Assignment #6, CSC 746, Fall 2021

Zhuozhuo Liu*
San Francisco State University

ABSTRACT

We explored Message Passing Interface (MPI) by creating a distributed-memory parallel version of Sobel filtering. We tested how the concurrency levels and grid decomposition strategies affected the performance on scatter, Sobel, and gather phases. We ran our program on 5 KNL nodes on Cori. We observed that as concurrency level got larger, the time of scattering got larger while the time of Sobel operation and gathering decreased. We also found that tiled decomposition transferred least data among all 3 ways. We summarized future improvement in the end.

1 INTRODUCTION

In this assignment, we applied Sobel filter on an image by using stencil to compute gradients of the image. We used distributed memory system to implement paralleled computing by using MPI where rank 0 sent data to other ranks (i.e., scatter), then other ranks sent back the result to rank 0 (i.e., gather).

As concurrency level got larger, the performance of Sobel operation got accelerated. However we observed a difference between scatter phase and gather phase: although the amount of MPI functions called were the same, the scatter took way longer time than gather, which may because that $MPI_Send()$ was called serially while $MPI_Recv()$ was called parallel. We also found that tiled decomposition is the most efficient one since it transferred least data among all 3 ways, although the messages transferred were the same.

2 IMPLEMENTATION

To goal is to implement a Sobel filter which takes an input image (2D array), apply convolution on each pixel and output the filtered image. We distributed the job on multiple processes and used MPI to send and receive data between processes/ranks.

2.1 Overall code harness

In the main function, we initialized MPI and created P ranks. We also decomposed the input image into P rectangle tiles. There are 3 ways to decompose the input: by row, by column, by tile. Among all P ranks, rank 0 not only took care of its own tile 0 but also sent out the rest tiles to the rest ranks, we called this a scatter phase. In this way each rank was matched with one tile and was able to apply Sobel filtering on its own tile. Then during a gather phase, each rank except rank 0 sent back the output to rank 0 to gather all results so that we have a global output. We also fixed the ghost zone issue by modifying scatterAllTiles() and Sobel operation.

2.2 sendStridedBuffer() implementation

This function (see Listing 1) is called by rank 0 in scatter phase, or called by rank 1 to rank P-1 in gather phase. It first converted the tile from a 2d matrix to an 1d array and then call $MPI_Send()$. In this way the data being sent is an 1d array so that we can only call $MPI_Send()$ once per tile instead of calling it multiple times to send every row, which may work more efficiently.

*email:zliu15@mail.sfsu.edu

Inputs are srcBuf(a pointer to the input matrix, can be a global matrix or a local tile), srcWidth(the width of the global matrix), srcHeight(the height of the global matrix), srcOffsetColumn(y coordinate of the top left pixel of the tile), srcOffsetRow(x coordinate of the top left pixel of the tile), sendWidth(the width of the tile), sendHeight(the height of the tile), fromRank(the rank which sends the tile) and toRank(the rank which receive the tile).

The function returns void.

```
{
   off_t s_offset=0, d_offset=0;
   vector <float> d;
   d.reserve(sendWidth * sendHeight);
   float *source = srcBuf + srcOffsetRow *
    srcWidth + srcOffsetColumn;
   for (int j=0;j<sendHeight;j++, s_offset+=
    srcWidth, d_offset+=sendWidth)
   {
      memcpy((void *)(d.data()+d_offset), (void *)
    (source+s_offset), sizeof(float)*sendWidth);
   }

   MPI_Send(d.data(), sendWidth * sendHeight,
    MPI_FLOAT, toRank, msgTag, MPI_COMM_WORLD);
}
```

Listing 1: the implementation of sendStridedBuffer(), which performs sending of data using $MPI_Send()$, going from fromRank to toRank. We converted 2d tile to 1d array in order to send all data once.

2.3 recvStridedBuffer() implementation

This function (see Listing 2) is called by rank 1 to rank P-1 in scatter phase, or called by rank 0 in the gather phase. It first call $MPI_Recv()$ which received an flat 1d-array tile and then parse the tile back from 1d array to 2d matrix and finally stored the matrix in the output buffer.

Inputs are dstBuf(a pointer to the output matrix, can be a global matrix or a local tile), dstWidth(the width of the output matrix), dstHeight(the height of the output matrix), dstOffsetColumn(y coordinate of the top left pixel of the tile), dstOffsetRow(x coordinate of the top left pixel of the tile), expectedWidth(the width of the tile), expectedHeight(the height of the tile), fromRank(the rank which sends the tile) and toRank(the rank which receive the tile).

The function returns void since output buffer is passed as an parameter.

2.4 Sobel implementation

sobelAllTiles() (see Listing 3) is a function that call $do_sobel_filtering()$ only when the ith process finds the ith tile. That's why there are 2 nested loop over tileArray. It is for each process to match with its tile and then apply Sobel filter on it. To do so, we iterated over tile in tileArray and assign the tile to its corresponding rank.

The inputs of sobelAllTiles() are myrank(the rank of the process), a tileArray is the global matrix after decomposed into many tiles. $t->inputBuffer.data()$ is a padded tile. $t->outputBuffer.data()$ is smaller than the inputBuffer.data since it's

```
{
    vector <float> source;
    source.reserve(expectedWidth * expectedHeight);
    MPI_Recv(source.data(), expectedWidth *
     expectedHeight, MPI_FLOAT, fromRank, msgTag,
     MPI_COMM_WORLD, &stat);
    off_t s_offset=0, d_offset=0;
    float *dest = dstBuf + dstOffsetRow * dstWidth
     + dstOffsetColumn;
    for (int j=0;j<expectedHeight;j++, s_offset+=
     expectedWidth, d_offset+=dstWidth)
    {
       memcpy((void *)(dest+d_offset), (void *)(
     source.data()+s_offset), sizeof(float)*
     expectedWidth);
    }
}
```

Listing 2: the implementation of recvStridedBuffer(), which performs sending data using $MPI_Recv()$, going from fromRank to toRank. We parsed the 1d data received to 2d matrix.

the same size of the tile without padding. The function returns void since output buffer is passed as an parameter. In the end the output buffer of each tile is filled with Sobel value.

```
sobelAllTiles(int myrank, vector < vector < Tile2D
     > > & tileArray) {
    for (int row=0;row<tileArray.size(); row++)
    {
       for (int col=0; col<tileArray[row].size();
     col++)
       {
          Tile2D *t = &(tileArray[row][col]);
          if (t->tileRank == myrank)
          {
             // t->inputBuffer.data() is a padded
     tile. t->outputBuffer.data() is smaller than
     the inputBuffer.data since it's the same size
     of the tile without padding
             do_sobel_filtering(t->inputBuffer.data
     (), t->outputBuffer.data(), t->width+2, t->
     height+2);
          }
       }
    }
}
```

Listing 3: the implementation of sobelAllTiles(), which make each process match with its tile and then apply Sobel filter on the tile.

In $do_sobel_filtering()$ (see Listing 4), we looped over each pixel, calculated the Sobel value for each, and wrote the value in out buffer. The input buffer is 2 cols and 2 rows larger than the original tile size. Since it was composed of the core area and a padding layer around the core, we only computed the Sobel value of the core area.

The inputs of $do_sobel_filtering()$ are in(input 2d tile with padding), out (output buffer without padding), cols(how many columns in the input), rows(how many rows in the output). The function returns void since output buffer is passed as an parameter.

2.5 Halo implementation

We implemented halo version to fix the ghost zone (see Listing 5). In the scatter phase, rank 0 pads the global input with one layer of 0 around 4 directions (up, down, left, right). When rank 0 sends the

```
do_sobel_filtering(float *in, float *out, int cols
     , int rows)
{
    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};
    for (int i=1; i<rows-1; i++) {
       for (int j=1; j<cols-1; j++) {
          out[(i-1)*(cols-2)+(j-1)] =
     sobel_filtered_pixel(in, i, j, cols, rows, Gx,
     Gy);
       }
    }
}
```

Listing 4: the implementation of $do_sobel_filtering()$, which looped over each pixel, calculated the Sobel value for each, and wrote the value in out buffer. We only compute the Sobel value of the core area.

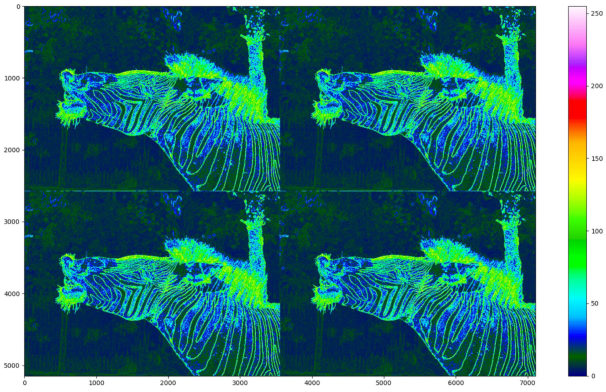

Figure 1: The outputs of program is the same with the correct output (before implementing a fix on the ghost zone).

input, it sends out (width+2)*(height+2) values. Accordingly, rank 1 to rank P-1 receives (width+2)*(height+2) values from rank 0.

One optimization that we want to make in future is to pad the global input outside of the scatter function instead of creating it each time calling scatter. We will analyze the performance of the current implementation in the next section.

3 Results

3.1 Computational platform and Software Environment

All tests were running on Cori KNL node. The processor was Intel Xeon Phi Processor 7250. The clock rate was 1.4 GHz. The size of L1 cache was 64 KB. The size of L2 cache was 1MB. The memory size was 96 GB for DDR4 and 16 GB for MCDRAM. Cmake version was 3.14.4. GCC version was 8.3.0. Optimization level was O3 by default. OpenMPI version was 4.0.2. [1]

3.2 Methodology

We first validated the program by comparing the output image with the correct output image (see Figure 1). The dimension of the input image was 7112 * 5146 (col * row).

There are 2 metrics that we measured: 1) elapsed runtime of different stages of the code in ms, and 2) number of messages and amount of data moved between ranks.

We recorded the run time of scatter phase, gather phase and Sobel operation phase. We used chrono timer to record the run time.

```
if (myrank == 0 && t->tileRank != 0) {
    off_t s_offset=0, pad_offset=0;
    // We created padded the global input here
    // We should optimize it by creating a global vector instead of creating it each time calling scatter
    vector <float> padS((global_width+2) * (global_height+2), 0);
    float *padPtr = padS.data() + (global_width+2) + 1;
    for (int i=0; i<global_height; i++) {
        memcpy((void *)(padPtr + pad_offset), (void *)(s+s_offset), sizeof(float)*global_width);
        pad_offset += global_width+2;
        s_offset += global_width;
    }

    sendStridedBuffer(padS.data(),
         global_width+2, global_height+2,
         t->xloc, t->yloc,
         t->width+2, t->height+2,
         myrank, t->tileRank);
}

if (myrank != 0 && t->tileRank == myrank)
{
    int fromRank=0;
    t->inputBuffer.resize((t->width+2)*(t->height+2));
    t->outputBuffer.resize(t->width*t->height);
    recvStridedBuffer(t->inputBuffer.data(), t->width+2, t->height+2,
          0, 0,  // offset into the tile buffer: we want the whole thing
          t->width+2, t->height+2, // how much data coming from this tile
          fromRank, myrank);
}
```

Listing 5: the implementation of recvStridedBuffer(), which performs sending data using $MPI_Recv()$, going from fromRank to toRank. We parsed the 1d data received to 2d matrix.

We computed the number of messages and the amount of data moved between ranks by inferring from the implementation code.

we tested 3 grid decomposition strategies: row-slab, column-slab, and tiled. For each strategy, we ran the program on 5 KNL nodes at 8 different concurrency levels P = 4, 9, 16, 25, 36, 49, 64, 81. In total, we ran 3 * 8 tests.

3.3 Runtime performance study

Figure 2, 3, 4 show the speedup chart for row-slab, row-slab, and tiled decomposition configuration respectively. The y-axis is the run time after apply natural log since the scatter time is very large. In each figure, there are 3 data sets for scatter phase, Sobel operation phase, and gather phase.

Among all decomposition configurations, the pattern seems very similar: the run time of scatter phase increased as concurrency got larger; the run time of Sobel operation decreased as concurrency got larger. The run time of gather phase first decreased and then remained flat as concurrency got larger. This suggested that Sobel operation was accelerated with larger concurrency. We hypothesized that row-slab should performed slightly better than the other two because of the advantage of locality but this didn't happen in our test. It might because that the run time was dominated by other factors such as $MPI_Send()$ and $MPI_Recv()$.

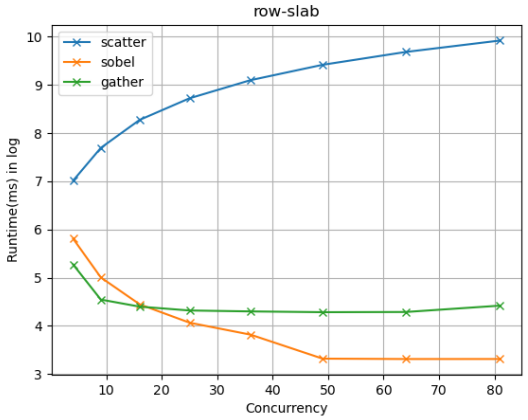

Figure 2: The speedup chart for row-slab decomposition configuration. The run time of scatter phase increased as concurrency got larger; the run time of Sobel operation and gather phase decreased and remained flat as concurrency got larger. This figure was produced by the program `plot.py` file.

The run time of scatter phase increased as concurrency level got larger since 1) we created a padded matrix during scatter phase and this is called P-1 times. So when P got larger, number of tiles got larger and this was called more times. We should move the logic outside of scatter to only initialize the padded matrix onece. 2) We also observed the same performance even on the basic version (without halo implemented). Therefore, we think it might because of the nature of MPI messaging. Rank 0 had to send out data to other ranks in a serial fashion, which took longer time than than rank 0 receiving data from other ranks asynchronously. In the gather phase, other ranks can write to the global buffer at the same time since they are writing to the different locations so it took less time.

3.4 Data movement performance study

Table 1 shows the comparison on data movement among different decomposition configurations (in 10^5). Let's say P is the level of concurrency, height is 5146, width is 7112. We got the numbers based on the following calculation.

For row-slab, the number of messages is $(P-1)*2$. For data moved in scatter phase, we sent the padding so it is $4bytes*((5146/P+2)*(7112+2))$. For data moved in gather phase, since we didn't send padding, it is $4bytes*((5146/P)*7112)$.

For col-slab,the number of messages is $(P-1)*2$. For data moved in scatter phase, we sent the padding so it is $4bytes*((5146+2)*(7112/P+2))$. For data moved in gather phase, since we didn't send padding, it is $4bytes*(5146*(7112/P))$.

For tiled-slab,the number of messages is $(P-1)*2$. For data moved in scatter phase, we sent the padding so it is $4bytes*((5146/sqrt(P)+2)*(7112/sqrt(P)+2))$. For data moved in gather phase, since we didn't send padding, it is $4bytes*((5146/sqrt(P))*(7112/sqrt(P)))$.

For all 3 cases, since the data in rank 0 doesn't moved, the total data movement is $(P-1)*(dataMoveInScatter+dataMoveInGather)$

We observed that the message sent remains the same among decomposition configurations. It's because that the number of tiles are the same for a given P. Therefore, no matter in what decomposition fashion, rank 0 will need to send P-1 messages for scattering and other ranks will need to send P-1 messages back for gathering.

We observed a difference on the amount of data sent among decomposition configurations. Tiled version transferred the least

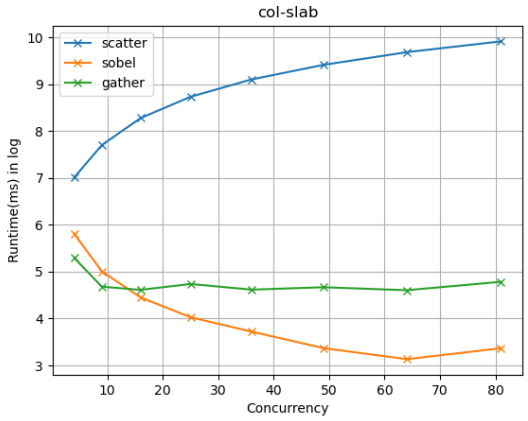

Figure 3: The speedup chart for col-slab decomposition configuration. The run time of scatter phase increased as concurrency got larger; the run time of Sobel operation and gather phase decreased and remained flat as concurrency got larger. This figure was produced by the program `plot.py` file.

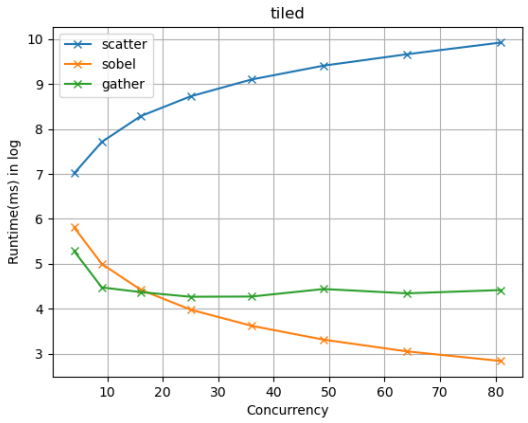

Figure 4: The speedup chart for tiled decomposition configuration. The run time of scatter phase increased as concurrency got larger; the run time of Sobel operation and gather phase decreased and remained flat as concurrency got larger. This figure was produced by the program `plot.py` file.

P	Row-msg	Row-data	Col-msg	Col-data	Tile-msg	Tile-data
4	6	2197.92	6	2197.56	6	2197.37
9	16	2607.47	16	2606.35	16	2605.17
16	30	2753.80	30	2751.59	30	2748.56
25	48	2824.80	48	2821.18	48	2815.46
36	70	2866.86	70	2861.51	70	2852.26
49	96	2895.84	96	2888.44	96	2874.85
64	126	2918.38	126	2908.63	126	2889.85
81	160	2937.66	160	2925.23	160	2900.45

Table 1: Number of messages sent and the total data moved between all ranks (in 10^5). The message sent remains the same among decomposition configurations. Tiled version transferred the least data and the row version transferred the most data.

data and the row version transferred the most data. The pattern can be found in all concurrency levels Ps. since we implemented padding. It suggests that tiled version can be the most efficient way in terms of transferring less data.

Another thing we observed is that the amount of data sent increased as P got larger. Since the image was decomposed to more tiles as P got larger, the padding data got larger as well.

3.5 Overall findings and discussions

Tiled version transferred the least data and the row version transferred the most data. It indicated that tiled version is the most efficient among all 3 ways. But we are not sure if it's because that the tileHeight/tileWeight proportion is close to globalHeight/globalWeight proportion. This depends on the input of height = 5146 and width = 7112 and might work differently among heights and weights. We will need to do more math and test on image sizes to understand where the best decomposition strategy come from.

One optimization that we want to make in future is to pad the global input outside of the scatter function to save a lot unnecessary time in scatter time. Currently the scatter time is very large since we padded it each time calling scatter which is not efficient.

For the measurement, there might be some inaccuracies since the elapsed time included rank 0 copied tile 0 into its own buffer, which didn't belong to MPI's time. Also since we implemented padding, there can be some time wasting on memcopy the padding. The data movement is not accurate since there can be some slight difference on the size of each tile if the height and width is not divisible by P.

REFERENCES

[1] https://docs.nersc.gov/systems/cori/. Cori system specification.