MPI Programming
Assignment #6, CSC 746, Fall 2021

Arulselvan Madhavan*
SFSU

ABSTRACT

The aim of this study is to understand the feasibility of distributed memory programming in executing computationally expensive problems. Sobel edge algorithm was chosen to represent a computationally expensive problem and run under three different data decomposition strategies - row decomposition, column decomposition and tiled decomposition. The effectiveness of the distributed memory paradigm was measured using runtime, number of messages sent between processes and the data moved between processes. In the experiment, tiled decomposition strategy came out as the winner by aiming to distribute the load equally among both dimensions of the tile. The experiments also showed that while the kernel computation shows good speedup, the overall speedup of the program is largely dependent on the number of messages and data sent between processes. The lower they were the better the speedup was.

1 INTRODUCTION

The main objective of this work is to understand how distributed memory programming could be leveraged to solve a computationally intensive problem like Sobel filtering. Distributed memory programming involves using multiple nodes connected together to solve problems. Message Passing Interface(MPI) [1] provides a set of specifications to implement distributed memory programming by leveraging multiple nodes. A well known implementation of the MPI specifications called OpenMPI [2] was used to implement sobel filtering by leveraging multiple nodes. The results show that the time taken to send data over the network to other nodes could dominate the computation time if the computation kernel is not very expensive to compute. Distributed memory programming shines well when the kernel is expensive to compute and the amount of data moved between nodes is maintained to a minimum.

2 IMPLEMENTATION

Sobel edge detection algorithm was implemented using the parallel programming framework provided by MPI [1]. In the distributed sobel computation, the input image is split into chunks and assigned to a processor, that could be either in the same node or on a different node. One chunk per processor was maintained in all the experiments.

The decomposition strategy used on the data was one of three types.

- Row decomposition
- Column decomposition
- Tiled decomposition

2.1 Row decomposition

Split the data horizontally as shown in Figure:1

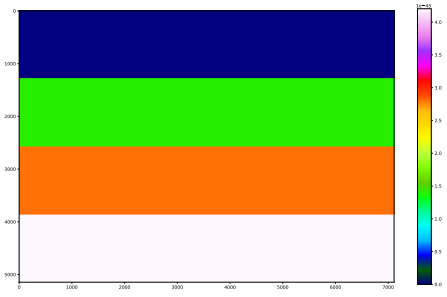

Figure 1: Row decomposition

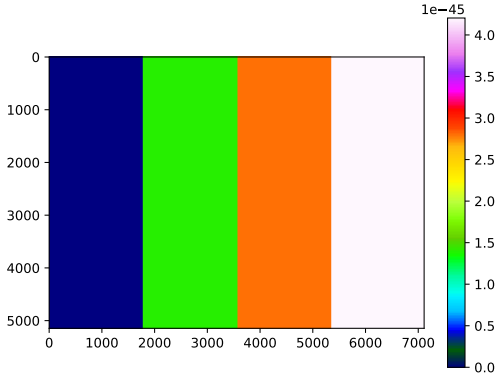

Figure 2: Column decomposition

2.2 Column decomposition

Split the data vertically as shown in Figure:2

2.3 Tiled decomposition

Split the data into equal sized tiles as show in Figure:3. Tile width and tile height were chosen as the sqrt(P), where P is the total number of processes made available to the program.

2.4 Phases of a distributed computation

The usual phases of a distributed computation are

- Scatter Phase
- Ghost cell update phase
- Kernel computation phase
- Gather Phase

*email:amadhavan@sfsu.edu

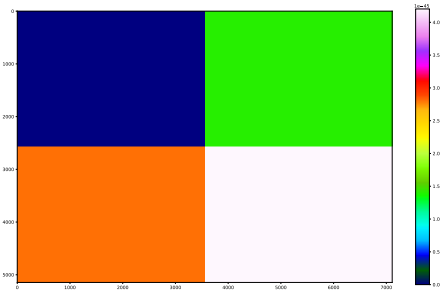

Figure 3: Tiled decomposition

2.5 Scatter Phase

Scatter Phase involves sending data from one node to all the other nodes. In the sobel implementation, one node reads the input image and sends a chunk(selected by the decomposition method described above) to each process involved in the sobel computation. The scatter phase logic is shown in Listing:1. The most important line is MPI_Isend which sends the data to a specific rank(process) and returns immediately to send the data to the next rank.

```
void sendStridedBuffer(float *srcBuf, int srcWidth
    , int srcHeight,
                       int srcOffsetColumn, int
    srcOffsetRow, int sendWidth,
                       int sendHeight, int
    fromRank, int toRank) {
  int msgTag = 0;
  MPI_Request request;
  int srcPos = srcOffsetRow * srcWidth +
    srcOffsetColumn;
  int count = sendWidth * sendHeight;
  float *startPos = srcBuf + srcPos;
  float *buffer = (float *)malloc(count * sizeof(
    float));
  for (int j = 0; j < sendHeight; j++) {
    int bPos = j * sendWidth;
    int rPos = j * srcWidth;
    memcpy((void *)(buffer + bPos), (void *)(
    startPos + rPos),
           sizeof(float) * sendWidth);
  }
  MPI_Isend(buffer, count, MPI_FLOAT, toRank,
    msgTag, MPI_COMM_WORLD, &request);
  MPI_Request_free(&request);
}
```

Listing 1: Sobel Scatter Phase

2.6 Ghost cell update

Ghost cells are cells that represent the neighbor cells. Each process can have a neighbor to its left, right, top or bottom in a 2D grid. Cells from the neighbors are needed to compute the sobel kernel. So, each grid maintained by a rank(process) is padded by one on each side. This padding is referred to as the ghost cell since it's initially empty and has to be updated in the ghost cell phase. In the ghost cell phase, each process figures out its neightbors and sends the data the neighbor might need, and also receives the data it might need from the neighbors. A sample ghost cell update to the left and right neighbors is shown in Listing:2. A synchronization is needed after exchanging cells to the neighbors on left and right, before exchanging data with neighbors on top and bottom. At the end of ghost cell update, all ranks should have all the data they need to do the sobel filtering.

2.7 Sobel Implementation

After the ghost cell update, all ranks have the data they need to operate on. Each tile executes the sobel computation on its grid. The ghost cells from the neighbors are used at the edges of the grid. When a neighbor is not present, the grid is operating at the edge of the image. So, a zero value is used. The sobel kernel invocation is show in Listing:3. The sobel filtering logic is shown in Listing:4

2.8 Gather Phase

In the gather phase, all the ranks send their data to the rank that is responsible for writing the output to the filesystem. No parallel IO was needed or implemented in this work. So, all the output was written by rank 0. The implementation of the gather phase is the reverse of scatter phase and it is shown in Listing:5

3 Evaluation

The aforementioned implementations were run on a 2D image of size: 7112x5146. The experiments were run on the Cori cluster. The CPUs used were Intel's Knights Landing Processor [3].

3.1 Computational platform and Software Environment

The specifications were taken from `https://docs-dev.nersc.gov/cgpu/hardware/` CPU specs:

- CPU: Intel's Knights Landing Processor
- CPU clock cycle: 1.4GHZ
- Memory Bandwidth: 102 GiB per sec
- Theoretical Peak FLOPS per node: 3 PFlops
- Number of cores: 68
- CPU per core: 4
- Maximum number of threads: 272
- L1 caches: 64 KB (32 KiB instruction cache, 32 KB data)
- L2 cache: 1MB
- 16 GB of MCDRAM (spread over 8 channels)
- Compiler: gcc 8.3.0 (O3 optimization)
- OS: SUSE Linux Enterprise Server 15
- OpenMPI version: 4.1

3.2 Methodology

The experiments were run using the three decomposition modes described in Section:2. The runtime of the each phase was computed by introducing the chrono timer before the start and after the end of each phase. The total number of messages used for communication in each phase was calculated by counting the number of sends and receives that happen in each phase. For example: The scatter phase that uses row decomposition has to send data to P-1 ranks, where P is the total number of processes involved in the computation. Similarly, the total number of bytes involved in the transfer was computed by calculating the size of the data that is submitted to the MPI for send/receive. For example, in the scatter phase, data sent to other processes = total size of image - size of data in rank 0. Size of data sent in gather phase is equal to the size of data sent in scatter phase.

In the ghost cell update phase, the data calculation is dependent on the available neighbors and the size of ghost row/column. The logic is show below in Listing:6. For each process, we identify the neighbors. For each neighbor, there is a send involved. The size is equal to the width/height of the row/column exchanged.

Size of the image is 7112*5146. Each pixel is represented using a 8 byte float. So, total data = 7112*5146*8.

3.3 Runtime Performance study

Runtime behavior of the three decomposition methods are shown in Figures:4, 5, 6. All three decomposition methods, show an increase in runtime across scatter, gather, ghost cell update phases, and show a decrease in runtime across the sobel computation phase as the number of processes involved in the computation increases. This shows that the MPI programming is suitable for reducing the kernel computation time. This also shows that for best results, the number of messages and the data sent over the network should be reduced. In this experiment, we loaded the data in one rank and sent it to all the other ranks. MPI can support parallel IO. So, the increase in runtime in scatter,gather phases can be reduced significantly, if not eliminated using parallel IO. The main takeaway is the feasibility of distributed memory programming in sobel computation. As the results indicate, the runtime of sobel computation decreases as the parallelization increases.

Speedup of sobel phase in row decomposition saw a major jump from P=36 to P-49. But after that, the speedup wasn't significant. This shows that dividing the the height of the image 5146 by more than 50 chunks of length 7112 isn't very useful.

Speedup of sobel phase in column decomposition saw consistent increase until P=64. But saw a sharp decline at P=81. This shows that dividing the image's width of 7112 into 81 pieces isn't very conducive to speedup.

Speedup of sobel phase in tiled decomposition saw consistent increase across all parallelization levels. The speedup was almost linear. Since the speedup was consistent and linear, this seems to be the best strategy for dividing a 2D grid into smaller pieces and computing results.

The reason the tiled decomposition showed better speedup characteristics is the equal distribution of load in both dimensions. The scatter phase of row decomposition performed better than the other two because there is less memory copy involved as we can exploit the C language's row major allocation of values in memory.

3.4 Data Movement Performance study

The distribution of messages and the data transferred via messages is shown in table:1. The number of messages sent increases as P increases. This is expected since the number of processes that needs to communicate with another increases. Similar reasoning could account for the increase in data movement. In the scatter and gather phases, the data sent and received increases as the number of data kept in rank 0 decreases. The number of messages sent in tiled decomposition is high because the tiles are more likely to have more than one neighbor in this strategy. So, the number of messages sent in ghost cell update phase is higher. One caveat to note is that the image size is around 36MB. Image dimensions are 7112*5146. Since each pixel is represented by a 8 byte float, the total data involved is 7112*5146*8 = 292MB. Scatter and Gather phase has to nearly (292MB - size of tile at rank 0) messages between processes. This accounts for most of the data movement. The number of ghost messages and data sent in the ghost update phase also increases, there are more tiles and processes to communicate to.

The data sent in row decomposition is greater than data sent in column decomposition and data sent in tiled decomposition. This is because the width of the image 7112 is significantly larger than the height. In tiled decomposition strategy, the sizes of the tiles sent across processes is more equally distributed. So, the data transferred is more optimal in the tiled decomposition phase.

P	Row		Column		Tiled	
	msgs	data(MB)	msgs	data(MB)	msgs	data(MB)
4	12	439.58	12	439.43	14	439.38
9	32	521.51	32	521.19	40	520.93
16	60	550.75	60	550.25	78	549.58
25	96	564.98	96	564.17	128	562.95
36	140	573.40	140	572.24	190	570.31
49	192	579.09	192	577.59	264	574.80
64	252	583.64	252	581.62	350	577.80
81	320	587.51	320	585.00	448	579.92

Table 1: Data Movement study

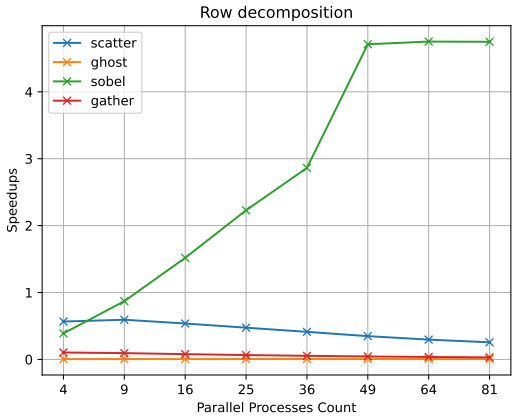

Figure 4: Speedup - Row Decomp

3.5 Overall Findings and Discussion

This experiment shows that distributed memory programming can be effectively leveraged to solve computationally intensive problems. However the effectiveness and the efficient use of the resources relies on carefully lowering the number of messages and the data sent between processes. One technique to lower the amount of data sent between processes is to use an effective data decomposition strategy. In this experiment, we studied three different decomposition strategies and found that the tiled decomposition strategy which tries to maintain an equal width and equal height 2D grid is more effective than other simpler techniques like row decomposition or column decomposition. The runtime of the sobel computation showed a linear speedup in the tiled decomposition strategy crowning it as the undisputed winner of the three.

There are several rooms for improvement in the MPI version of sobel computation. Moving parallel IO is one. If each process could read/write the image in parallel, the data sent in scatter/gather phases could go away completely. Then the runtime only is bound by the slowest process and not by the network bandwidth. Another improvement is to MPI datatypes like MPI_subarray to prevent the data copy that happens in the scatter, gather and ghost update phases. Another improvement is to avoid ghost update by reading the halo cells while reading the file data in parallel. Because sobel computation is not an iterative process, there is no sync required between processes if each can read just one layer of ghost cells from the file.

Above measurement methods doesn't provide any way of measuring network bandwidth related slowness. So, it's possible between runs, the runtime could differ based on how congested the traffic is between the nodes used for sobel computation. Similarly, the

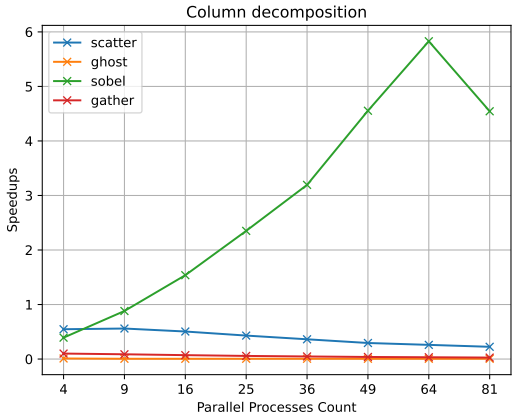

Figure 5: Speedup - Column Decomp

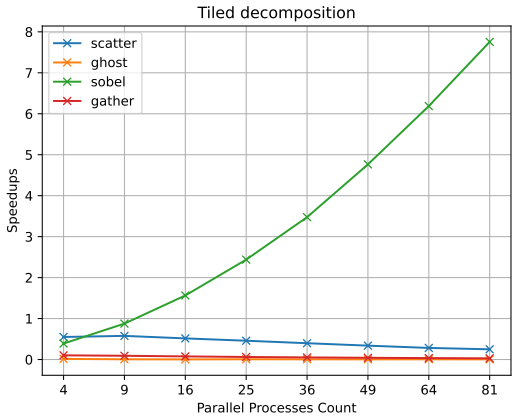

Figure 6: Speedup - Tiled Decomp

time spent in MPI_waitall is not measured. So, it's possible that the CPU is underutilized for a long time and the runtime calculated here doesn't distinguish the wait time.

REFERENCES

[1] Message Passing Interface - https://en.wikipedia.org/wiki/Message_Passing_Interface
[2] OpenMPI - https://www.open-mpi.org/
[3] https://docs.nersc.gov/systems/cori/knl_modes/

```
    if (t->nleft != MPI_PROC_NULL) {
      int icount = 0;
      int s_off = nhalo;
      for (int j = 0; j < tileH; j++, s_off += tileW) {
        for (int ll = 0; ll < nhalo; ll++) {
          xbuf_left_send[icount] = in[s_off + ll];
          icount++;
        }
      }
    }
    if (t->nrght != MPI_PROC_NULL) {
      int icount = 0;
      int s_off = nhalo + t->width - nhalo;
      for (int j = 0; j < tileH; j++, s_off += tileW) {
        for (int ll = 0; ll < nhalo; ll++) {
          xbuf_rght_send[icount] = in[s_off + ll];
          icount++;
        }
      }
    }
    MPI_Irecv(&xbuf_rght_recv, bufcount, MPI_FLOAT, t->nrght, 1001,
              MPI_COMM_WORLD, &request[0]);
    MPI_Isend(&xbuf_left_send, bufcount, MPI_FLOAT, t->nleft, 1001,
              MPI_COMM_WORLD, &request[1]);
    MPI_Irecv(&xbuf_left_recv, bufcount, MPI_FLOAT, t->nleft, 1002,
              MPI_COMM_WORLD, &request[2]);
    MPI_Isend(&xbuf_rght_send, bufcount, MPI_FLOAT, t->nrght, 1002,
              MPI_COMM_WORLD, &request[3]);
    MPI_Waitall(4, request, status);

    if (t->nrght != MPI_PROC_NULL) {
      int icount = 0;
      int d_off = nhalo + t->width;
      for (int j = 0; j < tileH; j++, d_off += tileW) {
        for (int ll = 0; ll < nhalo; ll++) {
          in[d_off + ll] = xbuf_rght_recv[icount];
          icount++;
        }
      }
    }
    if (t->nleft != MPI_PROC_NULL) {
      int icount = 0;
      int d_off = 0;
      for (int j = 0; j < tileH; j++, d_off += tileW) {
        for (int ll = 0; ll < nhalo; ll++) {
          in[d_off + ll] = xbuf_left_recv[icount];
          icount++;
        }
      }
    }
```

Listing 2: Sobel Ghost Phase

```
void 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) {
        float *in = t->inputBuffer.data();
        float *out = t->outputBuffer.data();
        int dims[2] = {t->width, t->height};
        int ghdims[2] = {t->width + 2 * nhalo, t->height + 2 * nhalo};

        for (int y = nhalo; y < dims[1] + nhalo; y++) {
          int off = y * ghdims[0];
          for (int x = nhalo; x < dims[0] + nhalo; x++) {
            int outIdx = off + x;
            out[outIdx] = sobel_filtered_pixel(in, x, y, ghdims, Gx, Gy);
          }
        }
      }
    }
  }
}
```

Listing 3: Sobel Kernel Phase

```
float sobel_filtered_pixel(float *in, int x, int y, int dims[], float *gx,
                           float *gy) {
  int gi = 0;
  float sum = 0.0;
  float gytemp = 0.0;
  float gxtemp = 0.0;
  for (int ky = -1; ky < 2; ky++) {
    for (int kx = -1; kx < 2; kx++) {
      int cxIdx = x + kx;
      int cyIdx = y + ky;
      int cyPos = cyIdx * dims[0];
      int cxy = cyPos + cxIdx;
      float sxy = in[cxy];
      gxtemp += gx[gi] * sxy;
      gytemp += gy[gi] * sxy;
      gi++;
    }
  }
  return sqrt((gxtemp * gxtemp) + (gytemp * gytemp));
}
```

Listing 4: Sobel Filter Phase

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

  int msgTag = 0;
  MPI_Status status[4];
  MPI_Request request[4];

  int ecount = expectedHeight * expectedWidth;
  float *buffer = (float *)malloc(ecount * sizeof(
    float));
  MPI_Recv(buffer, ecount, MPI_FLOAT, fromRank,
    msgTag, MPI_COMM_WORLD,
          &status[0]);
  int s_off = 0;
  int Dw = dstWidth + 2 * nhalo;
  int d_off = ((dstOffsetRow + nhalo) * Dw) + (
    dstOffsetColumn + nhalo);
  for (int j = 0; j < expectedHeight;
       j++, s_off += expectedWidth, d_off += Dw) {
    memcpy((void *)(dstBuf + d_off), (void *)(
    buffer + s_off),
          sizeof(float) * expectedWidth);
  }
}
```

Listing 5: Sobel Gather Phase

```
def calc_gmsg(rank, nx, ny, nX, nY):
    nleft = rank - 1 if (nx > 0) else
    MPI_PROC_NULL
    nrght = rank + 1 if (nx < nX - 1) else
    MPI_PROC_NULL
    ntop = rank - nX if (ny > 0) else
    MPI_PROC_NULL
    nbot = rank + nX if (ny < nY - 1)  else
    MPI_PROC_NULL
    gy = [1 for np in [nleft, nrght] if np !=
    MPI_PROC_NULL]
    gx = [1 for np in [ntop, nbot] if np !=
    MPI_PROC_NULL]
    return sum(gx) + sum(gy)

def calc_gdata(rank, nx, ny, nX, nY):
    nleft = rank - 1 if (nx > 0) else
    MPI_PROC_NULL
    nrght = rank + 1 if (nx < nX - 1) else
    MPI_PROC_NULL
    ntop = rank - nX if (ny > 0) else
    MPI_PROC_NULL
    nbot = rank + nX if (ny < nY - 1)  else
    MPI_PROC_NULL
    gy = [yd for np in [nleft, nrght] if np !=
    MPI_PROC_NULL]
    gx = [xd for np in [ntop, nbot] if np !=
    MPI_PROC_NULL]
    return sum(gx) + sum(gy)
```

Listing 6: Number of messages sent in ghost cell update phase