A study on how Concurrency and Decomposition Methods Affect the Performance of Stencil Operations in Distributed Programming Assignment #6, CSC 746, Fall 2024

Ruxue Jin*
SFSU

ABSTRACT

This study explores how the degree of concurrency and decomposition methods affect the performance of Sobel filter stencil operations in distributed programming. The paper employs row-slab, column-slab, and tile-slab methods, with various levels of concurrency. The performance is evaluated by measuring the runtime of three phases (scatter, process, and gather), as well as calculating the number of messages sent and the amount of data transmitted. The result shows that the tile slab method scales efficiently to 81 ranks, outperforming the row and column slabs, which peak at 64 ranks due to better load balancing and cache utilization. Scatter and gather speedup remains consistent across all methods due to similar data transfer volumes.

1 INTRODUCTION

The Sobel filter is an edge detection algorithm that uses stencil operations via convolutional multiplication across image pixels. This paper implements distributed programming to compute the Sobel operation across four nodes, aiming to study how concurrency and decomposition methods impact the performance of stencil operations in distributed programming.

This paper implements three decomposition methods for distributed Sobel filter operations: row-slab (dividing the image horizontally), column-slab (dividing the image vertically), and tile-slab (dividing the image into tiles). These methods are executed with varying levels of concurrency, using MPI for data distribution through send and receive operations. The runtime of three phases—scatter, process, and gather—is measured for each method, along with the number of messages sent and the total amount of data transferred at each concurrency level.

This paper observes that scatter and gather processes show minimal variation in speedup across all methods due to the consistent total data transfer, with only slight improvements attributed to increased ranks. While the row and column slab methods achieve peak performance at a concurrency of 64 before declining at 81, the tile slab method continues to scale efficiently, benefiting from better load balancing and cache utilization at higher concurrency levels.

2 IMPLEMENTATION

The code harness enables distributed Sobel edge detection on tiled image data using MPI(Message Passing Interface). It includes functions like sendStridedBuffer and recvStridedBuffer for efficient data communication with proper strides. The sobelAllTiles function processes tiles assigned to the current rank, applying the Sobel filter to compute edge gradients and store results in the tile's outputBuffer. This design ensures scalable, parallel processing of image data across multiple ranks in a distributed environment.

*email:rjin@sfsu.edu

2.1 sendStridedBuffer() Implementation

The function `sendStridedBuffer` is called by both function `scatterAllTiles` and function `gatherAllTiles`. It is responsible for sending data from one rank to another. Given a source buffer $srcBuf[srcWidth \times srcHeight]$ and a desired subregion of size $sendWidth \times sendHeight$ at offset $(srcOffsetColumn, srcOffsetRow)$, this function allocates a temporary contiguous buffer of size $sendWidth \times sendHeight$, copy the subregion row by row, send the contiguuod data to the target rank using MPI_Send.

The codes are shown in Listing 1.

```
sendStridedBuffer(srcBuf, srcWidth, srcHeight,
    srcOffsetColumn, srcOffsetRow, sendWidth,
    sendHeight, fromRank, toRank){

    tempBuf = allocate array of size sendWidth *
    sendHeight

  // Copy the strided data to a temporary buffer
  for (int i = 0; i < sendHeight; i++) {
      // Calculate source and destination offsets
      int srcOffset = (srcOffsetRow + i) *
   srcWidth + srcOffsetColumn;
      int destOffset = i * sendWidth;

      // Copy one row at a time
      memcpy(&tempBuf[destOffset],
          &srcBuf[srcOffset],
          sendWidth * sizeof(float));}

  // Send the contiguous data
  MPI_Send(tempBuf, sendWidth * sendHeight,
   MPI_FLOAT,toRank, msgTag, MPI_COMM_WORLD);

  delete[] tempBuf;}}
```

Listing 1: sendStridedBuffer:send data from fromRank to toRank.

2.2 recvStridedBuffer() Implementation

The function `recvStridedBuffer` is called by both function `scatterAllTiles` and function `gatherAllTiles`. It is responsible for receiving data from fromRank to toRank. It allocates a temporary buffer of size $sendWidth \times sendHeight$ for the incoming data, receives the data with `MPI_Recv`, and copies it into the specified subregion of the destination buffer (`dstBuf`) with proper offsets and striding. The codes are shown in Listing 2.

2.3 Sobel Implementation

Sobel filter operation operates using convolution, where two 3x3 kernels (horizontal and vertical) are applied to an image to approximate the derivatives in the x and y directions. The horizontal kernel

```
recvStridedBuffer(dstBuf, dstWidth, dstHeight,
    dstOffsetColumn, dstOffsetRow, expectedWidth,
    expectedHeight, fromRank, toRank ) {

  tempBuf = allocate array of size sendWidth *
    sendHeight

  // Receive the data
  MPI_Recv(tempBuf, expectedWidth *
    expectedHeight, MPI_FLOAT, fromRank, msgTag,
    MPI_COMM_WORLD, communicator&stat);

  // Copy data from temporary buffer to
    destination with proper striding
  for (int i = 0; i < expectedHeight; i++) {
      // Calculate offsets
      int dstOffset = (dstOffsetRow + i) *
    dstWidth + dstOffsetColumn;
      int srcOffset = i * expectedWidth;

      // Copy one row at a time
      memcpy(&dstBuf[dstOffset],
          &tempBuf[srcOffset],
          expectedWidth * sizeof(float));    }

  delete[] tempBuf;}
```

Listing 2: recvStridedBuffer: receive data from fromRank to toRank

detects vertical edges, while the vertical kernel detects horizontal edges. The gradient magnitude in the Sobel filter is computed by combining the horizontal (G_x) and vertical (G_y) gradients using the formula:

$$G = \sqrt{G_x^2 + G_y^2} \qquad ([2])$$

Implementation involves iterating over each pixel, applying the kernels to the pixel's neighborhood, and calculating the gradient.

The algorithm `do_sobel_filtering` is called by function `sobelAllTiles`. It iterates over all input image pixels and invoke the sobel_filtered_pixel() function at each (i,j) location of input image `in`.

The function defines Sobel kernels `Gx` and `Gy`. For each pixel, `sobel_filtered_pixel` computes the horizontal and vertical gradients, and the resulting gradient magnitude is stored in `out`. The codes are shown in Listing 3.

The `sobelAllTiles` function iterates over a 2D array of `Tile2D` objects, each representing a tile of image data. For each tile, it checks if the tile's rank matches the current rank (`myrank`). If the rank matches, the function processes the tile by calling the `do_sobel_filtering` function.

3 Result

This section describes the testing environment for the three programs and outlines the performance metrics used in their evaluation: runtime speedup in each phase of distributed programming, the number of messages sent, and the amount of data transferred across ranks.

3.1 Computational platform and Software Environment

The experiments were conducted on four compute nodes of the NERSC Perlmutter system [1]. Each node's CPU configuration consists of dual AMD EPYC 7763 (Milan) processors, delivering 39.2 GFLOPS per core. The system features a three-level cache hierarchy: 32 KB L1 and 512 KB L2 caches per core, and a 256 MB shared L3 cache. The software environment included the Ubuntu

```
do_sobel_filtering(in, out, ncols, nrows):
  // Initialize Gx and Gy
  Gx = [1.0, 0.0, -1.0, 2.0, 0.0, -2.0, 1.0, 0.0,
    -1.0]
  Gy = [1.0, 2.0, 1.0, 0.0, 0.0, 0.0, -1.0, -2.0,
    -1.0]

  // Apply parallelization to nested loops for all
     pixels
  #pragma omp parallel for collapse(2)
  for i from 0 to nrows - 1:
    for j from 0 to ncols - 1:
      // Compute the Sobel filter result for the
    pixel at (i, j)
      out[i * ncols + j] = sobel_filtered_pixel(in
    , i, j, ncols, nrows, Gx, Gy)
```

Listing 3: Sobel: process the data in each rank

22.04 operating system with Linux kernel 5.15. Code compilation was performed using the GNU C++ compiler version 7.5.0.

3.2 Methodology

The program applies the Sobel operation using three different decomposition methods: row, column, and tile, based on occurrences of specific values: [4, 9, 16, 25, 36, 64, 81].

For each occurrence, the elapsed runtime is measured for three phases—scatter, Sobel, and gather—using the `chrono` function in the code. The speedup is based on a concurrency level of 4, where the speedup for a concurrency of 4 is always 1. For other concurrency levels, the speedup is calculated as the runtime with concurrency 4 divided by the runtime at the given concurrency level.

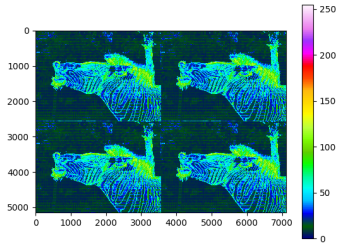
Additionally, the implementation of the code is analyzed to calculate the total number of messages sent and the total volume of data transferred between ranks. The total number of messages sent is determined by counting the number of calls to the `MPI_Send` function in the `scatterAllTiles` and `gatherAllTiles` functions, which is $2 \times (N-1)$, where N is the number of concurrent ranks. In this implementation, there is no overlap of boundary messages, so the amount of data transferred is the same for all ranks. The total volume of data transferred is calculated by

$$\text{Total data transferred} = \frac{7112 \times 5146 \times (N-1)}{N} \times 2 \times 4\,\text{bytes}$$

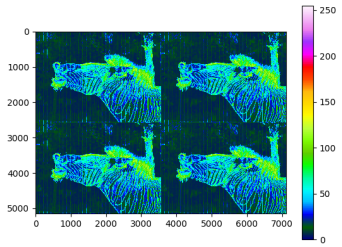
where N is the number of concurrent ranks.

3.3 Results of Sobel Operation with Different Decomposition Methods

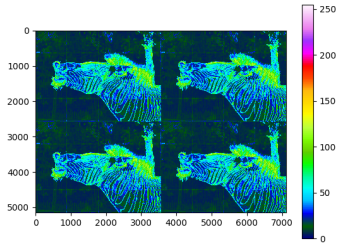
In Figures 1a,1b,1c, the results demonstrate the effects of the three decomposition methods—row-slab, column-slab, and tile-slab—on the Sobel operation. The program does not handle the boundaries of each tile effectively, resulting in visible artifacts: horizontal lines in the row-slab decomposition, vertical lines in the column-slab decomposition, and rectangular grid patterns in the tile-slab decomposition. This issue arises because the Sobel algorithm relies on neighboring pixel values to compute gradients. When the image is divided into slabs or tiles, the boundary pixels lack access to neighboring pixels in adjacent regions, leading to incomplete or incorrect gradient calculations. As a result, the edges near the boundaries are either omitted or appear discontinuous, creating the observed patterns in the output images.



(a) Row-slab



(b) Col-slab



(c) Tile-slab

Figure 1: Results of Sobel Operation using Row, Column, and Tile Decomposition Methods with 64 Concurrency.

3.4 Runtime performance study

3.4.1 Row Slab Method

In Figures 2a,the Sobel time (blue line) shows a clear upward trend, increasing almost linearly from 4 to 49 ranks, where it reaches a speedup of approximately 11.5x. The Sobel time then peaks at 12x with 64 ranks but declines slightly to 11x with 81 ranks, likely due to communication overhead or inefficiencies in data distribution at higher concurrency.

In contrast, the scatter time (red line) and gather time (green line) show minimal speedup, remaining consistently near 1x to 1.3x. The scatter time is slightly higher than the gather time, particularly beyond 36 ranks, suggesting that the scatter phase incurs greater communication overhead or synchronization delays. Overall, the Sobel phase demonstrates significant benefit from increased concurrency, while the scatter and gather phases appear to be bottlenecked by communication overhead, limiting their speedup.

3.4.2 Column Slab Method

In Figures 2b, The Sobel time (blue line) demonstrates a steady linear increase in speedup from 4 to 64 ranks, reaching a peak of approximately 12x at 64 ranks. However, the speedup decreases slightly to about 10x at 81 ranks, indicating diminishing returns or communication overhead at higher concurrency levels. In contrast, the scatter time (red line) and gather time (green line) show minimal improvement, with speedups remaining consistently close to 1x to 1.3x. Interestingly, the scatter time experiences a slight decline beyond 36 ranks, whereas the gather time stabilizes at around 1.3x. This behavior suggests that the Sobel phase benefits more significantly from increased concurrency, while the scatter and gather phases are likely constrained by communication overhead or synchronization delays.

3.4.3 Tile Slab Method

Figures 2c show that in the tile-based decomposition, the speedup of the Sobel phase increases steadily as concurrency levels rise, achieving its highest value at 81 ranks. This trend indicates that the Sobel phase benefits significantly from increased parallelism, with no signs of diminishing returns in this setup. The consistent growth in speedup suggests that the workload is well-suited for distributed processing and that the system's resources are effectively utilized without significant communication overhead or bottlenecks, even at higher concurrency levels.

The scatter and gather speedup decrease slightly but remain relatively stable.

3.4.4 Runtime findings

The three methods show similar results for scatter and gather speedup. The row and column slab methods exhibit a "sweet spot" at a concurrency level of 64 in the Sobel process, achieving the highest speedup before dropping at a concurrency level of 81. In contrast, the tile slab method continues to increase in speedup even at a concurrency level of 81.

3.5 Data Movement Performance Study

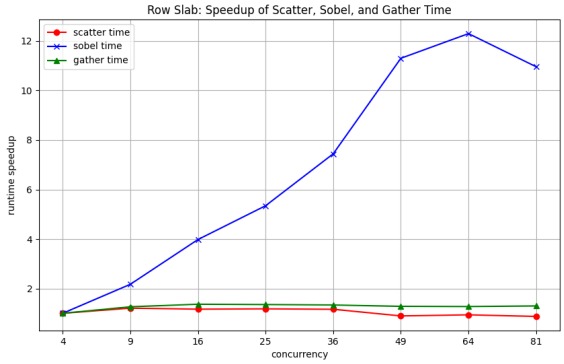
As concurrency increases, the number of messages sent rises linearly, while the size of each message decreases proportionally. This balance ensures that the total amount of data transmitted remains approximately the same.

The scatter and gather times increase slightly or remain nearly constant with increasing concurrency because these operations are primarily influenced by the total amount of data being transmitted, which does not change significantly with concurrency. Slight increases in scatter and gather times may occur due to minor overheads introduced by handling a greater number of smaller chunks, such as additional synchronization or network latency. Despite this, the impact is minimal, keeping the scatter and gather times nearly constant as concurrency grows.

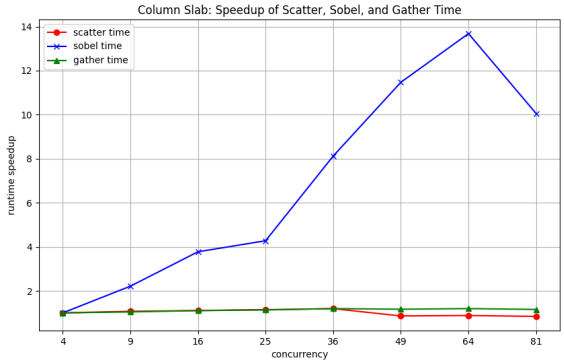
However, performance begins to degrade after reaching the "sweet spot" due to increasing overhead and resource contention. At higher concurrency levels, the overhead of managing more messages—such as creating, transmitting, receiving, and processing them—becomes significant. Additionally, shared resources like CPU, memory, and network bandwidth face heightened competition, leading to inefficiencies like context switching, queuing delays, and memory bottlenecks. Furthermore, the high volume of small messages can amplify latency through network congestion or queuing effects. These factors, combined with potential imbalances in workload distribution, cause the system to perform worse as concurrency exceeds its optimal range. The sweet spot represents the balance where resource utilization and overhead are optimized, and any further increase in concurrency results in diminishing returns.

3.6 overall findings and discussion

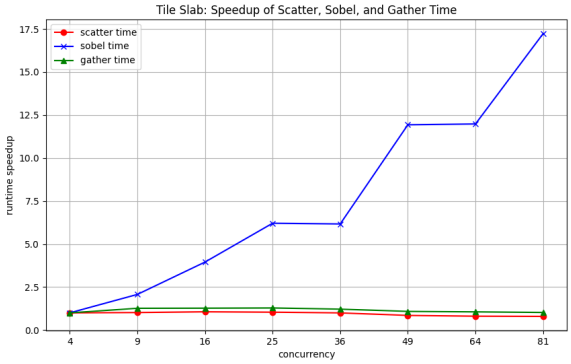
In all three methods, the speedup for the scatter and gather processes does not change significantly, as these processes are primarily influenced by the total amount of data transferred between ranks. Since the total amount of data remains similar across concurrency levels, the scatter and gather processes show only slight variations



(a) Row slab: runtime speedup of scatter, sobel, and gather state



(b) Column slab: runtime speedup of scatter, sobel, and gather state



(c) Tile slab: runtime speedup of scatter, sobel, and gather state

Figure 2: Runtime speedup comparison for scatter, sobel, and gather states using Row, Column, and Tile Decomposition Methods.

in performance. The small increase in speedup can be attributed to the increased number of ranks, which may optimize communication patterns slightly.

The row and column slab methods demonstrate a gradual increase in speedup during the Sobel process, reaching their best performance at a concurrency level of 64. However, their performance declines at a concurrency level of 81. This drop is likely caused by communication overhead and inefficiencies in data distribution at higher concurrency levels. As more ranks are added, the benefits of parallelism are offset by the increased complexity and time required for communication and synchronization among ranks.

Concurrency	Row-Slab		Row-Slab		Row-Slab	
	#msgs	tdm	#msgs	tdm	#msgs	tdm
4	6	220M	6	220M	6	220M
9	16	260M	16	260M	16	260M
16	30	275M	30	275M	30	275M
25	48	281M	48	281M	48	281M
36	70	285M	70	285M	70	285M
49	96	287M	96	287M	96	287M
64	126	288M	126	288M	126	288M
81	160	289M	160	289M	160	289M

Table 1: The number of messages send

Unlike the row and column slab methods, the tile slab method does not experience a decline in speedup at a concurrency level of 81. This superior performance can be attributed to the nature of the tile-based decomposition. By dividing the workload into smaller, more granular tiles, the tile slab method ensures better load balancing across ranks. This approach minimizes idle time and reduces the impact of communication overhead, enabling the system to maintain efficiency even at high concurrency levels.

Acknowledgments

The author would like to acknowledge the assistance of ChatGPT, a language model developed by OpenAI, for helping to articulate the interpretation of our data and refine the techinical writing.

References

[1] N. E. R. S. C. C. (NERSC). Perlmutter system overview. https://www.nersc.gov/users/systems/perlmutter/, 2024. Accessed: 2024-10-07.

[2] Wikipedia contributors. Euclidean distance, 2024. Accessed: 2024-11-19.