Performance Analysis of MPI-Based Sobel Filter Using Varied Grid Decomposition Strategies
CP#5, CSC 746, Fall 2023

Ekarat Buddharuksa*
San Francisco State University

Abstract

This study investigates the performance of an MPI-based Sobel filter application using row-slab, column-slab, and tiled grid decomposition strategies. By analyzing runtime speedup and communication overhead across various concurrency levels, the research identifies key differences in efficiency among the strategies. Primary findings indicate that while data movement remains consistent, the tiled strategy excels in scalability and efficiency, particularly at higher concurrency levels. This highlights the significance of grid decomposition choice and communication optimization in distributed computing environments.

1 Introduction

In the modern landscape of distributed computing, the necessity for effective data processing methods is becoming increasingly evident. This is particularly true in the context of grid decomposition, which includes specific techniques like row-slab, column-slab, and tile decomposition. Our report takes a detailed look at the `mpi_2dmesh.cpp` program, showcasing its application of these grid decomposition methods alongside the use of Message Passing Interface (MPI) in distributed systems. The focus is on understanding how the program's design influences the efficiency and scalability of operations in distributed computing environments.

Our approach to this study involves an in-depth examination of the `mpi_2dmesh.cpp` program's structure, with a special emphasis on its implementation of various grid decomposition strategies. These strategies are critical in enhancing the performance of large-scale data handling tasks within distributed systems. We analyze how different grid decomposition methods, namely row-slab, column-slab, and tile decomposition, affect the overall performance of distributed computing tasks. This includes a focused exploration of mesh-based data structures' processing and management.

The results of our study shed light on the interconnected relationship between the different grid decomposition strategies and their impact on distributed computing performance. The `mpi_2dmesh.cpp` program, through its implementation of these decomposition methods, demonstrates considerable improvements in data processing efficiency. These improvements are achieved by optimizing grid decomposition techniques and effectively utilizing MPI for communication. Our findings offer insights into the advantages and limitations of each grid decomposition strategy, providing valuable guidance for the design and enhancement of high-performance computing applications in similar contexts.

2 Implementation

2.1 Overall Code Harness

The code harness of the `mpi_2dmesh.cpp` program is engineered to efficiently manage the distribution and processing of data in a distributed computing environment using MPI (Message Passing Interface). At the core of the harness is the `main` function, which initiates MPI and orchestrates the program flow. Key steps in the `main` function include:

1. **Initialization**: MPI is initialized, and each process identifies its rank and the total number of ranks. Command-line arguments are parsed to set up the application state (`AppState`).

2. **Mesh Decomposition**: Based on the application state, a mesh decomposition is computed, determining how the data will be divided among the MPI processes.

3. **Data Distribution and Collection**: The program undergoes phases of scattering, processing, and gathering data. In the scatter phase, data is distributed among nodes. Each node then performs the Sobel operation on its data segment in the processing phase. Finally, the gather phase collects the processed data back to the root node.

4. **Timing and Reporting**: The program measures the time taken for the scattering, Sobel computation, and gathering phases, offering insights into the performance of the distributed processing.

This harness demonstrates an efficient use of MPI for handling complex data operations in a distributed environment, focusing on optimizing each phase of the computation, from initialization to data collection and processing.

2.2 sendStridedBuffer Implementation

The `sendStridedBuffer` function is integral to our program's inter-node communication, efficiently orchestrating the data exchange across MPI nodes while adhering to the predetermined grid decomposition strategy. This function specifically addresses the challenge of sending a subregion of a two-dimensional data buffer, where the subregion dimensions and its offset within the larger buffer are dynamically determined based on the program's current execution context.

```
1  void sendStridedBuffer(float *srcBuf,
2        int srcWidth, int srcHeight,
3        int srcOffsetColumn, int srcOffsetRow,
4        int sendWidth, int sendHeight,
5        int fromRank, int toRank )
6  {
7     int msgTag = 0;
8     MPI_Datatype SendData;
9     MPI_Type_vector(sendHeight, sendWidth, srcWidth
       , MPI_FLOAT, &SendData);
10    MPI_Type_commit(&SendData);
11    int offset =  srcOffsetRow * srcWidth +
       srcOffsetColumn;
12    MPI_Send(srcBuf + offset, 1, SendData, toRank,
       msgTag, MPI_COMM_WORLD);

14    messageCount++;
```

*email:ebuddharuksa@sfsu.edu

```
    dataMovement += sendWidth * sendHeight * sizeof
     (float);
    MPI_Type_free(&SendData);
}
```

Listing 1: sendStridedBuffer function implementation

By defining a custom MPI datatype, `SendData`, the function encapsulates a subregion of the source buffer for sending. The datatype, created with `MPI_Type_vector`, represents the strided layout of the data. After calculating the offset, `MPI_Send` is used to transmit this data to the target rank.

The function also updates communication metrics by incrementing `messageCount` and adding to `dataMovement`, which are essential for analyzing communication efficiency. Resource management is finalized with `MPI_Type_free`, ensuring the proper deallocation of the custom datatype.

2.3 recvStridedBuffer() Implementation

The `recvStridedBuffer()` function manages the receipt of strided data segments in our MPI-driven application. It is tasked with receiving subregions of data from a particular rank with a specified destination buffer while accounting for the expected dimensions and positional offsets.

```
void recvStridedBuffer(float *dstBuf,
      int dstWidth, int dstHeight,
      int dstOffsetColumn, int dstOffsetRow,
      int expectedWidth, int expectedHeight,
      int fromRank, int toRank ) {

    MPI_Datatype RecvData;
    MPI_Type_vector(expectedHeight, expectedWidth,
     dstWidth, MPI_FLOAT, &RecvData);
    MPI_Type_commit(&RecvData);
    MPI_Recv(dstBuf + (dstOffsetRow * dstWidth +
     dstOffsetColumn),
             1, RecvData, fromRank, 0,
     MPI_COMM_WORLD, &stat);

    dataMovement += expectedWidth * expectedHeight
     * sizeof(float);
    messageCount++;
    MPI_Type_free(&RecvData);
}
```

Listing 2: recvStridedBuffer function implementation

A specialized MPI datatype, `RecvData`, is defined to encapsulate the expected two-dimensional subregion for receipt. The `MPI_Type_vector` function facilitates the reception of non-contiguous data blocks that conform to the strided configuration of the destination buffer. After committing the datatype, the function utilizes `MPI_Recv` to intake data from the sending rank, accurately placing it within the destination buffer at the predetermined offset.

The function also updates communication metrics like in `sendStridedBuffer` function

2.4 Sobel Filter Implementation

The implementation of the Sobel filter is designed to apply the Sobel operator to an input image to highlight edges by approximating the gradient of the image intensity. The parallelism is introduced at the point of iterating over the image pixels, where each pixel's value is calculated independently of the others, making it an ideal candidate for concurrent computation. Detailed pseudocodes are in the following.

Algorithm 1 sobel_filtered_pixel function

function SOBEL_FILTERED_PIXEL($s, i, j, ncols, nrows, gx, gy$)
    Initialize gradient sum for x-direction
    Initialize gradient sum for y-direction
    for $ki \leftarrow -1$ **to** 1 **do** ▷ Kernel offsets for x-direction
        for $kj \leftarrow -1$ **to** 1 **do** ▷ Kernel offsets for y-direction
            Compute neighbor's x-coordinate
            Compute neighbor's y-coordinate
            if Neighbor is not out of bound **then**
                compute and accumulate Gx,Gy
            end if
        end for
    end for
    return $\sqrt{Gx^2 + Gy^2}$ ▷ Compute and return gradient magnitude
end function

Algorithm 2 do_sobel_filtering

$Gx \leftarrow [1,0,-1,2,0,-2,1,0,-1]$
$Gy \leftarrow [1,2,1,0,0,0,-1,-2,-1]$
for each pixel position (i, j) in the output image **do**
    call sobel_filtered_pixel function at position (i, j)
    Save the filtered value to the output image
end for

3 Evaluation

3.1 Computational Platform and Software Environment

Our computational experiments were conducted on the Perlmutter system [1] at NERSC. For our experiments, we specifically utilized the CPU nodes of the Perlmutter system. Each of these nodes is equipped with two AMD EPYC 7763 (Milan) CPUs, with each CPU boasting 64 cores. These processors support the AVX2 instruction set and have an L3 cache of 256MB. The nodes have a total of 512 GB DDR4 memory, providing a memory bandwidth of 204.8 GB/s per CPU. In terms of performance, each core can achieve 39.2 GFlops, leading to a total of 2.51 TFlops per socket. The Perlmutter system operates on a specialized version of the Cray Linux Environment.

For MPI communication, we employed the Cray MPICH implementation, which is optimized for high performance on the Perlmutter system. This MPI library is tailored to the hardware, leveraging the high-speed network interconnects and advanced communication features of the system, ensuring efficient message passing and synchronization between the compute nodes.

3.2 Methodology

Our methodology for assessing the performance of the MPI-based Sobel filter involved a series of tests under varying conditions.
Performance Tests: We executed the Sobel filter across different concurrency levels and grid decomposition strategies. The concurrency levels tested were 4, 9, 16, 25, 36, 49, 64, and 81, distributed across 4 CPU nodes of the Perlmutter system. The grid decomposition strategies employed were row-slab, column-slab, and tiled.
Performance Metrics: The key performance metrics collected were as follows:

- **Runtime**: We measured the elapsed time for the scatter, process, and gather stages. The timing was recorded on rank 0 after all MPI ranks completed their respective activities. This method provides an accurate measure of the processing time for each stage.

- **Number of Messages and Data Movement**: We calculated the number of messages and the total amount of data moved between ranks. This was done by analyzing the size and number of tiles being processed, as well as reviewing the implementations of `sendStridedBuffer()` and `recvStridedBuffer()`. While there are more complex methods for runtime instrumentation in MPI, our approach relied on static analysis and calculation based on tile size and count.

This methodology was designed to comprehensively evaluate the performance of the MPI-based Sobel filter under different operational conditions, providing insights into the efficiency of different grid decomposition strategies and concurrency levels.

3.3 Runtime Performance Study

Performance metrics were collected for various concurrency levels using three grid decomposition strategies: row-slab, column-slab, and tile. The speedup of runtime was observed across different stages: scatter, process (Sobel), and gather.

In the scatter stage, all strategies showed a decrease in speedup as concurrency increased, with the row-slab strategy exhibiting the most gradual decline. The tile strategy maintained a superior speedup at lower concurrency levels but experienced a significant reduction at a concurrency level of 9, as shown in Figure 1.

The process stage revealed a marked improvement in speedup for all strategies with increasing concurrency levels, particularly noticeable for the tile strategy at the highest concurrency level of 81. Row-slab and column-slab strategies displayed similar performance until a concurrency level of 64, after which the tile strategy's speedup surged ahead, as illustrated in Figure 2.

For the gather stage, the tile strategy achieved the highest speedup at lower concurrency levels, suggesting a more efficient reassembly process. Conversely, the row-slab strategy demonstrated a consistent decrease in speedup as concurrency levels rose, indicating potential data collection bottlenecks, depicted in Figure 3.

The analysis indicates that the performance of the Sobel filter application is highly dependent on the selected grid decomposition strategy. The tile strategy appears to offer better scalability in processing at high concurrency levels, while the row-slab strategy provides more stable performance in the scatter stage. These findings underscore the importance of strategic selection in grid decomposition to optimize the performance of MPI-based applications.

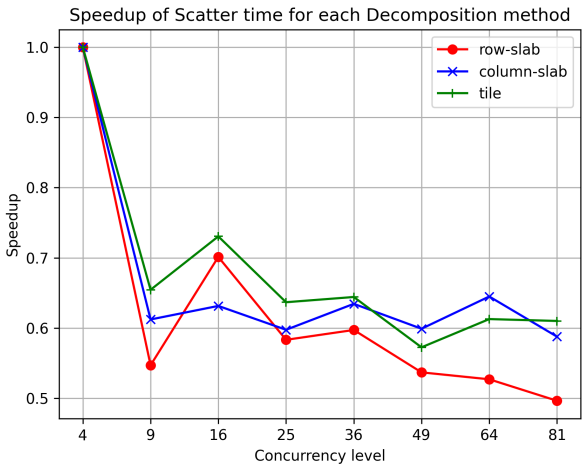

Figure 1: Speedup of Scatter time for each Decomposition method.

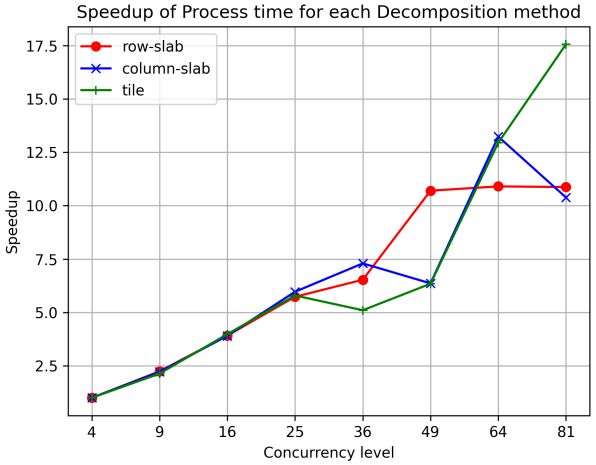

Figure 2: Speedup of Process time for each Decomposition method.

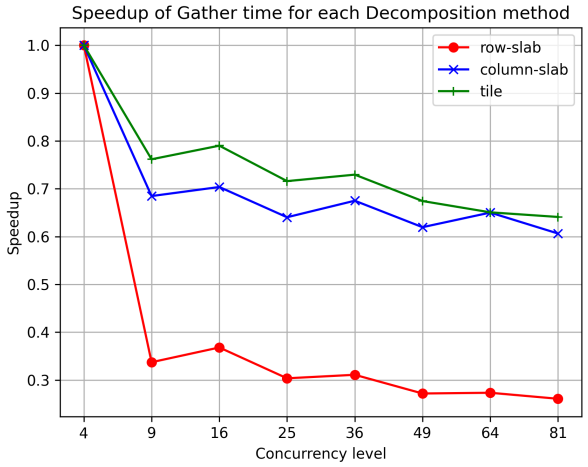

Figure 3: Speedup of Gather time for each Decomposition method.

3.4 Communication Efficiency

The following table summarizes the communication efficiency in terms of the number of messages sent and the total amount of data moved between all ranks during both scatter and gather phases.

Concurrency Level	# Messages	Data Moved (MB)
4	6	219
9	16	260
16	30	274
25	48	281
36	70	284
49	96	286
64	126	288
81	160	289

Table 1: Row-Slab decomposition: number of messages and total data moved at different concurrency levels.

Concurrency Level	# Messages	Data Moved (MB)
4	6	219
9	16	260
16	30	274
25	48	281
36	70	284
49	96	286
64	126	288
81	160	289

Table 2: Column-Slab decomposition: number of messages and total data moved at different concurrency levels.

Concurrency Level	# Messages	Data Moved (MB)
4	6	219
9	16	260
16	30	274
25	48	281
36	70	284
49	96	286
64	126	288
81	160	289

Table 3: Tiled decomposition: number of messages and total data moved at different concurrency levels.

Analyzing the communication overhead for row-slab, column-slab, and tiled grid decompositions reveals a direct relationship between increasing concurrency levels and the number of messages and data transferred. As concurrency grows, so does the communication load, regardless of the decomposition method. This increase is linear for message counts and less so for data volume, hinting at a saturation effect due to the fixed size of the dataset being processed.

When we consider how this overhead might affect the scatter, process, and gather phases of execution, we anticipate that the increased messaging and data movement could lead to longer runtimes for the scatter and gather phases. However, the processing phase may not experience such a linear increase in runtime thanks to the benefits of parallel processing.

Despite the similar trends in communication metrics, the tiled strategy suggests potential for greater efficiency, possibly due to underlying optimizations not apparent from the communication data alone. This pattern highlights the importance of efficient communication protocols, especially at higher concurrency levels where they can significantly affect performance. The processing phase seems to benefit from increased concurrency, underscoring the need for detailed runtime analysis to understand the full impact of communication overhead on performance.

4 Overall Findings and Discussion

Integrating the speedup data for scatter, gather, and process times into our performance analysis reveals distinct insights into the efficiency of different grid decomposition strategies. In scatter and gather stages, a decrease in speedup with increased concurrency levels was observed across all strategies. This pattern suggests that communication overhead becomes more significant at higher concurrency levels, particularly affecting the row-slab and column-slab strategies. However, the tiled strategy demonstrated a relatively higher efficiency in data reassembly during the gather phase, pointing to its potential advantage in handling communication overhead.

In the process stage, the rise in speedup at higher concurrency levels across all strategies indicates effective parallel processing. Notably, the tiled strategy excelled at the highest concurrency, suggesting superior scalability and efficient utilization of parallel computation resources.

While the total amount of data moved was consistent across strategies, the increased message count with higher concurrency levels highlights communication efficiency as a pivotal factor affecting runtime performance. To enhance this, optimizing the balance between data chunk size and the number of messages could be crucial, potentially by adopting non-blocking communication methods to reduce the impact of communication delays.

The accuracy of performance measurements is another critical aspect. Variations in system load and network behavior could affect the timing and data movement metrics, underscoring the need for robust and precise measurement techniques.

Overall, the findings suggest that while data movement is consistent, the choice of grid decomposition strategy significantly impacts communication efficiency and computational scalability. The tiled decomposition, in particular, shows promise in balancing these aspects, especially at higher concurrency levels. This analysis emphasizes the need for strategic optimizations in communication and computation to enhance the overall performance of distributed computing applications.

References

[1] NERSC. Perlmutter architecture. https://docs.nersc.gov/systems/perlmutter/architecture/. Accessed: 2023-10-1.