Distributed-memory stencil operations with MPI Assignment #6, CSC 746, Fall 2024

Shun Usami[*]
SFSU

Abstract

This study investigates the implementation of distributed-memory stencil operations using the Message Passing Interface (MPI) [3] to enhance computational performance through parallel execution across multiple CPU nodes. The objective was to design and evaluate a distributed Sobel filter for edge detection, employing three decomposition strategies: Row-slab, Column-slab, and Tiled. Custom MPI datatypes were utilized to optimize data exchange, particularly for non-contiguous memory regions, while ensuring scalable parallelization.

Experimental results show that all decomposition strategies improved runtime performance with increasing concurrency. The tiled decomposition strategy demonstrated superior scalability due to its even workload distribution. Additionally, the runtime for scattering and gathering was consistent across methods and concurrency levels, benefiting from the use of subarray datatypes to minimize inter-node communication. These findings underscore the importance of workload balance and communication efficiency in optimizing distributed-memory stencil operations.

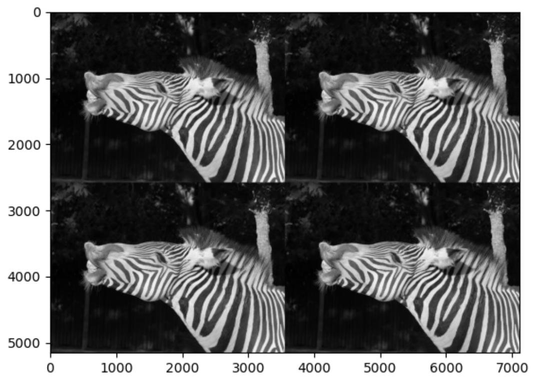
1 Introduction

This assignment explores the problem of implementing distributed-memory stencil operations using the Message Passing Interface (MPI) [3] to enhance computational performance by leveraging parallel execution across multiple CPU nodes. Distributed-memory implementations partition workloads across nodes, each with its own memory space, enabling scalability for larger problems and efficient utilization of computational resources. The primary objective is to design and evaluate an efficient distributed Sobel filter for edge detection that achieves high performance and scalability through effective task partitioning and data communication strategies. Figure 1a presents the original grayscale image used for testing, while Figure 1b shows the resulting image after applying the Sobel filter.

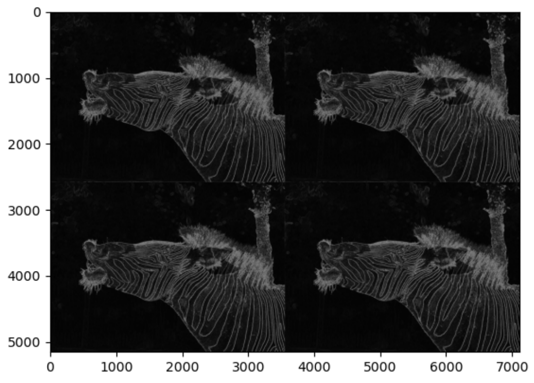
We divided the computational domain using three decomposition strategies: Row-slab, Column-slab, and Tiled. For efficient data exchange, we employed custom MPI datatypes, which were especially crucial when transferring non-contiguous memory regions. These datatypes optimized communication by reducing the number of messages between ranks. The Sobel filter operations were parallelized across nodes to exploit concurrency. Further details about the implementation are provided in Section 2.

The results of our experiments demonstrate that all decomposition strategies improve runtime performance of Sobel filtering with increasing concurrency, but the tiled decomposition strategy exhibits superior scalability thanks to its relatively even workload distribution. Additionally, the runtime for scattering and gathering was consistent across all decomposition methods and concurrency levels due to the use of subarray datatypes, which minimized and optimized inter-node communication. These findings highlight the critical role of balancing workloads and minimizing communication in optimizing distributed-memory stencil operations.

[*]email:susami@sfsu.edu



(a) Original Grayscale Image



(b) Sobel Filter Output

Figure 1: Example results showing the original grayscale image and the Sobel filter output.

2 Implementation

This section describes the key components of the distributed-memory Sobel filter implementation. First, we provide an overview of the task distribution and coordination among MPI ranks in the Overall Code Harness (Section 2.1). Next, we detail the three decomposition strategies employed for dividing the computational domain among ranks: Row-slab, Column-slab, and Tiled decompositions, and how we treated the halo of each tile (Section 2.2). Next, we describe efficient data exchange using custom MPI datatypes in the Send and Receive Strided Buffer (Section 2.4). Finally, the Sobel Implementation (Section 2.5) explains the application of the Sobel operator for edge detection across distributed data.

2.1 Overall Code Harnesss

Listing 1 presents the structure of the distributed-memory stencil operation. Each process computes the mesh decomposition independently to determine how the computational domain is divided among the available ranks. The host process (rank 0) reads the input file and distributes the corresponding data segments (tiles) to all ranks using a scatter operation. Each rank then applies the Sobel filter to its assigned tiles, ensuring an even distribution of computational workload. After processing, the results are sent back to the host process via a gather operation. Finally, the host aggregates the data and writes the output file.

```
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    parseArgs();
    computeMeshDecomposition();
    loadInputFile();   // Only Rank 0 loads
    scatterAllTiles(); // Rank 0 sends, the others
     receive
    sobelAllTiles();
    gatherAllTiles();  // Rank 0 receives, the
    others send
    writeOutputFile(); // Only Rank 0 writes
    MPI_Finalize();
    return 0;
}
```

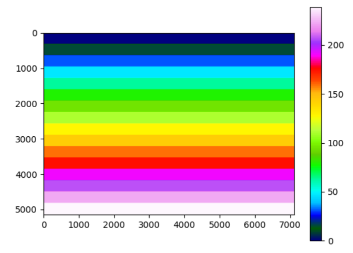
Listing 1: Overall code harness of the distributed-memory stencil operations with MPI.

2.2 Decomposition Strategies

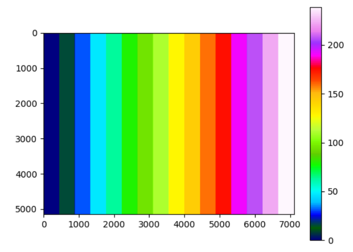
To parallelize the Sobel filter, we implemented three decomposition methods to divide the computational domain among MPI ranks:

- **Row-slab Decomposition:** The input image is divided horizontally into row-wise slabs, and each rank processes one row.
- **Column-slab Decomposition:** The input image is divided vertically into column-wise slabs, and each rank processes one column.
- **Tiled Decomposition:** The input image is divided into smaller rectangular tiles, where each rank processes one tile.

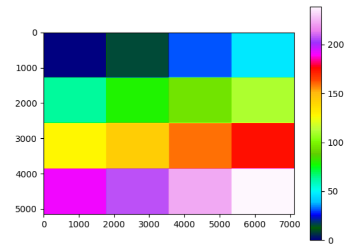
Figure 2 illustrates these decomposition methods using a total of 16 ranks.



(a) Row-slab Decomposition



(b) Column-slab Decomposition



(c) Tiled Decomposition

Figure 2: Graphical representation of the decomposition strategies (Row-slab, Column-slab, and Tiled) using a total of 16 ranks.

2.3 Handling Halo cells

Proper handling of tile boundaries is crucial to ensure the accuracy of our Sobel computation. As depicted in Figure 3, neglecting to process tile boundaries appropriately results in assigning a value of 0.0 to pixels along the shared tile edges.

To address this issue, halo cells are incorporated into the computation. During the scatter phase, halo cells are sent along with the data tiles. These halo cells are then utilized during the Sobel processing phase to ensure seamless computation across tile boundaries. It is important to note that halo cells are not transmitted back during the gathering phase.

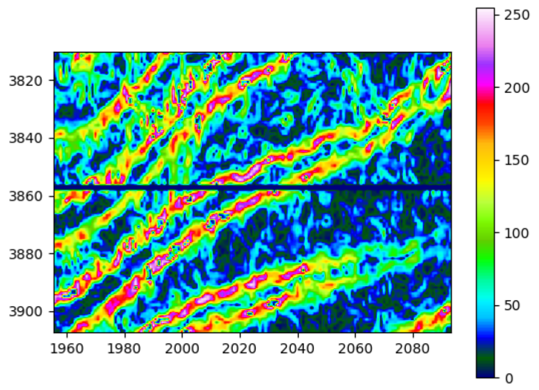


Figure 3: Illustration of tile boundaries without appropriate halo cell processing.

2.4 Send and Receive Strided Buffer

The *sendStridedBuffer* and *recvStridedBuffer* functions are designed to efficiently facilitate data exchange between MPI ranks, particularly for handling non-contiguous memory regions. These functions are crucial for two key operations: scattering input image data from rank zero to non-zero ranks and gathering computed results from non-zero ranks back to rank zero.

2.4.1 Send Strided Buffer

The *sendStridedBuffer* function facilitates the transmission of a subregion from a source buffer (*srcBuf*) on one MPI rank (*fromRank*) to another (*toRank*). As illustrated in Listing 2, this function uses *MPI_Type_create_subarray()* to define a subarray datatype. This datatype specifies the full buffer dimensions, the size of the subregion (*sendWidth* by *sendHeight*), and the offset of the subregion within the buffer (*srcOffsetColumn*, *srcOffsetRow*). By encapsulating the data representation into a single MPI datatype, the function eliminates the need for intermediate buffers, manual data packing, or multiple *MPI_Send()* calls.

```
void sendStridedBuffer(float *srcBuf, int srcWidth
    , int srcHeight, int srcOffsetColumn, int
    srcOffsetRow, int sendWidth, int sendHeight,
    int fromRank, int toRank) {
  int msgTag = 0;

  // Create the subarray datatype
  MPI_Datatype subarray_type;
  int dimensions_full_array[2] = {srcHeight,
    srcWidth};
  int dimensions_subarray[2] = {sendHeight,
    sendWidth};
  int start_coordinates[2] = {srcOffsetRow,
    srcOffsetColumn};
  MPI_Type_create_subarray(2,
    dimensions_full_array, dimensions_subarray,
    start_coordinates, MPI_ORDER_C, MPI_FLOAT, &
    subarray_type);
  MPI_Type_commit(&subarray_type);

  // Send the message
  int count = 1;
  MPI_Send(srcBuf, count, subarray_type, toRank,
    msgTag, MPI_COMM_WORLD);

  // Free the datatype
  MPI_Type_free(&subarray_type);
}
```

Listing 2: Implementation of the send strided buffer function.

2.4.2 Receive Strided Buffer

The *recvStridedBuffer* function receives a subregion of data into a destination buffer (*dstBuf*) from one MPI rank (*fromRank*) to another (*toRank*). As shown in Listing 3 and similar to the *sendStridedBuffer* function described in Section 2.4.1, it uses a custom subarray MPI datatype to define the dimensions and offset of the subregion within the larger buffer. This ensures efficient placement of the received data directly into the appropriate non-contiguous memory region.

```c
void recvStridedBuffer(float *dstBuf, int dstWidth
    , int dstHeight, int dstOffsetColumn, int
    dstOffsetRow, int expectedWidth, int
    expectedHeight, int fromRank, int toRank) {
  int msgTag = 0;
  MPI_Status stat;

  // Create the subarray datatype
  MPI_Datatype subarray_type;
  int dimensions_full_array[2] = {dstHeight,
    dstWidth};
  int dimensions_subarray[2] = {expectedHeight,
    expectedWidth};
  int start_coordinates[2] = {dstOffsetRow,
    dstOffsetColumn};
  MPI_Type_create_subarray(2,
    dimensions_full_array, dimensions_subarray,
    start_coordinates, MPI_ORDER_C, MPI_FLOAT, &
    subarray_type);
  MPI_Type_commit(&subarray_type);

  // Receive the message
  int count = 1;
  MPI_Recv(dstBuf, count, subarray_type, fromRank,
     msgTag, MPI_COMM_WORLD, &stat);

  // Free the datatype
  MPI_Type_free(&subarray_type);
}
```

Listing 3: Implementation of the receive strided buffer function.

2.5 Sobel Implementation

2.5.1 Sobel Operator

The Sobel operator approximates the gradient of image intensity for edge detection [4]. It uses two 3×3 kernels, convolved with the original image A, to compute horizontal and vertical gradients G_x and G_y:

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * A, \quad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$

The gradient magnitude at each pixel is computed as:

$$G = \sqrt{G_x^2 + G_y^2}$$

The function `sobel_filtered_pixel()` computes the Sobel filter at a pixel location, as shown in Listing 4. The arrays *gx* and *gy* represent the kernel weights for the horizontal and vertical filters, respectively.

```
Function sobel_filtered_pixel(img[width, height],
    x, y, gx[3, 3], gy[3, 3]) {
    if x or y is at the boundary of the img
        return 0
    Gx = 0.0, Gy = 0.0;
    for j in {0, 1, 2}
        for i in {0, 1, 2}
            xx = x - 1 + i;
            yy = y - 1 + j;
            Gx += img[xx, yy] * gx[i, j];
            Gy += img[xx, yy] * gy[i, j];
    return sqrt(Gx * Gx + Gy * Gy);
}
```

Listing 4: Sobel filtered pixel computation. Computes the Sobel filter at a specific pixel location.

2.5.2 Overall Sobel Filtering

The implementation of `do_sobel_filtering()` function is shown in Listing 5, which applies the Sobel filter to a specified region of an input buffer (`in`) and stores the processed output in a designated output buffer (`out`). This function processes each pixel in the region by invoking the `sobel_filtered_pixel()` function (detailed in Listing 4), which calculates the gradient magnitude for the pixel using the Sobel operator. The computed values are then placed into their corresponding positions in the output buffer. Each rank independently calls `do_sobel_filtering()` to compute the filtered output for its assigned tile, ensuring parallel processing across the distributed tiles.

```c
void do_sobel_filtering(float *in, float *out, int
     width, int height) {
  // Define Sobel operator kernels
  float Gx[] = {1.0, 0.0, -1.0, 2.0, 0.0, -2.0,
    1.0, 0.0, -1.0};
  float Gy[] = {1.0, 2.0, 1.0, 0.0, 0.0, 0.0,
    -1.0, -2.0, -1.0};

  // Apply Sobel filtering to each pixel
  for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
      // Compute the Sobel-filtered value for the
    current pixel
      out[y * width + x] =
          sobel_filtered_pixel(in, x, y, width,
    height, Gx, Gy);
    }
  }
}
```

Listing 5: Implementation of the Sobel operation for a given region.

3 Results

In this section, we analyze how our distributed-memory implementation of the Sobel filter improves computational efficiency by leveraging parallel execution across multiple CPU nodes. We evaluate the performance of three grid decomposition strategies—Column-slab, Row-slab, and Tiled—under various concurrency levels. We start by describing the computational platform and software environment we used for our experiments. Next, we explain our methodology, including decomposition techniques, runtime measurement, data movement tracking, and speedups to assess parallel performance. Through our runtime performance and data movement studies, we identify how task distribution, communication, and synchronization influence overall efficiency. Finally, we discuss our findings and suggest optimizations to enhance scalability and performance accuracy.

3.1 Computational platform and Software Environment

3.1.1 System Overview

The experiments were conducted on four CPU nodes, running *SUSE Linux Enterprise Server 15 SP4*, with kernel version *5.14.21-150400.24.81_12.0.87-cray_shasta_c*, of the Perlmutter supercomputer at the National Energy Research Scientific Computing Center (NERSC). Each node is equipped with two AMD EPYC 7763 (Milan) CPUs.

3.1.2 CPU Specifications

The AMD EPYC 7763 processor features 64 cores running at a clock rate of 2.45 GHz. It supports Simultaneous Multi-threading (SMT), enabling two threads per core. Each core has a 32 KiB L1 cache and a 512 KiB L2 cache, while every 8 cores share a 32 MiB L3 cache. The processor also offers 8 memory channels per socket, with 2 DIMMs per channel, and 4 NUMA domains per socket (NPS=4). The system is supported by 256 GiB of DDR4 DRAM, providing a CPU memory bandwidth of 204.8 GiB/s [1, 5].

3.1.3 Software Environment

The computational environment is based on a Linux operating system optimized for high-performance computing. The `g++` compiler from the GNU Compiler Collection (gcc) version 12.3.0 (SUSE Linux) was used for compiling the code. The source code was compiled with the `-O3` optimization flag, which enables aggressive compiler optimizations such as loop unrolling, vectorization, and inlining, ensuring efficient utilization of hardware resources.

We employed the Message Passing Interface (MPI) standard [3] for parallel programming. Specifically, we used the HPE Cray MPI implementation [2], version 8.1.28.

3.2 Methodology

3.2.1 Problem Sizes and Concurrency Levels

To evaluate the Sobel filter implementations, we tested a fixed grayscale image with dimensions 7112×5146. Experiments were conducted using three decomposition methods: Row-slab, Column-slab, and Tiled. For each method, we analyzed performance across varying total ranks: 4, 9, 16, 25, 36, 49, 64, and 81, capturing the impact of concurrency on runtime.

3.2.2 Runtime Measurement

Performance data was collected by measuring the runtime of three key phases: Scatter, Sobel computation, and Gather. Timing was performed on the CPU using the C++ `chrono_timer()`, which encapsulated the `scatterAllTiles`, `sobelAllTiles`, and `gatherAllTiles` functions, respectively.

To ensure accurate synchronization across all ranks, `MPI_Barrier` was employed before starting and stopping timers for each phase. This approach guarantees that all ranks finish the preceding phase before advancing to the next, thereby isolating the runtime measurements for each phase and avoiding overlap between operations.

3.2.3 Data Movement

The number of messages and total data size were measured by implementing counters before calling `MPI_Send` and calculating the data size using `MPI_Type_size`. These values were aggregated using `MPI_Reduce` to provide a global summary.

3.2.4 Speedup Calculation

Speedup was derived from the runtime data to evaluate the parallel performance of the Sobel filter implementations. The speedup $S(n, p)$ is defined as:

$$S(n,p) = \frac{T^*(n)}{T(n,p)}$$

Here, $T^*(n)$ is the runtime of the best sequential algorithm for a problem of size n, measured as the runtime with a single process per CPU node (total ranks of 4). $T(n, p)$ represents the runtime for the same problem size n using p parallel threads. Since the problem size was fixed (7112×5146 image), the variability in p allowed us to assess the scalability of each decomposition method.

3.3 Runtime Performance Study

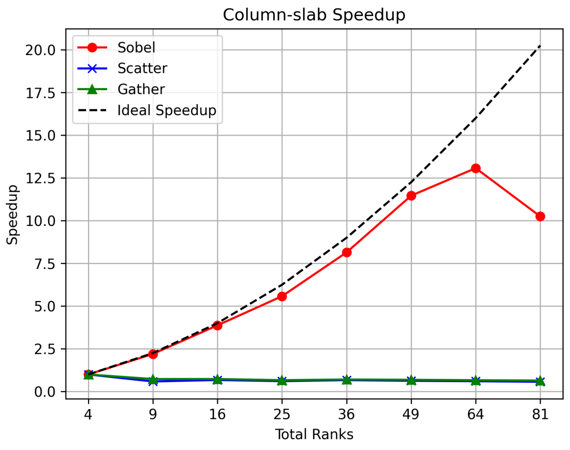


Figure 4: Speedup chart for Column-slab decomposition.

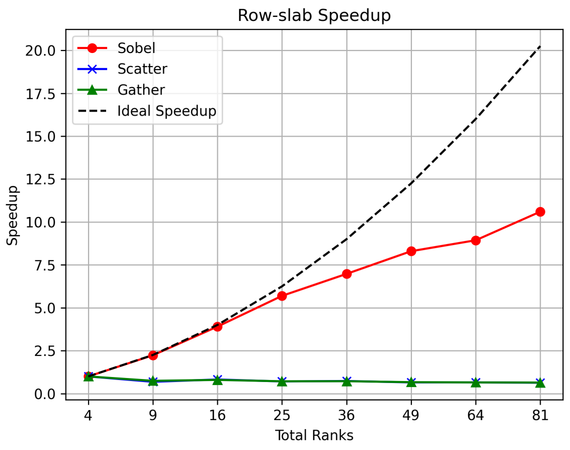


Figure 5: Speedup chart for Row-slab decomposition.

In this section, we analyze the runtime performance of the Sobel filter using three grid decomposition strategies: Column-slab, Row-slab, and Tiled decomposition. The speedup charts for these strategies, shown in Figures 4, 5, and 6, illustrate the speedup of runtimes for the three computational stages: scatter, process, and

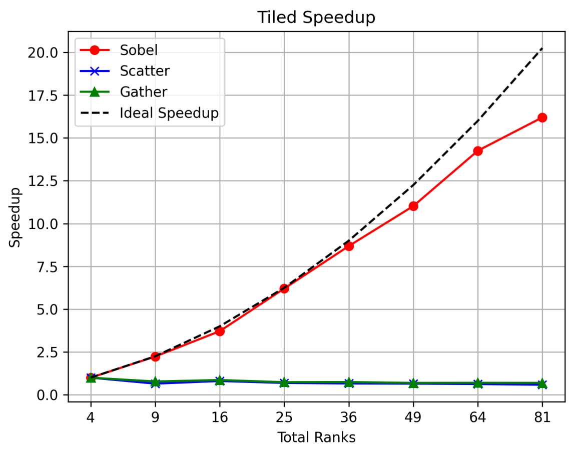


Figure 6: Speedup chart for Tiled decomposition.

gather. Below, we describe the observed performance characteristics for each strategy and discuss the underlying reasons.

As shown in Figure 4, the Column-slab decomposition achieved consistent speedup up to a total rank of 49, nearing ideal performance. However, the speedup gains plateaued at a total rank of 64, and a performance degradation was observed at a total rank of 81.

In contrast, as illustrated in Figure 5, the Row-slab decomposition showed consistent speedup with increasing total ranks, although it began to deviate slightly from ideal performance starting at a total rank of 36.

As shown in Figure 6, the Tiled decomposition consistently achieved near-ideal speedup, maintaining strong performance up to a total rank of 64. Unlike the Column-slab, it did not exhibit performance degradation at higher ranks.

The performance behavior of the Column-slab decomposition was initially expected to outperform the others due to its narrower tile width, which enhances cache efficiency when transitioning between rows. While it did show better speedup than Row-slab for lower ranks, the unexpected performance drop at a total rank of 81 may be attributed to inherent load imbalances. For instance, as shown in Table 2 and 2, the base grid size for 64 ranks is 111×5146, with a last tile size of 119×5146. In contrast, the base grid size for 81 ranks is 87×5146, with a last tile size of 152×5146. This disparity arises because the current implementation uses fixed dimensions for all tiles except the last one, causing the last tile to be disproportionately larger. This imbalance likely limited overall performance. To address this, a more balanced tile computation scheme should be implemented to ensure uniform tile sizes.

The superior performance of the Tiled decomposition is likely due to its more evenly distributed tile sizes, which mitigate the impact of slower ranks on runtime. As the runtime is ultimately constrained by the slowest rank, reducing disparities in tile sizes directly improves performance.

The scatter and gather runtimes exhibited similar performance across all decomposition strategies. While their speedup dropped by approximately 30% when increasing the total rank from 4 to 9, the performance stabilized for higher ranks. This consistency can be attributed to the use of subarray datatypes, which optimize communication by transforming data before transmission. By sending subregions of data in a single operation instead of row-by-row transfers, the number of communication events between ranks is minimized, leading to nearly identical scatter and gather performance across all methods.

Rank	**Column-slab**	**Row-slab**	**Tiled**
4	1778 x 5146	7112 x 1286	3556 x 2573
9	790 x 5146	7112 x 571	2370 x 1715
16	444 x 5146	7112 x 321	1778 x 1286
25	284 x 5146	7112 x 205	1422 x 1029
36	197 x 5146	7112 x 142	1185 x 857
49	145 x 5146	7112 x 105	1016 x 735
64	111 x 5146	7112 x 80	889 x 643
81	87 x 5146	7112 x 63	790 x 571

Table 1: Base grid size of the tile for each decomposition method.

Rank	**Column-slab**	**Row-slab**	**Tiled**
4	1778 x 5146	7112 × 1288	3556 × 2573
9	792 x 5146	7112 × 578	2372 × 1716
16	452 x 5146	7112 × 331	1778 × 1288
25	296 x 5146	7112 × 226	1424 × 1030
36	217 x 5146	7112 × 176	1187 × 861
49	152 x 5146	7112 × 106	1016 × 736
64	119 x 5146	7112 × 106	889 × 645
81	152 x 5146	7112 × 106	792 × 578

Table 2: Grid size of the last tile for each decomposition method.

3.4 Data Movement Performance Study

In this section, we analyze the data movement performance for different grid decomposition strategies (Column-slab, Row-slab, and Tiled) in terms of the number of messages sent and the total data moved between ranks during the scatter and gather phases. Table 3 summarizes the number of messages and the total data moved for varying levels of concurrency.

The observed data movement is consistent across all decomposition methods due to the use of subarray data types for inter-rank communication. This approach minimizes the number of messages sent, as subarrays allow the transmission of contiguous subregions of data in a single operation rather than row-by-row communication.

Ranks	**Column-slab**		**Row-slab**		**Tiled**	
	Count	**Bytes**	**Count**	**Bytes**	**Count**	**Bytes**
4	6	220MB	6	220MB	6	220MB
9	16	261MB	16	261MB	16	260MB
16	30	275MB	30	275MB	30	275MB
25	48	282MB	48	282MB	48	281MB
36	70	286MB	70	287MB	70	285MB
49	96	289MB	96	290MB	96	287MB
64	126	291MB	126	292MB	126	289MB
81	160	292MB	160	294MB	160	290MB

Table 3: Number of messages sent and total data moved between ranks during the scatter and gather phases.

3.5 Overall Findings and Discussion

The amount of data moved remains relatively consistent across decomposition strategies, likely due to the use of the subarray data type. This approach appears to achieve near-optimal data movement efficiency. To further improve the runtime performance of the Sobel operation, it is recommended to ensure that the grid size of all tiles is as evenly distributed as possible. The current approach, where all tiles have the same size except the last one, can lead to performance bottlenecks, as the overall runtime is constrained by the rank with the longest execution time.

Potential sources of errors or inaccuracies in performance measurement, particularly in runtime and data movement, include the timing methodology employed in rank 0. While using `MPI_Barrier` ensures that all ranks complete the preceding code region before proceeding, it does not guarantee synchronization in the actual execution of subsequent operations across all ranks. This lack of strict synchronization may lead to discrepancies in the recorded execution times.

Acknowledgments

This paper was edited with the assistance of *ChatGPT (GPT-4o and GPT-4o with canvas)* (accessed November 2024), which was primarily used for correcting grammatical errors and rephrasing. This research was supported by resources from the National Energy Research Scientific Computing Center (NERSC), a Department of Energy Office of Science User Facility, under NERSC award DDR-ERCAP m3930 for 2024.

References

[1] AMD. HPC Tuning Guide for AMD EPYC 7003 Series Processors. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/high-performance-computing-tuning-guide-amd-epyc7003-series-processors.pdf, 2022. Accessed: November 2024.

[2] H. P. Enterprise. Hpe cray mpi: Implementation of the mpi standard. `https://cpe.ext.hpe.com/docs/latest/mpt/mpich/`. Accessed: November 2024.

[3] M. Forum. Message passing interface (mpi): A standard for parallel computing. `http://www.mpi-forum.org`. Accessed: November 2024.

[4] N. Kanopoulos, N. Vasanthavada, and R. L. Baker. Design of an image edge detection filter using the sobel operator. *IEEE Journal of solid-state circuits*, 23(2):358–367, 1988.

[5] NERSC. Perlmutter Architecture. https://docs.nersc.gov/systems/perlmutter/architecture/, 2024. Accessed: November 2024.