Performance analysis of Distributed-Memory Parallelization of Sobel Filter Using MPI
Assignment #6, CSC 746, Fall 2023

Malavya Raval*
SFSU

ABSTRACT

Utilizing the Sobel filter algorithm with implement sending halo cells so that the distributed memory Sobel computation is actually correct at tile boundaries, this study employs the MPI library to investigate the performance of three decomposition strategies: row, column, and tile. The elapsed time for each phase (scatter, processing, and gather) is recorded, and the study explores the impact of varying levels of concurrency. Comparing performance parameters such as speedup, messages transfered and data movement, the results reveal that tile decomposition outperforms row and column strategies slightly. Moreover, the scatter time exhibits an increase with higher concurrency levels, while the processing reduces exponentially at until it comes to a constant, this leads to increase in speedup at a significant amount.

1 INTRODUCTION

In this study of Sobel filter algorithm, particularly in large-scale image processing, we use an efficient parallel implementation along with implementation of sending halo cells so that the distributed memory Sobel computation is actually correct at tile boundaries. Traditional, sequential execution may prove inadequate for massive datasets, urging the exploration of parallel computing paradigms to enhance processing speed and scalability. We will be using the Message Passing Interface Library for the purpose of this study.

The approach involves scattering of data, Sobel filter processing on individual ranks, and a subsequent gathering of results, this is done through MPI functions. The study encompasses variations in concurrency levels which are 4, 9, 16, 25, 36, 49, 64, 81 and then running these levels of concurrencies across 4 CPU nodes and different grid decomposition strategies, namely row-slab, column-slab, and tiled, to comprehensively evaluate the algorithm's performance across diverse computational scenarios. The send strided buffer and receive strided buffer functions are invoked to call and send each of the ranks. The location of the stride is calculated using the offset value and the height or width of the decomposition (height for row and width for column). For tile decomposition tiles are made and location is calculated accordingly. The performance in terms of runtime and data movement is calculated and the results are compared and evaluated. The halo cells are sent for addictional computation so that the ghost cells are not created and we get a perfect image without any "seams".

Upon testing with varying concurrency levels and grid decomposition strategies, some observations were made that the tile decomposition strategy proves marginally superior in terms of runtime compared to row and column strategies. along with that, the study unveils the a relationship between concurrency levels and Sobel filter processing time, showcasing an initial exponential reduction before reaching a stable phase. The scatter time increases with increase in the concurrency. The findings provide a detailed understanding of the algorithm's behavior, offering valuable guidance for optimizing large-scale image processing with the MPI-based Sobel filter.

*email:mrava@sfsu.edu

2 IMPLEMENTATION

In this study, we explore three distinct implementations of the Sobel filter algorithm. The first implementation employs MPI with Row Decomposition, the second utilizes Column Decomposition, and the third adopts Tile Decomposition. In each implementation there is some modifications done so that halo cells are also sent for computation so that there are no "seams" in the processed image that is generated. Subsequently, we provide a comprehensive overview of each implementation, offering detailed insights into the code harness that underlies the entire experimentation.

2.1 Overall Code Harness

The overall code harness for this MPI-based framework is designed to facilitate processing of the Sobel filter algorithm. The code begins by loading input data from a file, and the type of decomposition whether row, column, or tile is determined through command line arguments or a bash script. The implementation iterates through the y-axis of the computational mesh, creating a new Tile2D object for each row of tiles. The Tile2D object encapsulates the location, size, and ghost zone information for each tile. The ghost zone information is updated for each tile to ensure that it has access to the necessary data from neighboring tiles. The updated tiles are then added to the tileArray vector. The code is modified to take one extra row of tiles for computing output and then updating it iteratively, this is done through incrementing ghost_xmax, ghost_ymax when required. Similarly the ghost_ymin and ghost_xmin are decremented when there is a need to get data from its neighbouring tile.

The framework then constructs a 2D array of Tile objects based on the selected domain decomposition strategy. Subsequently, the scatter phase is executed, distributing the data across MPI ranks, a task efficiently accomplished by the send strided buffer function. Following the scatter phase, the Sobel filtering takes place during the processing phase. Lastly, the gather phase, facilitated by the receive strided buffer function, consolidates the processed data. All input data is centralized to rank 0, and the final output is aggregated and written from rank 0. The Rank 0 reads the image at the start of the program, and writes the result at the end of it, this is the reason for every implementation that Rank 0 is observed at the end of all the ranks. This streamlined code structure ensures a systematic and parallelized execution of the Sobel filter across multiple MPI ranks.

2.2 Send Strided buffer

The sendStridedBuffer function tackles a crucial aspect of our MPI-based Sobel filter implementation – the transmission of data across MPI ranks. The problem here is the need to distribute specific subregions of the source buffer (srcBuf) to different ranks for parallel processing. To address this, the code employs MPI_Send, strategically sending sendWidth and sendHeight from one rank to another.

The code incrementally updates the messagesNum and dataMoved counters, which are metrics for tracking the communica-

tion efficiency of the distributed Sobel filter algorithm and it will be analized ahead. The use of MPI data types, exemplified by the MPI_Type_vector function, creates a specialized data structure for the strided buffer. This allows for a seamless transmission of the row, column or the tile data, considering both its dimensions and the specified offset, ensuring the correct portion of the source buffer is transmitted. The MPI_Send function is then employed to dispatch the strided buffer to the designated rank.

This approach ensures not only the precision of data transmission but also optimizes the overall communication within diverse decomposition strategies. The relationship of this code to the larger problem is fundamental, as it establishes a robust framework for seamless data sharing among MPI ranks, a pivotal element in parallel Sobel filtering. The proper execution of the sendStridedBuffer logic ensures that each MPI rank receives the requisite data, fostering the collaborative and distributed nature of the Sobel filter algorithm across the computational program.

```
1  messagesNum++;
2  dataMoved += sendHeight * sendWidth * sizeof(
    srcBuf);

4  int msgTag = 0;

6  MPI_Datatype result;
7  MPI_Type_vector(sendHeight, sendWidth, srcWidth
    , MPI_FLOAT, &result);
8  MPI_Type_commit(&result);
9  MPI_Send(srcBuf + (srcOffsetRow * srcWidth +
    srcOffsetColumn), 1, result, toRank, msgTag,
    MPI_COMM_WORLD);
```

Listing 1: sendStridedBuffer()

2.3 Receive Strided buffer

In our program of the MPI-based Sobel filter, the recvStridedBuffer function addresses the counterpart of data transmission which is the reception and integration of data sent from all the other MPI ranks. The problem to solve is the collection and assimilation of incoming expectedHeight, expectedWidth into the local computational domain.

The code incrementally updates the messagesNum and dataMoved counters, which are metrics for tracking the communication efficiency of the distributed Sobel filter algorithm and it will be analized ahead. Utilizing MPI data types, exemplified by the MPI_Type_vector function, is crucial in creating a specialized data structure for the strided buffer. This tailored structure facilitates the seamless reception of the tile's data, taking into account its dimensions and the specified offset. The MPI_Recv function is then employed to capture the incoming strided buffer from the designated rank.

By encapsulating this functionality, the code forms a crucial component of the larger problem—constructing a robust framework for parallelized Sobel filtering. The successful execution of the recvStridedBuffer logic ensures that each MPI rank obtains the necessary data, fostering the collaborative and distributed nature of the Sobel filter algorithm across the computational domain.

2.4 Sobel Filter Implementation

The sobelAllTiles function serves as the backbone of our Sobel filter implementation, using the parallelized processing of tiles across MPI ranks. The primary problem addressed by this code is the need for a systematic approach to apply the Sobel filter algorithm to each tile within the distributed computational domain. The nested loops iterate through the 2D array of tiles, and for each tile associated

```
11 messagesNum++;
12 dataMoved += expectedHeight * expectedWidth *
    sizeof(dstBuf);

14 int msgTag = 0;
15 int recvSize[2];
16 MPI_Status stat;

18 MPI_Datatype result;
19 MPI_Type_vector(expectedHeight, expectedWidth,
    dstWidth, MPI_FLOAT, &result);
20 MPI_Type_commit(&result);
21 MPI_Recv(dstBuf + (dstOffsetRow * dstWidth +
    dstOffsetColumn), 1, result, fromRank, msgTag,
    MPI_COMM_WORLD, &stat);
```

Listing 2: recvtridedBuffer()

with the current rank (myrank), the Sobel filter is applied using the do_sobel_filtering function. This section of code is pivotal as it ensures the Sobel filter is consistently applied across all relevant tiles, facilitating the parallel nature of the algorithm within the distributed-memory framework. It establishes the connection between individual tiles and the Sobel filtering process, laying the groundwork for the collaborative computation that characterizes the larger Sobel filter implementation.

Within the sobelAllTiles framework, the do_sobel_filtering function embodies the core Sobel filtering algorithm applied to each tile. The problem here is the computation of Sobel filtering for a given tile, involving intricate calculations based on the tile's input and output buffers, as well as its width and height. The code adeptly utilizes the Sobel filter stencil, as defined by the Gx and Gy arrays, to compute the Sobel filter result for each pixel within the tile. The relationship of this code is that it encapsulates the essential Sobel filtering logic, ensuring each tile undergoes the filtering process as part of the larger parallelized Sobel filter implementation.

In order to take care of the Ghost cells, additional arguments are also used while calling the do_sobel_filtering, along with inputBuffer, outputBuffer, width and height, we will also pass ghost_xmax ghost_xmin ghost_ymax and ghost_ymin. This will take care of ghost cells by making sure that the tile that it is going to process on has sufficient data for the computating it entirely and leaving no "seams".

```
23 for (int row=0;row<tileArray.size(); row++)
24   {
25     for (int col=0; col<tileArray[row].size();
      col++)
26       {
27         Tile2D *t = &(tileArray[row][col]);

29         if (t->tileRank == myrank)
30         {
31             do_sobel_filtering(t->inputBuffer.data
      (), t->outputBuffer.data(), t->width, t->
      height, t->ghost_xmax, t->ghost_xmin, t->
      ghost_ymax, t->ghost_ymin);

33         }
34       }
35   }
```

Listing 3: sobelAllTiles()

3 EVALUATION

In our study, we begin by outlining the computational environment and methodology before delving into three comparative analyses, examining runtime performance, number of messages moved along with data movement across three decompositions which are Row, Column, and Tile. Our comprehensive approach involves the presentation of individual graphs for each decomposition strategy, illustrating scatter, processing, and gather times for runtime analysis. Simultaneously, a detailed table is provided for the data movement study, offering insights the number of messages moved and the volume of data transferred for each decomposition strategy by both sendStridedBuffer and recvStridedBuffer. This helps us in understanding the relation between the speedup achived with various concurency and the amount of data transfered.

3.1 Computational platform and Software Environment

All the programs were run using CPU nodes on Nersc. Perlmutter consists of 1536 GPU accelerated nodes with 1 AMD Milan processor and 4 NVIDIA A100 GPUs, and 3072 CPU-only nodes with 2 AMD Milan processors. Here we accessed 4 CPU nodes on Perlmutter. The experiments were performed on a computing platform powered by an NVIDIA GPU. The Sobel filter implementations were compiled and executed using the CPU environment. In order to check the filtered image, we ran a python code with version Python 3.9.6. This socket was accessed using a Mac OS with a apple Intel silicon processor and 16 GB RAM. The Mac terminal was used to access the shell. The Cmake version used was 3.20.4. The make version used is GNU Make 4.2.1.

3.2 Methodology

In our study, we focus on assessing the runtime performance of various decomposition strategies, examining specific stages such as data scattering, sobel filtering processing, and data gathering. Utilizing the chrono timer library to measure elapsed time in each phase, we conduct the experiments across different concurrency levels of [4, 9, 16 , 25, 36, 49, 64, 81]. These concurrency levels represent the number of ranks, with grid division into either row, column or tile decomposition. Following the collection of timings for each implementation, we generate a comprehensive graph to analyze the observed differences and potential reasons behind them. Additionally, we explore the number of messages sent and the total data moved between ranks as supplementary performance indicators, providing insights into the impact of data movement on the runtime for each decomposition, particularly in the scatter, sobel, and gather phases.

3.3 Runtime performance study

Here we explored varying levels of concurrency, meticulously recording elapsed times and constructing Speedup graphs to illustrate the findings. Notably, the three distinct phases—scatter, gather and process exhibited divergent trends. The Speedup, charted on the y-axis, showcased concurrency levels on the x-axis. we can ovserve A common trend in the increased Speedup for the process phase, indicating the advantageous distribution of work among a greater number of processors. Intriguingly, both scatter and gather Speedups reached a plateau after a specific concurrency threshold.

Turning our attention to the Row implementation (Figure 1), the Speedup for Sobel time demonstrated exponential growth with escalating concurrency. This phenomenon results from the increased number of processes, leading to a more extensive division of tasks among processors and subsequently boosting speed. However, the runtime plateaued around a concurrency level of 49, suggesting a limitation imposed by the finite number of processors. In the gather phase, there was a marked drop initially, followed by a stabilization, aligning with the Sobel filtering runtime.

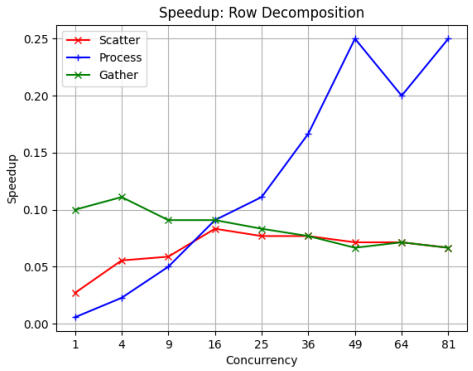

Figure 1: Comparison of Speedup of the scatter, process and gather phase in Row decomposition

For Column Decomposition a similar approach to the Row Decomposition was followed for Column decomposition with the only difference being on how the global buffer was divided. (Now in cols or width as compared to rows or height as before). In figure 2, the line chart for scatter time and sobel time behaviour is quite similar to Figure 1 or the row decomposition. The Speedup is almost linear for Scatter phase of the program. The reason for this seems similar to Row Decomposition.

Within the of Tile Decomposition, where the buffer was subdivided into smaller tiles, distinct improvements were observed. Figure 3 shows the trends of Row and Column Decomposition but with significantly faster runtimes. The noteworthy increase in Speedup for Tile Decomposition underscored its efficiency relative to other strategies.

The Tile decomposition shows a very good speedup with increase in concurency levels as compared to row and column decomposition strategy, this is because the way the image is divided, there comes a point where it cannot properly divide the image in equal parts and the last part of the process has more data to process and as we are using the MPI_Barrier, this does not let the other process start unless all the process in parallel are completed, this makes all the threads wait till the last one finishes its job, which usually has more work compared to other threads as the data is not divided perfectly across all the concurency. This slows the program after a certain amount of concurrency, which is very easily achived in row and column decomposition. It will also occur in Tile decomposition as well but it will take more amount of concurencies to achive at that point.

We can also observe that the row decomposition has a better runtime for scatter phase of the process, this is because of the row major format in which the data is stored and read, hence there is less memory copy incvolved.

3.4 Data Movement Performance Study

From Table 1 it is clearly visible that number of messages transfered is same for each of the strategy, but it can also be observed that the total data moved in tile decomposition is the least, the data moved in column decomposition is in the middle and the data moved from the row is the most. This makes sense as the data in the tile decomposition is in the smallest buffer and this also makes the process much faster as it is divided into smaller units. This is probably why the run time on average for tile decomposition is also the fastest. Although the number of messages sent for each of the decomposition is the same, the amount of data sent per message differs and the greater the data the longer it takes to scatter or gather that particular message.

The Row Decomposition uses more data compared to Column

Concurency	ROW		COLUMN		TILE	
	Message	Data Transferred (Bytes)	Message	Data Transferred (Bytes)	Message	Data Transferred (Bytes)
1	0	0	0	0	0	0
4	6	439521600	6	439386064	6	439327344
9	16	521451840	16	521145712	16	520900528
16	30	550696384	30	550210320	30	549553656
25	48	564920384	48	564125104	48	562927232
36	70	573340992	70	572194032	70	570290008
49	96	579030592	96	577545872	96	574789376
64	126	583582272	126	581580336	126	577789800
81	160	587451200	160	584956112	160	579916368

Table 1: Number of messages sent and the total data moved between ranks for each of the Row, Column and Tile decomposition Strategy

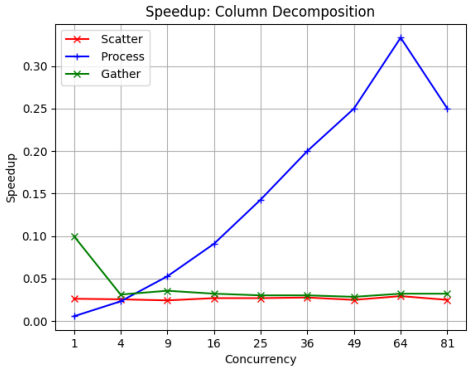

Figure 2: Comparison of Speedup of the scatter, process and gather phase in Column decomposition

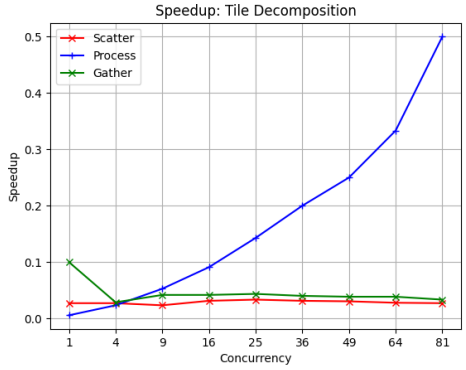

Figure 3: Comparison of Speedup of the scatter, process and gather phase in Tile decomposition

decomposition as the width of the image is much greater than its height. This means that the data is better evenly distributed in the column decompistion compared to the row decomposition strategy.

We find the Data Movement for send by the formula of :-

$$\text{dataMoved} = \text{dataMoved} + \text{sendHeight} \times \text{sendWidth} \times \text{sizeof}(\text{srcBuf}) \tag{1}$$

and for recive :-

$$\text{dataMoved} = \text{dataMoved} + \text{expectedHeight} \times \text{expectedWidth} \times \text{sizeof}(\text{dstBuf}) \tag{2}$$

3.5 Findings and Discussion

The performance analysis reveals distinctive patterns in data movement, directly influencing the runtime of Sobel filter processes across different decomposition strategies. Specifically, the tile decomposition strategy demonstrates the most efficient data transfer, moving the least amount of data.. This correlation between data movement and runtime aligns with the intuitive understanding that increased data implies longer processing times, given that each pixel invokes the Sobel filter method. Consequently, individual processes experience slightly prolonged durations, contributing to an overall impact on runtime. The Speedup by Tile Decomposition Strategy has a linear speedup so this is considered the best among row, column and tile decomposition strategy.

The amount of data transfered is comparatively more than the normal sobel filter implementation this is because we are also sending the halo cells to process. This leads to more data movement across nodes but this also increases the time to compute because now there is more data to be computed as well.