for both simulators. These VTK files were then inputted into ParaView for visualization purposes.

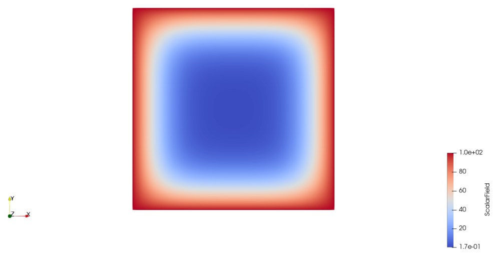
To validate the Distributed Heat Simulator, we used the `diff` command in Linux bash to compare the outputs. The command checks for differences between the two result files, ensuring that the Distributed Heat Simulator produces the expected results.

```
$ diff output_10000.vtk output_reshuffle_10000.vtk
$ diff output_20000.vtk output_reshuffle_20000.vtk
$ diff output_30000.vtk output_reshuffle_30000.vtk
```

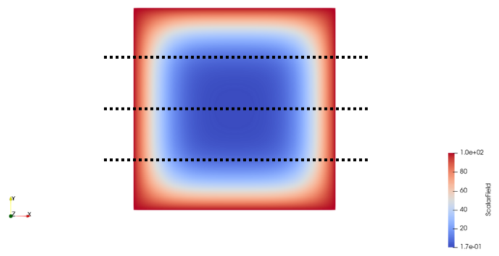
The comparison showed that the results from both versions are identical at each iteration checkpoint, confirming that the reshuffle library implementation is correct and matches the expected output from the Serial Heat Simulator.

This validation step is crucial to ensure the accuracy and reliability of the reshuffle library in parallelizing the heat transfer simulation.

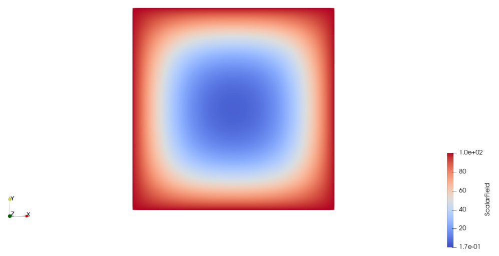
The figures in Figure 4.1 demonstrate the consistency of the reshuffle library's output with the Serial Heat Simulator across different iterations, validating the application's results.



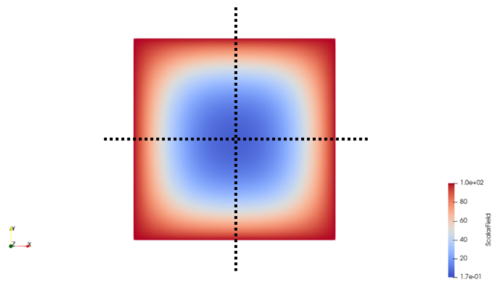
(a) Serial Heat Simulator - 10,000 iterations



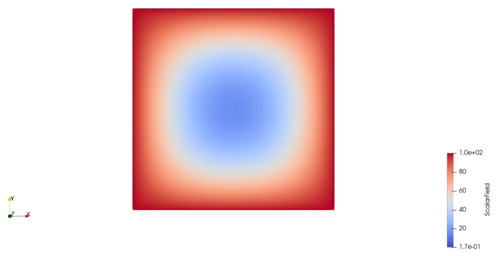
(b) Reshuffle (1x4) - 10,000 iterations



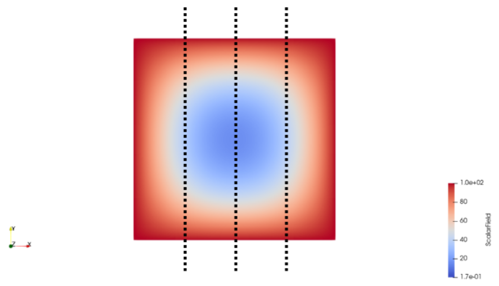
(c) Serial Heat Simulator - 20,000 iterations



(d) Reshuffle (2x2) - 20,000 iterations



(e) Serial Heat Simulator - 30,000 iterations



(f) Reshuffle (4x1) - 30,000 iterations

Figure 4.1: Comparison of heat transfer simulation results between Serial Heat Simulator and Distributed Heat Simulator at different iterations.

4.2 Benchmarking of Reshuffle Library

In this section, we present the benchmarking results of the reshuffle library through three distinct experiment configurations. These experiments are designed to evaluate the performance and efficiency of the reshuffle library under various conditions:

- **Time Analysis for Varying Grid Sizes:** The first experiment measures the time taken to reshuffle data from a single rank to 16 ranks arranged in a 4x4 layout. This experiment is conducted with varying grid sizes to assess how the reshuffle library handles different data volumes and configurations.
- **Scalability Evaluation with Varying Processors:** In the second experiment, we evaluate the time consumed in reshuffling data from a single rank to n^2 ranks in an n x n layout. This experiment is performed with a fixed grid size and varying number of processors to determine the scalability of the reshuffle library as the number of ranks increases.
- **Efficiency Comparison Across Layout Configurations:** The third experiment aims to compare the efficiency of the reshuffle library across different layouts with a fixed number of processors. The layouts considered are (1 x n^2), (n x n), and (n^2 x 1). This comparison helps in understanding the impact of layout configuration on the performance of the reshuffle library.

These experiments provide a comprehensive evaluation of the reshuffle library's performance, offering insights into its efficiency and scalability in parallel computing environments.

4.2.1 Results and Analysis of Reshuffling Efficiency for Different Grid Sizes

This experiment aims to benchmark the reshuffle efficiency against varying grid sizes. The experiment is conducted using 16 MPI ranks, with grid sizes ranging from 1000 x 1000 to 40000 x 40000, increasing in length by 100. The 40000 x 40000 grid size was the maximum limit due to storage

The process begins by initializing the grid in the root rank. The grid is then distributed to a 4 x 4 MPI process layout, and the time taken for this distribution is measured. For each grid size, six measurements are taken to calculate the average distribution time, ensuring the reliability of the results.

The results indicate that as the grid size increases, the time taken for reshuffling also increases. This trend is expected due to the larger data volumes being handled. The standard deviation of the reshuffling times is relatively small, indicating consistent performance across trials and is too small to be presented in the plot. Figure 4.2 shows the average times for each grid size.

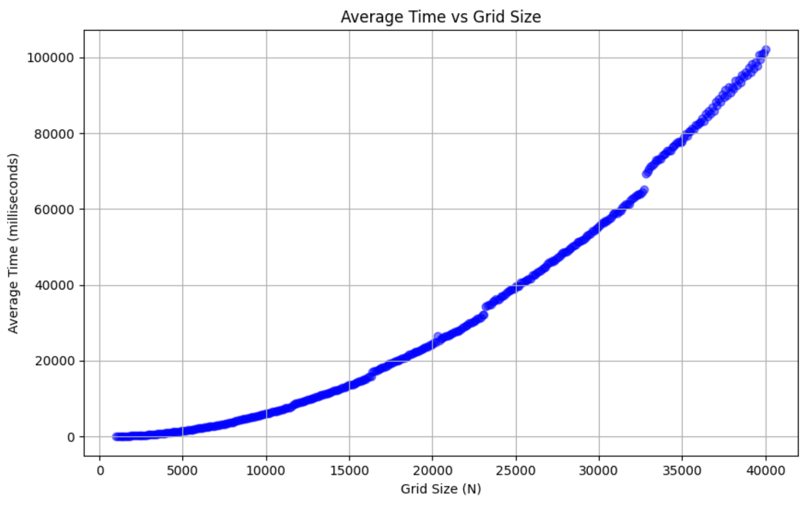


Figure 4.2: Average reshuffling time as grid size increases. This figure shows how reshuffling time grows with grid sizes from 1000 x 1000 to 40000 x 40000 in a 4x4 MPI layout, demonstrating the library's scalability with larger data volumes.

Further analysis with a regression line, shown in Figure 4.3, reveals an $O(n^2)$ relationship, indicating that the time spent is proportional to the total number of data points in the grid.

4.2.2 Results and Analysis of Reshuffling Efficiency for Different MPI Ranks

This experiment aims to benchmark the reshuffle efficiency against the size of the MPI rank. The experiment is conducted with a fixed grid size of 10,000 x 10,000, while varying the size of MPI ranks from 2 x 2 to 10 x 10.

The process begins by initializing the grid in the root rank. The grid is then distributed to an n x n MPI process layout, and the time taken for this distribution is measured. For each configuration, six measurements are taken to calculate the average distribution time, ensuring the reliability of the results.

The results indicate that as the number of MPI ranks increases, the time taken for reshuf-

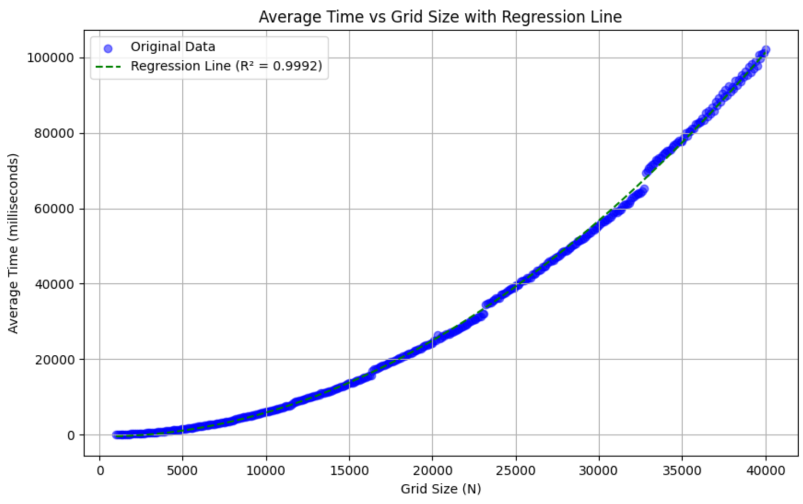


Figure 4.3: Average reshuffling time vs. grid size with regression line, showing $O(n^2)$ complexity. This indicates quadratic growth in reshuffling time as grid size increases, demonstrating the library's scalability.

fling changes. This trend provides insights into the scalability of the reshuffle library with respect to the number of processes.

Further analysis with a regression line, shown in Figure 4.4, reveals a second-order relationship, indicating that the time spent is proportional to the total number of processors. This demonstrates the efficiency of the reshuffle library in managing increasing numbers of MPI ranks effectively.

This analysis helps in understanding the impact of increasing the number of MPI ranks on the reshuffle library's performance, highlighting its efficiency and scalability in parallel computing environments.

4.2.3 Results and Analysis of Reshuffling Efficiency Across Different Layout Patterns

This experiment aims to benchmark the reshuffle efficiency between different layout patterns with a fixed number of MPI ranks. The experiment is conducted using 16 MPI ranks, with a grid size of 10,000 x 10,000.

The reshuffling time is measured for transitions between different layout patterns: from

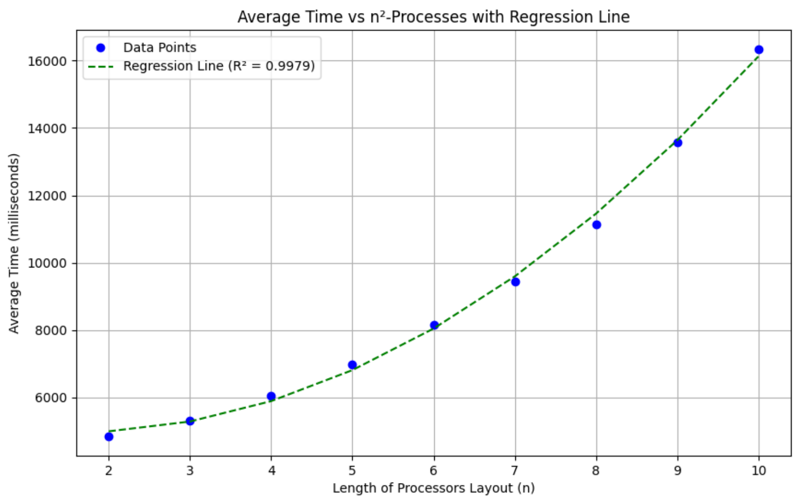


Figure 4.4: Average reshuffling time vs. number of MPI processes with regression line, showing $O(n^2)$ complexity. This indicates quadratic growth in reshuffling time as the number of MPI processes increases, demonstrating the library's scalability.

16 x 1 to 4 x 4, 4 x 4 to 1 x 16, and 1 x 16 to 16 x 1. For each layout transition, six measurements are taken to calculate the average reshuffling time, ensuring the reliability of the results.

The results are represented in Figure 4.5, which shows the average reshuffling time for each layout transition. This analysis provides insights into the impact of different layout configurations on the reshuffle library's performance. The standard deviation of the reshuffling times is relatively small, and these values are presented in Table 4.1.

This experiment highlights the efficiency of the reshuffle library in handling various layout configurations, offering valuable insights into its performance in parallel computing environments.

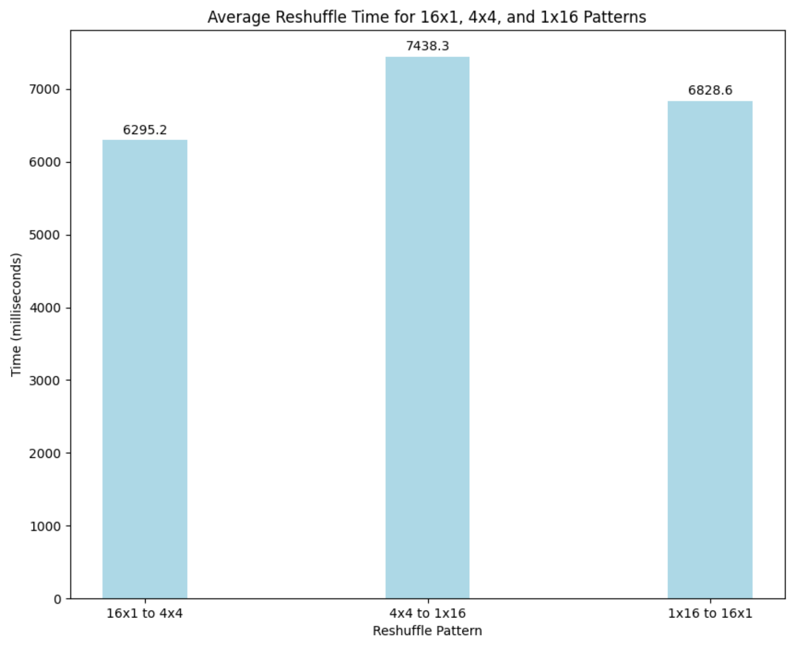
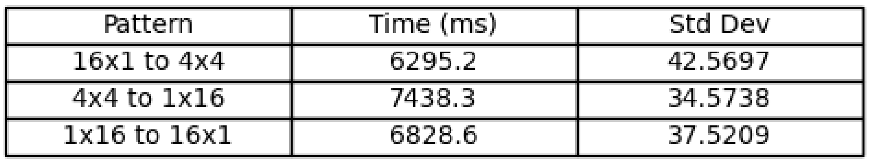


Figure 4.5: Average reshuffling time for layout transitions: 16x1 to 4x4, 4x4 to 1x16, and 1x16 to 16x1.



Pattern	Time (ms)	Std Dev
16x1 to 4x4	6295.2	42.5697
4x4 to 1x16	7438.3	34.5738
1x16 to 16x1	6828.6	37.5209

Table 4.1: Average reshuffling time and standard deviation for layout transitions: 16x1 to 4x4, 4x4 to 1x16, and 1x16 to 16x1, highlighting the reshuffle library's efficiency.

5 Conclusion

This thesis has explored the implementation and evaluation of the Reshuffle library within a distributed heat transfer simulation framework, focusing on the potential benefits of elastic data distribution in high-performance computing (HPC) environments. The study was motivated by the limitations of traditional static data distribution methods, which can hinder performance in dynamic workloads. By leveraging the Reshuffle library, this research aimed to demonstrate the advantages of runtime adaptability in data distribution, particularly in the context of a computationally intensive heat transfer simulation.

The introduction of the Reshuffle library into the heat transfer simulator was achieved through a modular system architecture, which separated core computational logic from parallel distribution mechanisms. This design facilitated the integration of the Reshuffle library, allowing for dynamic data redistribution across multi-node systems. The Jacobi method was employed as the core algorithm for the simulation, chosen for its simplicity and effectiveness in parallel computing environments.

The performance evaluation of the Reshuffle library was conducted through a series of benchmarking experiments, which assessed its efficiency and scalability under various conditions. The results demonstrated that the Reshuffle library effectively managed data distribution across different grid sizes and MPI rank configurations, highlighting its potential to enhance load balancing and resource utilization in distributed computing environments. The experiments also revealed that the library's performance is influenced by the layout configuration, with certain patterns offering more efficient reshuffling times.

In summary, this research has contributed to the growing field of elastic and adaptable HPC frameworks by providing a practical implementation and assessment of the Reshuffle library. This study not only validates the functionality of the Reshuffle library but also offers insights into its potential applications beyond heat transfer simulations, paving the way for further research into elastic data management in HPC.

Future work could explore the integration of the Reshuffle library with other computational frameworks and investigate its performance in different types of simulations. Additionally, further optimization of the library's algorithms and communication patterns could enhance its efficiency and broaden its applicability in diverse computational environments.

Appendix

6 Detailed Descriptions

6.1 List of Figures

6.2 List of Tables

Bibliography

[1] Santiago Narvaez Rivas and Contributors. Reshuffle library. https://gitlab.lrz.de/reshuffle/reshuffle, 2024. Accessed: 2024-11-29.

[2] Abseil Contributors. Abseil c++ common libraries. https://github.com/abseil/abseil-cpp, 2024. Accessed: 2024-11-29.

[3] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Kitware, Inc., 4th edition, 2006. https://vtk.org.

[4] Google. Googletest: Google testing and mocking framework. https://github.com/google/googletest, 2024. Accessed: 2024-12-02.

[5] MPI Forum. Mpi: A message-passing interface standard. https://www.mpi-forum.org/docs/, 2024. Accessed: 2024-11-29.

[6] Martin Schreiber Ahmad Tarraf and Alberto Cascajo. Malleability in modern hpc systems: Current experiences, challenges, and future opportunities. *IEEE Transactions on Parallel and Distributed Systems*, 2024. Accessed: 2024-12-03.

[7] Martin Schreiber Jan Fecht, Howard Pritchard Martin Schulz, and Daniel J.Holmes. An emulation layer for dynamic resources with mpi sessions. *Lecture Notes in Computer Science High Performance Computing. ISC High Performance 2022 International Workshops, 2022*, 2022. Accessed: 2024-12-03.

[8] Lawrence C. Evans. *Partial Differential Equations*, volume 19. American Mathematical Society, 2010.

[9] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. Brooks/Cole, Cengage Learning, Boston, MA, 9th edition, 2010.