Exploring MPI-based Sobel code in multiple configurations and analyzing multiple performance metrics: runtime of the different processing phases, and the amount of data moved between MPI ranks Assignment #6, CSC 746, Fall 2022

Akshay Mhatre*
SFSU

ABSTRACT

The problem we will be looking at is the performance of different phases of sobel code. We will be analyzing runtime of different process of sobel code such as send buffer, recieve buffer and sobel filtering as well as data moved between MPI ranks. We will run the test at different level of concurrency on 5 different nodes. Run time decreases with concurrency for process phase whereas it varies (increases, decreases) for scattering and gathering phase

1 INTRODUCTION

In this assignment we will be analyzing performance of different phase of sobel by running it on different concurrency level on different KNL node. There are three phases that will be analyzed i.e., send buffer where data is send from one rank to another, receive buffer where the data is received from another rank and sobel filtering which applies the filter to given image. Run time for all phases will be calculated and compared as well as amount of data moved between MPI ranks.

2 IMPLEMENTATION

In this section we will go over different phases of implementations, how they work. There are 3 different phases of implementation, send buffer, receive buffer and sobel filtering.

2.1 Overview of Code

The code is an MPI-based framework that performs several important processing steps: loading input, performing domain decomposition, building a 2D array of Tile objects reflecting the domain decomposition, executing scatter processing. There are three main components we will looking is sending data using send buffer, receiving data using receive buffer and sobel filtering.

2.2 Data sent with Send Buffer

In this implementation we send data using send buffer and is responsible for sending data from one rank to another rank. We do this by creating a vector which stores data of height by width of send data from the source buffer.The vector datatype doesn't support an initial offset as part of the MPI API therefore we use a manual offset on the array address and send the vector using MPI_Send() with data offset and the destination rank. Ref. Listing 1.

2.3 Data Received with Receive Buffer

In this implementation we receive data using recieve buffer and is responsible for receivig the data from another rank. We do this by creating a vector which stores data of height by width of destination data which will be received. The vector datatype doesn't support an initial offset as part of the MPI API therefore we use a manual offset on the array address and pass the vector using MPI_Recv() with data offset and the from rank. The data received is initially stored in vector and then copied into destination buffer. Ref. Listing: 2.

```
void sendStridedBuffer(float *srcBuf,
      int srcWidth, int srcHeight,
      int srcOffsetColumn, int srcOffsetRow,
      int sendWidth, int sendHeight,
      int fromRank, int toRank )
{
   messagesNum++;
   dataMoved += sendHeight * sendWidth * sizeof(
    srcBuf);
   int msgTag = 0;
   MPI_Datatype result;
   MPI_Type_vector(sendHeight, sendWidth, srcWidth
    , MPI_FLOAT, &result);
   MPI_Type_commit(&result);
   MPI_Send(srcBuf + (srcOffsetRow * srcWidth +
    srcOffsetColumn), 1, result, toRank, msgTag,
    MPI_COMM_WORLD);
}
```

Listing 1: Send Buffer

2.4 Sobel Filtering

In this implementation we perform sobel filtering by computing the convolution of Gx and Gy at Si. The magnitude of the vector (Gx, Gy) is the sobel operator value at that point which is calculated by square root of Gx^2 and Gy^2. Two loops are used to iterate over 3x3 matrix. We also consider xmin, xmax, ymin and ymax values to consider halo cells. Ref. Listing: 3.

3 RESULTS

In this section we will go over the results of different implementations phases. We will look at the run time at different concurrency across five different nodes.

3.1 Computational platform and Software Environment

The tests were ran on Perlmutter's interactive node:
Processor:
For CPU:
CPU Nodes: Dual AMD EPYC 7763 (Milan) CPUs,
Core Count: 64 cores per CPU (128 total cores)
Clock rate: 1.4 GHz
Level 1 Cache: 64 KB
Level 2 Cache: 512 KB
DRAM - 512 GB
Complier:
CXX compiler: GNU 11.2.0

*email:amhatre@sfsu.edu

```
void recvStridedBuffer(float *dstBuf,
      int dstWidth, int dstHeight,
      int dstOffsetColumn, int dstOffsetRow,
      int expectedWidth, int expectedHeight,
      int fromRank, int toRank )
{
   messagesNum++;
   dataMoved += expectedHeight * expectedWidth *
    sizeof(dstBuf);
   int msgTag = 0;
   int recvSize[2];
   MPI_Status stat;
   MPI_Datatype result;
   MPI_Type_vector(expectedHeight, expectedWidth,
    dstWidth, MPI_FLOAT, &result);
   MPI_Type_commit(&result);
   MPI_Recv(dstBuf + (dstOffsetRow * dstWidth +
    dstOffsetColumn), 1, result, fromRank, msgTag,
     MPI_COMM_WORLD, &stat);
}
```

Listing 2: Receive Buffer

```
float sobel_filtered_pixel(float *s, int i, int j
    , int ncols, int nrows, float *gx, float *gy)
{
float
sobel_filtered_pixel(float *s, int i, int j ,
    float *gx, float *gy, int xmax, int xmin, int
    ymax, int ymin)
{
   float t=0.0;
   float Gx = 0.0;
   float Gy = 0.0;
   if (i > ymin && i < ymax - 1 && j > xmin && j <
     xmax - 1) { //Prevent segment fault
      for (int x = 0; x < 3; x++) {
         for (int y = 0; y < 3; y++) {
            Gx += gx[x * 3 + y] * s[(i - ymin + x
    - 1) * (xmax - xmin) + (j - xmin + y - 1)];
            Gy += gy[x * 3 + y] * s[(i - ymin + x
    - 1) * (xmax - xmin) + (j - xmin + y - 1)];
         }
      }
   }
   t = sqrt(Gx*Gx + Gy*Gy);
   return t;
}
```

Listing 3: Sobel Filtering

Reference:
Perlmutter: https://docs.nersc.gov/systems/perlmutter/architecture/

3.2 Methodology

The implementations for three different phase were tested at different level of concurrency across 5 nodes with different grid decomposition strategies. Performance metrics: Elapsed time for different stages of the code which was calculated using chronotimer() function. Total number of messages is calculated by counting number of times send and receive buffer function is called and the total amount of data between ranks is calculated by multiplying the number of data by height and width (number of rows and columns) of sent and expected data. Tests ran over the following sizes:
Concurrency levels: 4, 9, 16, 25, 36, 49, 64, 81.
Grid composition strategies: Row-slab, Column-slab and tiled (1, 2, 3).

3.3 Phase: Send data

In this phase we send data using MPI_Send(). We create a vector of size height by width of send data and copy data from source buffer of size width by height of source buffer, calculate the offset manually on the array address and pass it to MPI_Send() along with the rank, MPI then does striding. Number of messages sent and the total amount of data moved between ranks and time taken was calculated at different level of concurrency.

3.4 Phase: Receive Data

In this phase we receive data using MPI_Recv(). We create a vector of size height by width of expected data which is used to store data been received which is then copied data to destination buffer of size width by height of destination buffer, calculate the offset manually on the array address and pass it to MPI_Recv() along with the rank, MPI then does striding. Number of messages received and the total amount of data moved between ranks and time taken was calculated at different level of concurrency.

3.5 Phase: Sobel Filtering

In this experiment we ran code from Sobel Filter in which we go over a 3 by 3 block of the given image and apply filter. Time was calculated at different level of concurrency. Time taken was less for higher level of concurrency. We use xmin, xman, ymin, ymax values to encounter halo cells which are the seams on the images and can be seen after the filter is applied. We have these values to eliminate seems or remove halo cells. These cells are sent during sobel filtering phase but are not returned to gathering phase thus eliminating the seams. Sec. 3.2

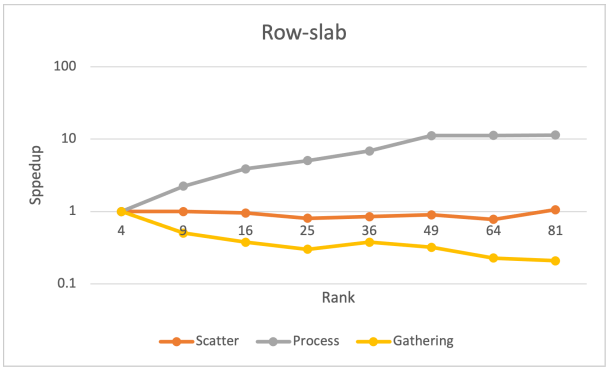

Figure 1: Speedup chart for Row-slab

3.6 Runtime performance study

In all the charts speedup for process is increasing with concurrency. For higher level of concurrency, time taken for process to complete is less. Scattering's speedup chart is also similar for all three charts but gathering is different for row-slab speedup chart. Gathering in row-slab take more time for higher concurrency and the run time increases more compared to column-slab and tiled.
Performance for process phase is consistent in all the decomposes, run time for this phase decreases in all decompose. Scatter has somewhat consistent performance as its time increases, decreases or stays somewhat same across all decompose. Whereas gather phase run time increased more in row-slab decompose compared to column-slab and tiled. Results for this experiment can be found in Fig 1, Fig 2 and Fig 3.

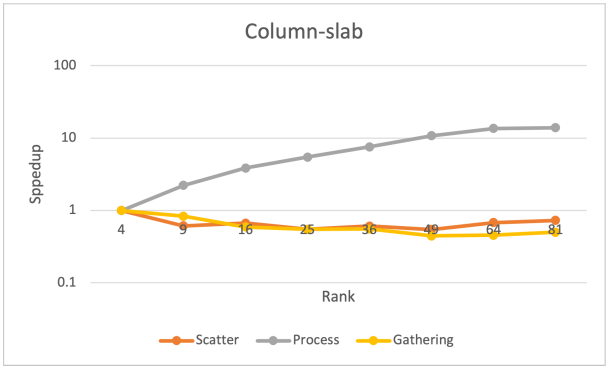

Figure 2: Speedup chart for Column-slab

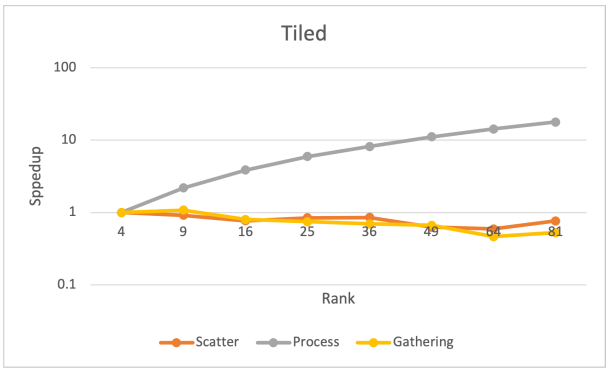

Figure 3: Speedup chart for Tiled

Row-slab		
Rank	Message	Data moved (Mb)
4	6	439.52
9	16	521.45
16	30	550.7
25	48	564.92
36	70	573.34
49	96	579.03
64	126	583.58
81	160	587.45

Table 1: Number of messages and data moved in Row-slab

Column-slab		
Rank	Message	Data moved (Mb)
4	6	439.39
9	16	521.15
16	30	550.21
25	48	564.12
36	70	572.19
49	96	577.55
64	126	581.58
81	160	584.96

Table 2: Number of messages and data moved in Column-slab

Tiled		
Rank	Message	Data moved (Mb)
4	6	439.33
9	16	520.9
16	30	549.55
25	48	562.93
36	70	570.29
49	96	574.79
64	126	577.79
81	160	579.92

Table 3: Number of messages and data moved in Tiled

There can be rows with no data in them but it will still be counted which will produce errors.

3.7 Data movement performance study

Number of messages sent are same from all decompose at a given concurrency and they increase as the concurrency increases. Data moved is also similar for all three decompose and increases with concurrency. As the data moved increases with concurrency, run time in scatter and process also increases as they have to move more amount of data between ranks and as the number of ranks also increases, it takes more time to move data. As process is running in parallel, run time decreases in that decompose as the concurrency increases, data moved between ranks is done in parallel.

3.8 Overall findings and discussion

Amount of data moved is similar across all three grid decomposition strategy and increases as concurrency increases. Running more parts in parallel might improve the run time since data moved between multiple ranks can be done at once. I believe calculating data moved between ranks might contribute to inaccuracy as currently we do that by multiplying the data's width by height by number of data.