Practical 3: MPI

Caleb Bredekamp[†] and Michael Du Preez[‡]
EEE4120F Class of 2020
University of Cape Town
South Africa
[†]BRDCAL003 [‡]DPRMIC007

***Abstract*—This practical investigates the effect of partitioning data for processing in a distributed memory programming model with a Message Passing Interface and compares speedup with regard to a serial implementation.**

I. Introduction

The aim of this practical is to explore the use of Message Passing Interfaces in distributed-memory processing tasks. In this case, the task was splitting up an image for distributed median filtering. This was done by selecting and describing a method for partitioning the data and deciding on a method for processing edges and explaining it. The data was then split up into slave processes, processed, and compiled at the master. It was expected that the MPI implementation of the median filter would far outpace a serial version and to test this, the serial version was timed against the MPI version with four slave processes. To align with good practices, cache warming was taken into account and the programs were run at least 10 times with the average time being taken for a speedup calculation. The effects of having different numbers of slave processes are also investigated.

II. Methodology

A. Hardware

These experiments were performed on a 2.1GHz quad-core, 64-bit laptop with 8GB of RAM running OpenMPI version 2.1.1. The machine is running Linux Mint version 19.3.

B. Implementation

1) Partitioning and Sending Data: The approach chosen for partitioning data is shown in Code Block 1. It was decided that the data should be cut equally amongst the slave processes in order to split the amount of processing each would do equally. The partitioned 2D arrays will change size according to how many processes are being used, hence *numprocs* being used to split them. Code Block 1 shows how the height of the input picture is first split up. The addition of extra rows of pixels to some partitions is to ensure edge processing is only done at the edges of the actual picture and not in between partitions. Thus the process of finding the median is uniform for the entire image barring the edges of the actual input image.
Following this the dimensions to be used for each worker process is sent to each respective slave. After the dimension array is sent, the Master process waits for the respective slave to send an acknowledgement message for the dimensions. Code Block 2 then shows how the slave 2D arrays are filled and sent to each slave. Again, after each array is sent by the Master process, it waits for the slave to send an ACK indicating that the message was successful.

```
// Splitting up the processing as evenly as possible
for (int i = 0; i<numprocs-1; i++)
{
  if (mod_check > 0)
  {
    if(i==(numprocs-2) || i ==0)
    {
      y_portions[i] = int(Input.Height/(numprocs-1)) + 1 + (int)WINDOW_Y/2;
    }
    else
      y_portions[i] = int(Input.Height/(numprocs-1)) + 1 +2*((int)WINDOW_Y/2);
    mod_check--;
  }
  else
  {
    if (i==(numprocs-2) || i ==0)
    {
      y_portions[i] = int(Input.Height/(numprocs-1)) + (int)WINDOW_Y/2;
    }
    else
      y_portions[i] = int(Input.Height/(numprocs-1)) + 2*((int)WINDOW_Y/2);
  }
}

boolean ack = false; //! ack: Slaves set true when array is received
// Telling slaves what sizes to expect
int largest = 0;
for(int j = 1; j < numprocs; j++)
{
  int dimensions[DIMS] = {y_portions[j-1] , Input.Width*Input.Components,
                                                          Input.Components};
  if (y_portions[j-1] > largest){
      largest = y_portions[j-1];
  }
  MPI_Send(dimensions, DIMS, MPI_INT, j, TAG, MPI_COMM_WORLD);
  MPI_Recv(dimensions, DIMS, MPI_INT, j, TAG, MPI_COMM_WORLD, &stat); // ACK
  printf("Process %d received a height of %d and length of %d pixels\n", j,
                                  dimensions[0], dimensions[1]/Input.Components);
}
```

Code 1. Finding and Sending Dimensions

```
//! indexer: for aligning the Input and Output array with split arrays
  int indexer = 0;
  // Writing the portioned arrays and sending to respective slaves
  for (int proc = 0; proc < numprocs-1; proc++)
  {
    unsigned char slave [y_portions[proc]][Input.Width*Input.Components];
    for(int j = 0; j < y_portions[proc]; j++)
    {
      if(proc > 0 && j ==0)
        indexer = indexer + y_portions[proc-1] - 2*(WINDOW_Y/2);

      for (int i =0; i < Input.Width*Input.Components; i++)
        slave[j][i] = Input.Rows[j+indexer][i];
    }

    int size = sizeof(unsigned char)*y_portions[proc]*Input.Width
                                                          *Input.Components;

    MPI_Send(slave, size, MPI_CHAR, proc+1, TAG, MPI_COMM_WORLD); //send array
    MPI_Recv(&ack, 1, MPI_CHAR, proc+1, TAG, MPI_COMM_WORLD, &stat);// recv ACK

    if (ack)
      printf("Data partition %d successful\n", proc+1);
  }
```

Code 2. Creating and Sending Partitioned Arrays

2) Edge Processing Methods: Looking at other image processing algorithms, the following options were discovered as options for dealing with edge cases.

- Ignore edges: Simply not assigning the edges and being left with a black border since the values of the array are pre-assigned to 0.
- Ignore edges and crop: This is similar to the above but would reslut in the image getting smaller and smaller each time it is filtered.
- Shrink window: As the edges are reached, the window would shrink. This should result in non-uniform blurring since the image would get sharper towards the edges.
- Duplicate edge values: At the edges, the values of the input are duplicated to fill the window. Depending on the position of the window, this could result in a similar case to the above where the image gets sharper towards the edges.
- Wrap around: At the edges, the window is filled with values elsewhere in the image, in this case, from the other side. This should result in more uniform blurriness but could result in abnormalities if the image is, for example, light on one side and dark on the other.

Our approach was most similar to the **Wrap Around** option above with one slight difference. Instead of wrapping around the entire image to fill the window, the window will be filled with values near to it. For example, if the window's centre pixel is at the leftmost edge of the 2D partitioned array, the leftmost values of the window are filled with values just outside the right edge of that window. This keeps the filtering somewhat uniform as stated above but with the added benefit that if the image is vastly different on one side than the other, the wrap around will not affect the median value since the chosen pixel value is close to the placement of the window. This is shown in Code Block 3.
It is shown how the partition of the array the slave receives is looped through, with each element forming the centre of their respective windows. The median of each window is calculated and the value at the centre is replaced with the median.
This process is complicated slightly by a colour JPEG. Since each pixel in the image is actually three components, the median is calculated separately for each colour value and replaced. This is why the window is a 2D array, one array for each colour.
The series of nested *if()* and *else if()* statements is the implementation of the **wrap around window** version of dealing with edges. This is explained above.

3) Reassembly: Once the Master process receives the processed array partition from the slave process, it is slotted back into an output array. An *Indexer* variable is used to correctly line up the partitions to ensure no line of blank pixels in between partitions. This effectively reverses the method above of having some extra lines of pixels in some partitions. This is shown in Code Block 4.

```
// Start of Distributed Median
  int window_y = WINDOW_Y;  //! window_y: y-dimension of median window
  int window_x = WINDOW_X;  //! window_x: x-dimension of median window
  int elements  = window_y*window_x; //! elements: total num of items in window
  int window [dimensions[2]][elements]; //! window[][]: 2D for RGB
  int counter  = 0; //! counter: used to find columns of the same colour
                    //! to find median
  int x, y; //! x, y: iterating variables through array items
  // Performing the filter
  for(y = 0; y < dimensions[0]; y++){
    for(x = 0; x < dimensions[1]; x=x+dimensions[2]){
      for(int j = -int(window_y/2); j<int(window_y/2)+1; j++){
        for(int i = -int(window_x/2)*dimensions[2];
            i<int(window_x/2)*dimensions[2]+1*dimensions[2]; i=i+dimensions[2]){
          for (int colour = 0; colour < dimensions[2]; colour++){
            // Checking if window is at edge (top or left)
            if (y+j > -1 && (x+i+colour)>-1 && (x+i+colour) < dimensions[1] &&
                                                              y+j < dimensions[0]){
              window[colour][counter] = input_arr[y+j][x+i+colour];
            }
            else if (y+j < 0){
              if ((x+i+colour)<0){
                window[colour][counter] = input_arr[y+j+window_y]
                                                          [x+i+colour+window_x];
              }
              else
                window[colour][counter] = input_arr[y+j+window_y][x+i+colour];
            }
            else if (y+j > -1 && (x+i+colour)<0){
              window[colour][counter] = input_arr[y+j][x+i+colour+window_x];
            }

            // Checking if window is at edge (bottom or right)
            else if (y+j >= dimensions[0]){
              if ((x+i+colour) >= dimensions[1]){
                window[colour][counter] = input_arr[y+j-window_y]
                                                          [x+i+colour-window_x];
              }
              else
                window[colour][counter] = input_arr[y+j-window_y][x+i+colour];
            }
            else if (y+j < dimensions[0] && (x+i+colour) >= dimensions[1]){
              window[colour][counter] = input_arr[y+j][x+i+colour-window_x];
            }

          }
          counter++;
        }
      }
      //For each input.component, get the Median so image doesnt change colours
      for (int colour = 0; colour < dimensions[2]; colour++)
        sort(window[colour], window[colour]+sizeof(window[colour])/
                                                    sizeof(window[colour][0]));

      for (int colour = 0; colour < dimensions[2]; colour++)
        out_arr[y][x+colour] = window[colour][int(counter/2)];

      counter =0;
    }
  }
  //Send back to Master
  MPI_Send(out_arr,dimensions[0]*dimensions[1], MPI_CHAR,0,TAG,MPI_COMM_WORLD);
```

Code 3. Median Filter and Edge Processing

```
//Waiting for slaves to finish and placing into output array
  for (int j=1; j<numprocs; j++){
    unsigned char slave [y_portions[j-1]][Input.Width*Input.Components];
    // MPI_Recv is blocking
    MPI_Recv(slave, y_portions[j-1]*Input.Width*Input.Components, MPI_CHAR, j,
                                                    TAG, MPI_COMM_WORLD, &stat);
    // The following checks eliminate black lines between slave arrays
    if (j==1)
      for(int y = 0; y < y_portions[j-1]- (int)WINDOW_Y/2; y++)
        for(int x = 0; x < Input.Width*Input.Components; x++)
          Output.Rows[y][x] = slave[y][x];

    else if(j==numprocs-1)
      for(int y = (int)WINDOW_Y/2; y < y_portions[j-1]; y++){
        if (j > 1 && y == (int)WINDOW_Y/2)
          indexer += y_portions[j-2]-WINDOW_Y+WINDOW_Y/2;
        for(int x = 0; x < Input.Width*Input.Components; x++)
          Output.Rows[y+indexer][x] = slave[y][x];
      }
    else if(j == 2){
      for(int y = (int)WINDOW_Y/2; y < y_portions[j-1]- (int)WINDOW_Y/2; y++){
        if (j > 1 && y ==(int)WINDOW_Y/2)
          indexer += y_portions[j-2]-WINDOW_Y/2-WINDOW_Y/2;

        for(int x = 0; x < Input.Width*Input.Components; x++)
          Output.Rows[y+indexer][x] = slave[y][x];
      }
    }
    else
      for(int y = (int)WINDOW_Y/2; y < y_portions[j-1]- (int)WINDOW_Y/2; y++){
        if (j > 1 && y ==(int)WINDOW_Y/2)
          indexer += y_portions[j-2]-WINDOW_Y+WINDOW_Y/2;
        for(int x = 0; x < Input.Width*Input.Components; x++)
          Output.Rows[y+indexer][x] = slave[y][x];
      }
  }
```

Code 4. Reassembly of Processes Arrays

C. Experiment Procedure

The JPEG received as input was split according to our even partitioning memory model and sent to each slave sequentially. The Master then blocks while waiting for each slave to return the median filtered arrays. It is compiled at the master process and written to an output image.

In terms of measuring speedup, the filtering was run sequentially using only 1 process for the entire image at a window size of 3, 5 and 7. The same image was filtered again but using 4 slave processes and the master process at the same window sizes. For each implementation, we made use of cache-warming and ran the program 10 times each, taking the average for each window size. The results of these tests are shown in Table I. It was expected that the performance of the distributed processing implementation would perform much better than the serial version since processing can take place simultaneously. It was also expected that the Speedup of the MPI implementation would increase as the window size increased.

The MPI version of the program was run again at a window size of 3, while varying the number of slave processes from 2 through to 100. Again for each number of processes, cache warming was taken into account and the average time from 10 runs was taken. The results are shown in Figure 1. It was expected that the performance would increase in speed until five processes were used, including the master process as the assumption was that each slave process would run on 1 core. It was thought that the performance should decrease from there as overhead starts to come in to play. The overhead includes and sending and receiving messages and arrays to and from the master process and reassembling the image.

III. Results

A. Tables

Table 1 below shows the comparison between the MPI implementation and the series implementation. The speed up appears to change as the window size changes in a way which does not match with the expectation asserted previously. It does however indicate that it might be beneficial in a practical implementation to find an ideal number of slave processes for a particular window size.

TABLE I
MPI vs Serial Avg Times at 49 Slave Processes

Window Size	Avg Serial Time[ms]	Avg MPI Time[ms]	Speedup
3	18.27	11.87	6.212
5	67.56	73.92	0.9140
7	180.1	156.76	1.148

B. Figures

It also does not match with expectation that the time to process the image does not increase as the number of slave processes surpasses the number of cores on the machine. Looking at Figure 1, the time actually continues to decrease as the number of slaves processes increases.

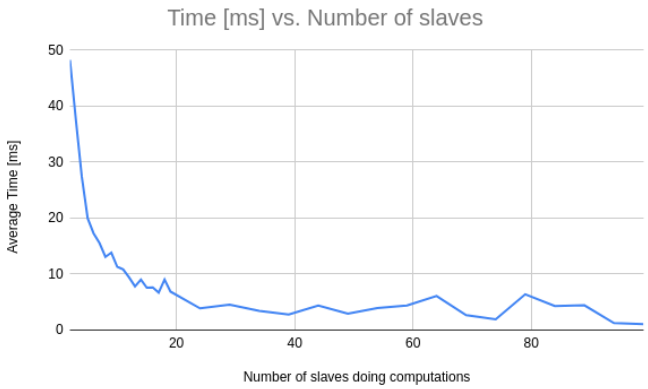

Fig. 1: Average MPI Computation Times vs Number of Slave Processes [Window size 3]

This means that the overhead as defined in the previous section does not become significant despite the number of processes. It can be seen, however, that the time decrease from 20 processes to 100 has a smaller gradient than the early part of the graph. This can be explained by the fact that the 2D arrays are forced to a constant size due to the size of the window. The image used for the test had a height of 256 pixels which means that the height of the window would force the 2D array partition to have a minimum height the same as the height of that window. This would cause the time taken to process that partition to remain near constant.

IV. Conclusion

The above practical has attempted to explore the use of Message Passing interfaces in an image processing application: Median Filtering. It has done this by deciding on and explaining a memory model and choosing an edge processing method and describing it. The distributed filtering was then implemented using the message passing interface: OpenMPI. It has shown that the MPI implementation outperforms a serial version for the same window size and has shown that the speedup increases as window size does. It has also examined the effects of adding more slave processes than there are physical processors on the hardware and compared them to expectation.