# Practical 3: MPI

Clara Esther Stassen[†]
EEE4120F Class of 2020
University of Cape Town
South Africa
[†]STSCLA001

***Abstract*—This report is an analysis of the performance of an MPI implementation of a median image filter compared to a sequential version and how the MPI overheads affect performance.**

## I. Introduction

Message Passing Interface (MPI) is a standardized message passing library used in systems with distributed memory. The model involves breaking computation amongst a set of tasks that each have their own local memory. As there is no global memory available to these tasks, communication is facilitated through MPI via explicitly sending messages. For this experiment, only a single CPU was available, therefore a distributed memory environment was emulated with memory virtualisation achieved by the use of the memory management unit. This report uses an image median filter algorithm as a 'dummy problem' to investigate the speedup achieved when using MPI with an increasing amount of computational tasks. These experiments are done by setting up MPI to do all of its communication via a master node which breaks up the computational problem, sends data to each slave, receives the computed data and then reassembles it.

## II. Methodology

### A. Hardware

The hardware used is an Intel Core i5-7200U CPU with 4 cores.

### B. Data Partitioning

The image data was split up amongst the slaves by calculating the amount of rows to send to each slave based on what size the filter window should be and how many slaves there are. Each slave was sent an amount of rows equivalent to the height of the image divided by the number of slaves, plus, (if for example the window size is 3x3), an extra 4 rows, two at the top of the partition and two at the bottom. The method to this is shown below.

```
 // Start Row for Slave
int ystart = (j-1) * Input.Height / slaves;
// End Row for Slave
int yend = ceil((float)Input.Height / slaves + ystart);
// Number of Image Rows in Partition
int numrows = yend - ystart + 1;
// Number of Image Rows Needed for Filter
int windowrows = numrows + windowsize - 1;
// Send Buffer
char buffer[windowrows][Input.Width*Input.Components];
// Partition Data
int cnt = 0;
for (int i = ystart - (windowsize / 2); i < yend + (windowsize / 2); i++)
{
 if (i < 0 || i >= Input.Height)
 {
  for (int j = 0; j < Input.Width * Input.Components; j ++)
  {
   buffer[cnt][j] = 0;
  }
 }
 else
 {
  for (int j = 0; j < Input.Width*Input.Components; j ++)
  {
   buffer[cnt][j] = Input.Rows[i][j];
  }
 }
  cnt++;
}
```

### C. Sending Data to Slaves

In order to send the image data to the slaves, the master needed to first send a message to the slave communicating what the size of the data is. This was done through sending a message of a known fixed size - an array of length 7 - that contained information the slave would need for its processing such as the windowsize, width of the image, and number of rows it needs to receive.
The master should then receive an ACK from the slave confirming it has received this information, and then send the actual image data, and then receive another ACK. The slave will have then received all the data it needs to do its computation.
The code excerpt below shows how the master sends data to the slaves, and this code is repeated for all slaves in a for loop.

```
int info[7];
info[0] = windowsize;
info[1] = windowrows;
info[2] = numrows;
info[3] = Input.Width;
info[4] = Input.Components;
info[5] = ystart;
info[6] = yend;
// Send Information and Data Size
MPI_Send(info, 7, MPI_INT, j, TAG, MPI_COMM_WORLD);
// Receive ACK
MPI_Recv(ack, 1, MPI_INT, j, TAG, MPI_COMM_WORLD, &stat);
// Send Data
MPI_Send(buffer, windowrows*Input.Width*Input.Components, MPI_CHAR, j,
                                                    TAG, MPI_COMM_WORLD);
// Receive ACK
MPI_Recv(ack, 1, MPI_INT, j, TAG, MPI_COMM_WORLD, &stat);
```

### D. Processing the Data

The image data was processed and filtered by the slaves through iterating through each pixel to create a 'window' of

neighbouring pixels of a specified size, finding the median of the window, and then replacing the pixel with that. As the images processed were in colour, the loop had to jump by 3 pixels to find neighbouring pixels in order to ensure each colour component was processed separately. The code below shows how the window was created through iteration.

```
  // Filter Data
 int r = 0;
 for (int y = windowsize / 2; y < windowsize / 2 + numrows; y++)
 {
  for (int x = 0; x < width * components; x++)
  {
   int window[windowsize * windowsize];
   int cnt = 0;
   for (int i = y - windowsize / 2; i <= y + windowsize / 2; i++)
   {
    for (int j = x-components*(windowsize/2); j <= x+components*(windowsize/2);
                                                          j = j + components)
    {
     if (j < 0 || j >= width * components)
     {
      window[cnt] = 0;
     }
     else
     {
      window[cnt] = in[i][j];
     }
     cnt++;
    }
   }
   int m = median(window, windowsize * windowsize);
   out[r][x] = m;
  }
  r++;
 }
 // send to rank 0 (master):
 int par[3];
 par[0] = ystart;
 par[1] = yend;
 par[2] = numrows;
 // Send Data Size to Master
 MPI_Send(par, 3, MPI_INT, 0, TAG, MPI_COMM_WORLD);
 // Send Data to Master
 MPI_Send(out, numrows*width*components, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
```

The following code was used to find the median value of the window through using a quicksort algorithm.

```
  // Compare Function for Quicksort
int compare(int* i,int* j) { return (*i>*j); }
// Median Function
int median(int* window, int size)
{
 qsort(window,size,sizeof(int),(int (*)(const void*,const void*))compare);
 return window[size/2];
}
```

### E. Reassembling Data

When receiving the processed data from the slaves, again an initial message of the size of the data (rec_info) had to be received. This message contained the information of the number of rows to receive and the start and end row indexes each slave had been given from the initial image. The master then receives the processed rows and inserts them into the output image using the indexes provided. The master does this in a loop for all slaves so that the data can be reassembled and outputted. The code below shows how this is done.

```
  // Receive Slave Data
 for (int j = 1; j <= slaves; j++)
 {
  // Receive Size of Slave Data
  int rec_info[3];
  MPI_Recv(rec_info, 3, MPI_INT, j, TAG, MPI_COMM_WORLD, &stat);
  int ystart=rec_info[0];
  int yend=rec_info[1];
  int numrows=rec_info[2];
  // Receive Data
  char output[numrows][Input.Width*Input.Components];
  MPI_Recv(output, numrows*Input.Width*Input.Components, MPI_CHAR, j, TAG,
                                                        MPI_COMM_WORLD, &stat);

  // Save Received Data to Output
  int cnt = 0;
  for (int i = ystart; i < yend; i++)
  {
   for (int j = 0; j < Input.Width * Input.Components; j ++)
   {
    Output.Rows[i][j] = output[cnt][j];
   }
   cnt++;
  }
 }
 // Write the output image
 if (!Output.Write("Data/Output.jpg"))
 {
  printf("Cannot write image\n");
  return;
 }
```

### F. Timing

A sequential implementation of the code that does not use MPI was created and timed using the tic() and toc() functions to provide a golden measure for comparison. The MPI code was timed by using tic() and toc() for the processing sections of the code, as well as the Unix 'time' command to provide an account for the total run-time of the code which is shown below:

```
#!/bin/bash

ts=$(date +%s%N)
mpirun -np 5 bin/Prac3
echo "Total runtime:"
echo $((($(date +%s%N) - $ts)/1000000))"
```

This allows us to gain some insight into the time it takes to set-up and destroy the MPI tasks. When recording the timing of the code, multiple runs were timed and averaged to achieve an accurate measure of the performance, and the first run-time discarded to account for cache-warming.

## III. Results

The three figures below show how the size of the median filter window affects the output image. It is evident that a larger window size degrades the quality of the image, as Figure 3 appears considerably more blurry than Figure 2.

Fig. 1: Original Image

Fig. 2: Filtered Image with 3x3 Window

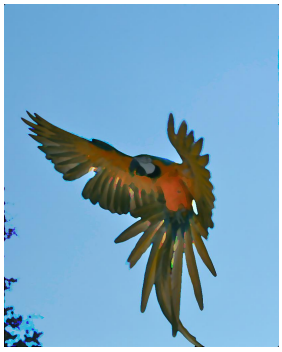

Fig. 3: Filtered Image with 9x9 Window

Figure 4 below shows the speedup obtained using MPI compared to the sequential golden measure across an increasing number of nodes and for two different window sizes. It is evident that for a window size of 3x3, the problem size is too small to benefit from the MPI parallelism and thus the performance is worse than the sequential version as the speedup is always below one. This is due to the MPI overheads such as setting up the slaves and communicating with them dominating the execution-time and therefore it is not worth running this problem size with MPI. However, with a window size of 9x9 the problem is large enough to benefit from parallelism and the MPI implementation starts to give a faster run-time past 3 nodes, and increases gradually as the number of nodes increases.

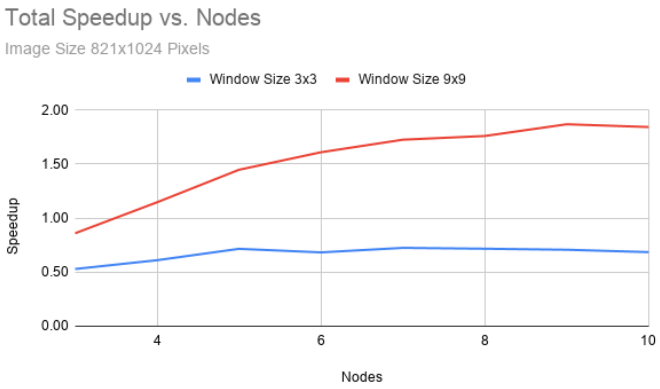


Fig. 4: MPI Speedup

It is worth noting that the processing time for the MPI implementation is always faster than the sequential version, however the MPI set-up and destroy time results in the total execution being slower than the sequential version. Figure 5 shows how although the total speedup of the 3x3 window MPI was never greater than one, the processing speedup does increase as more nodes are used for processing.

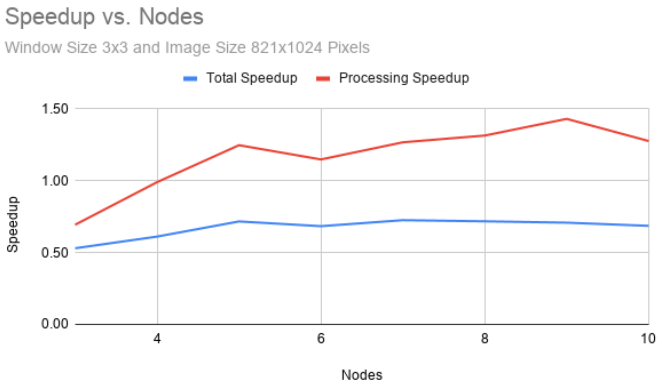


Fig. 5: Processing Speedup vs. Total Speedup

## IV. Conclusion

From this investigation it can be confirmed that using MPI for problems that are large enough does result in increased performance, however using MPI introduces many overheads such as set-up and destroy time, and communications, that have a negative impact on performance and could dominate the run-time for smaller problems. Further experimentation could be done to determine how images with various detail levels affect performance and how the MPI overheads scale with increasing number of nodes.