Parallel Programming in Raspberry Pi Cluster

A Design Project Report

School of Electrical and Computer Engineering of Cornell University

In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical and Computer Engineering

Submitted by:

Vishnu Govindaraj

Date Submitted:

May 24, 2016

MEng Project Advisor:

Dr. Joe Skovira

Contents

Project Title:
Parallel Programming in Raspberry Pi Cluster

Author:
Vishnu Govindaraj

Abstract:
The aim of the project is to build a Symmetric Multi-Processing (SMP) and Asymmetric Multi-Processing (ASMP) Platforms to develop Parallel Applications. The Project is planned to be used as a Lab project for the course "**ECE5725 – Design with Embedded Operating System**". Aimed at providing the students with exposure to SMP and ASMP in an inexpensive but solid way, the Project was built using Raspberry Pi 2 Model B embedded computers. Raspberry Pi 2 being a four core processor with shared physical memory acted as an excellent platform for SMP. Meanwhile, for ASMP, a server cluster was built by connecting four Raspberry Pi's using an Ethernet network router. Sobel Filter Edge Detection was chosen as a target application to analyze the performance and also to be used as a lab exercise for the course. Sobel filter with its two Dimensional array computation turned out to be an excellent application to learn the corner cases that are associated with the parallelization of a program. SMP and ASMP Programs, both individually and also as a hybrid, achieved the four and ten times better performance than sequential programming respectively.

Report Approved By,
Project Advisor: Prof.Joe Skovira

Executive Summary:

This project accomplished the goal of building Symmetric and Asymmetric Multiprocessing Platforms with a cheap embedded computer Raspberry PI. As planned, the project also achieved in extracting the desired performance from the target application which is “Edge Detection of Image using Sobel Filter”. By choosing a target application with enough parallelization complexities, this project managed to explore the corner cases and came up with possible optimization and resolution techniques that one should be aware of when parallelizing an application. This project also at multiple point proved that parallel program can go wrong not only with the incorrect output but also with performance by taking more time when compared to the sequential program. The lab which is designed based on this project will give students with enough opportunities to learn the corner cases associated with the parallel programming.

Critical Accomplishment from this project include Building an ASMP platform by making correct design choice. During initial phases of this project, Design options available to accomplish the goal turned out to be very broad. Especially with ASMP, Understanding different methods like NUMA, Message Passing, Hadoop and making a decision on its feasibility happened to be difficult. By choosing the Message Passing Multicomputer model for ASMP, this project achieved its goal by proving the performance improvement when compared to sequential programming. Another design issue arose while splitting the image between the nodes. The available options to split the image and end decision made were discussed extensively the ASMP section of this document.

This document covers all the design choices and decision made during the development and also lists the performance analysis and comparison among different models developed in this project. Finally by comparing the Hybrid Parallel applications of openMP and MPI to a sequential program running in High Frequency and High performance server, this project proved that using parallel program, it is possible to achieve better performance by building a system which is ten times less costly than a High performance machine running a sequential program.

1. Introduction:

"Survival of the Fittest" may be the law that governs the human evolution, but "Survival of the Performance" has always been the law for Computing and its related technologies. When the Mainframes become high maintenance and less accessible, Personal Computers evolved. When Structural Programming languages failed to provide required abstractions, Object oriented programming languages evolved. When high frequency scaling of processors resulted in expensive cooling solutions for the computers, Multi core processors evolved. Similarly, at this point in computing timeline where size of the data is growing at an exponential rate due to the advent of technologies like Internet of Things, Natural Language Processing, Machine Learning and Virtual Reality, Parallel programming is necessary to attain the satisfying performance. The input data given to the computers are no more the Keystrokes from a Keyboard or a mouse click. Input data has evolved to Voice commands, gestures and facial expressions which are read by sensors which in turn produces large amount of data to be processed. So far, sequential programming was able to give the desired performance as the input data was small. With the large input data, which needs to be processed by small processors distributed in a system, parallel programming becomes inevitable. Though, may not be in the same form as the design used in this project, parallel programming is going to move from clouds and big server clusters to the local distributed embedded processor space. It becomes necessary for the students to understand the complexities and corner cases associated with parallelizing the data. Keeping the above statements as a goal, in this project a server cluster was built using a very cheap embedded computer called Raspberry Pi. A Parallel application with enough parallelization complexities was built to analyze its performance when it ran in "Symmetric" and "Asymmetric" Multiprocessing System. The Application chosen to be parallelized was "Edge Detection using Sobel Filter". The analysis and comparison of the sequential and parallel programming with increasing image size were made and the results came out as expected. The programs will be also used as the lab exercise for the course "**ECE5725 – Design with Embedded Operating System**".

2. Design Choices:

2.1. Parallel Programming Model:

The initial plan was to build the server cluster first and then come up with applications targeting the cluster. The parallel applications can be either of below two types,

- Asymmetric Multiprocessing: Each Processor will be assigned to different tasks. One master processor will control all worker processors by sending and allocating task.

- Symmetric Multiprocessing: Physical memory is shared between the processors. No boss worker model. All processors act like peers.

It was decided between me and the project advisor to develop parallel applications that achieves both SMP and ASMP separately. Once both the models worked individually, It was

planned to build an application that is Hybrid of both SMP and ASMP. When it comes to memory access, i.e. Memory Access between the nodes, it can be either of below two types:

- Shared Memory with Non Uniform Memory Access (NUMA): Where memory access time depends on the memory location relative to the processor. Each Nodes will be connected to shared memory using High speed Interconnect.

- Message Passing Multicomputer: No processor can directly access another processor's memory. The data will be shared using Message Passing Libraries like MPI (Message Passing Interface).

Between these two memory access options, Message Passing Multicomputer was chosen as it is less costly and does not require any complex hardware like High Performance Interconnect or a Memory Controller. Beowulf clustering which is an ideal Multi computer architecture was chosen to achieve the Message Passing between the nodes. It is a system which usually consists of one server node, and one or more client nodes connected via Ethernet or some other network. It is a system built using commodity hardware components, like any PC capable of running a Unix-like operating system, with standard Ethernet adapters, and switches. It does not contain any custom hardware components and is trivially reproducible. Beowulf also uses commodity software like the FreeBSD, Linux or Solaris operating system, Parallel Virtual Machine (PVM) and Message Passing Interface (MPI).

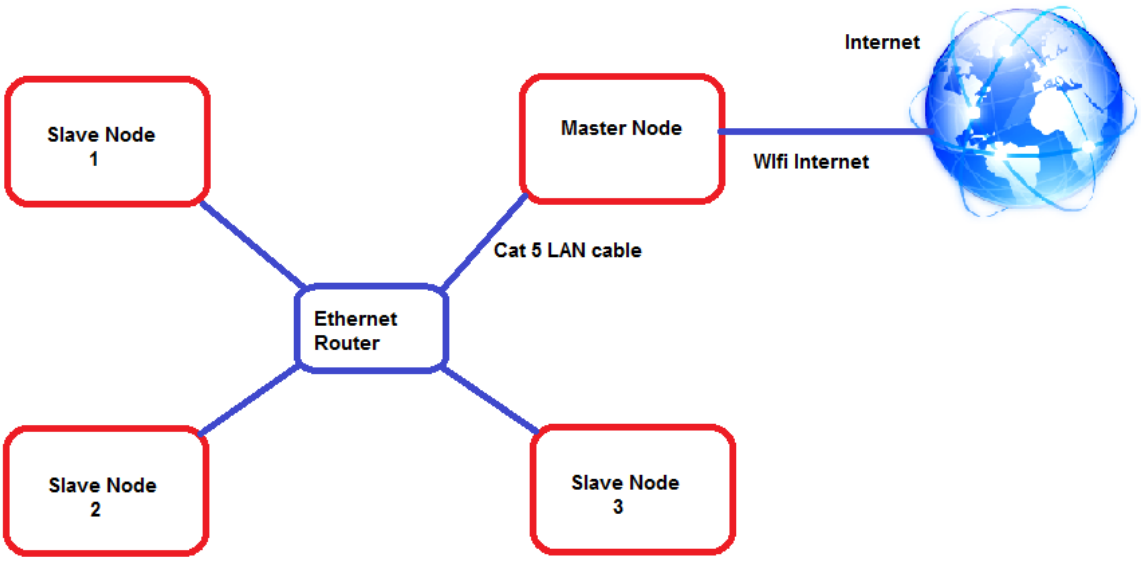

Figure 1: Beowulf Configuration for Message Passing Multi Computer model.

2.2 Technologies:

openMP(open Multi Processing) a technology exclusively available for shared memory multi-processing was chosen to build parallel applications for SMP. It consists of compiler directives and library routines for parallel application programming. All the memory threads of same parallel program will be sharing same address space.

OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer. It is an implementation of multithreading, a method of parallelizing whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and the system divides a task among them.

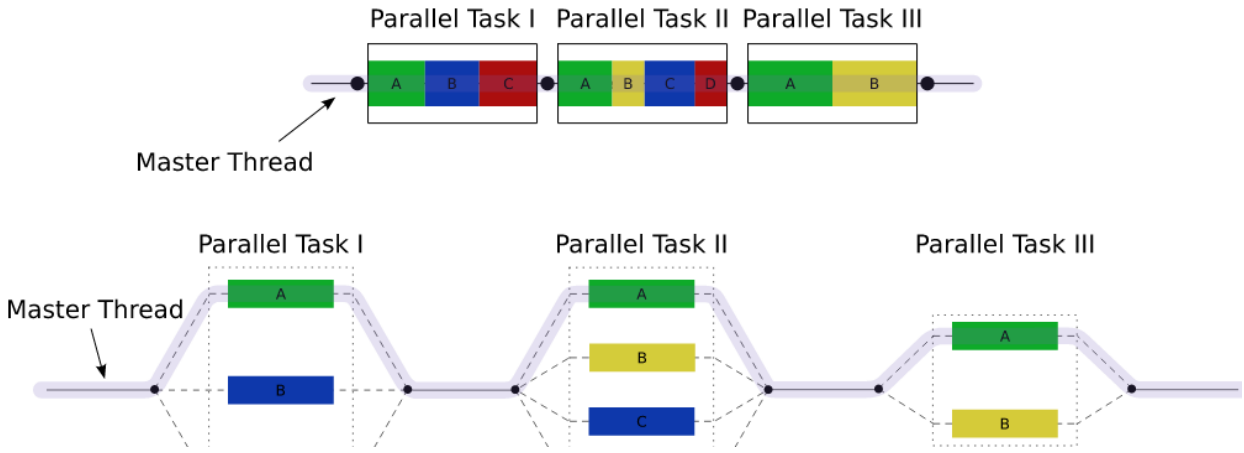

Figure 2: Illustration of multithreading using openMP. *Figure from Reference Number 6*.

For Asymmetric Multiprocessing, MPI (Message Passing Interface) was the technology chosen. It is a standardized and portable message-passing system that defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran. MPI is a communication protocol for programming parallel computers. Both point-to-point and collective communication are supported.

When it comes to MPI, there are two options available and they are MPICH and openMPI. MPICH was chosen as it is more widely accepted and used when compared to openMPI. Mpi4py is another implementation exclusively for python programmers is also available.

For this project, the language of choice for both SMP and ASMP Implementation is C

3. Building the Server Cluster:

Raspberry Pi with it four core processor, itself acts as a perfect platform for Shared Memory Processing. In accordance with the design choices made, Message Passing Multi Computer was built with Beowulf configuration as explained in the section 2.
Please refer the Appendix for the detailed installation steps for Linux and MPI. The steps include Linux installation from ECE 5990 – Embedded operating systems Course Lab 1.
Once completed, it should be possible to ssh into any slave node without giving password.
The wifi adapter was plugged into the Master Node to connect to the internet as Raspberry Pi has only one Ethernet port which was already utilized for connecting to other routers. With this Beowulf configuration of server cluster was completed and a SMP and ASMP platform to develop parallel programs was ready.

Below are the issues that arose while building the server cluster,

1) Each of the single step in Mpich is to be followed in correct order. Missing one of those steps lead to multiple confusion regarding compatibility of the system and correctness of the steps. Finding the correct steps to install mpich was the initial challenge faced.
2) SSH configuration was done wrongly many times resulting in master and slave nodes not able to communicate between each other.
3) Initial wifi setup failure lead to a situation were different makeup commands are added to the network configuration file of kernel leading to Ethernet not working in the Master node. This was resolved by installing the Linux again as advised by the Project Advisor.

Figure 3: Final Setup of 4 Node Beowulf Cluster.

4. Edge Detection using Sobel Filter - Sequential Program:

The Application chosen to be parallelized was Edge Detection with Sobel Filter.
Edge detection with sobel filter was chosen because it process two dimensional array of a grey scale Image. Greyscale Images of type pgm was chosen because each pixel acts as a single sample; it is easy to load it into an array to process. It provides parallelization complexities like,

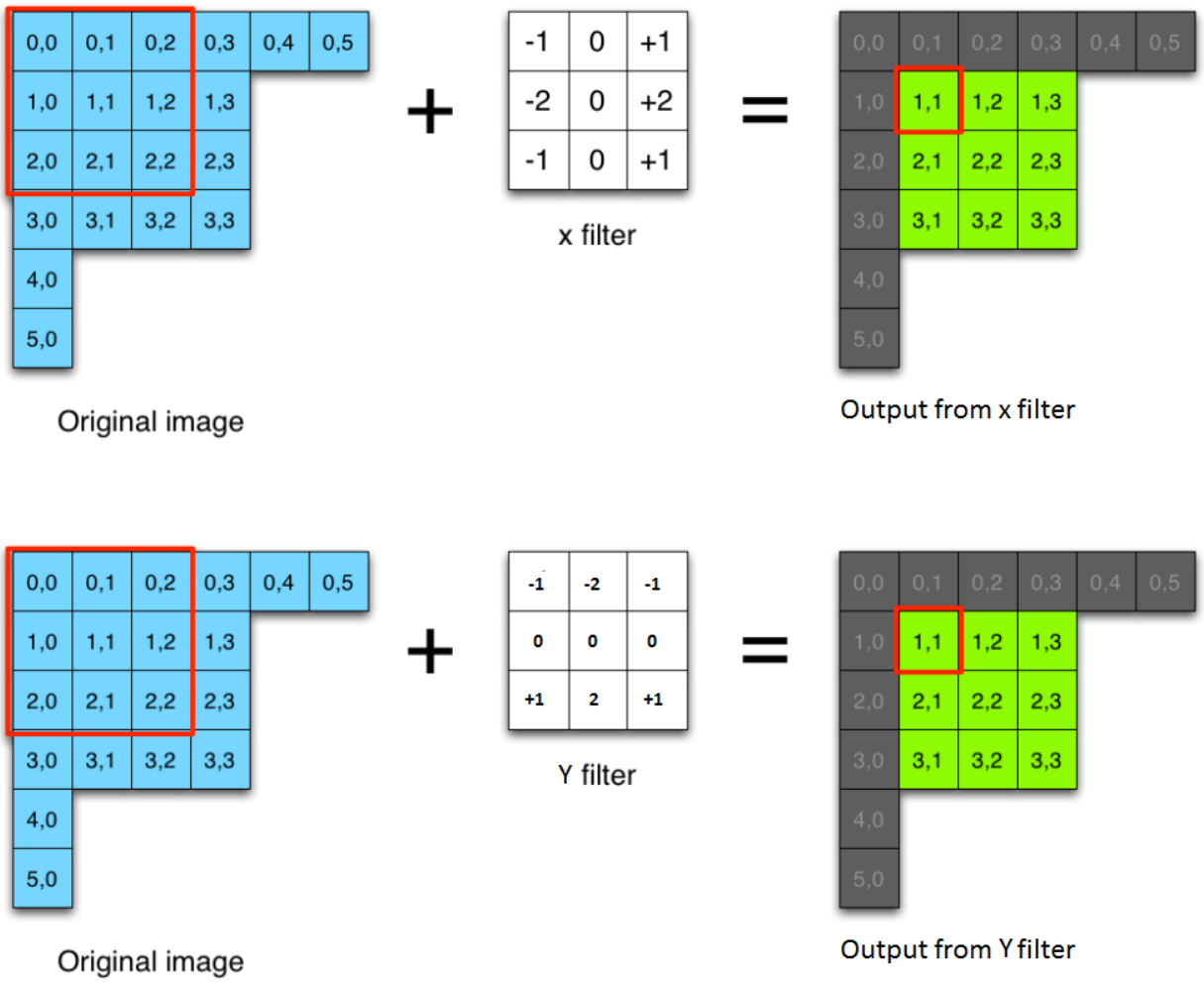

Figure 4: Convolution of Image Array using X and Y filters. Convolution requires pixel surrounding it to get the convolved pixel. *Figure from Reference Number 11*

In case of SMP, How to divide the two dimensional Image array and assign them to each thread.
In case of ASMP, it should be noted that for convolution of a pixel we need the pixels surrounding it. Check figure 4, the output pixel 1,1 is obtained by the convolution which involved all the pixels surrounding it. If we are dividing the Image array symmetrically either along y axis as shown in Figure 5 or x axis, and sent them to different nodes of the cluster, the edges in the image needs pixels from the previous row which is not local to that particular node. As shown in figure 5, The first and last row of second chunk needs last and first row of first and third chunks respectively.

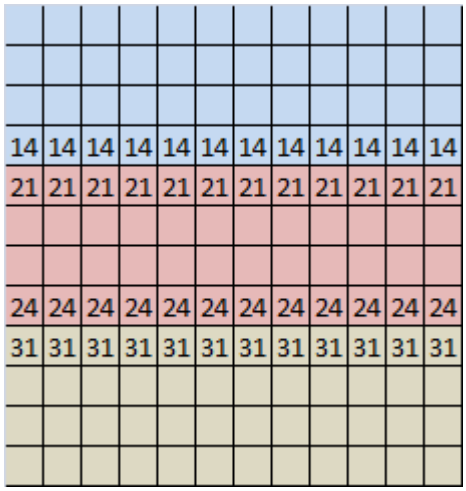

Figure 5 : Two Dimensional Array Representation of an Image where an image is symmetrically divided between three nodes. 14 – Fourth row of first chunk, 21 – Forst row of second chunk, 24 – fourth row of second chunk and 31 – First row of third chunk.

The Sequential Program of the Sobel filter consists of a below files,

mypgm.h - Header file for importing and exporting Image into Static Unsigned Char Array. It has following methods to process the Image,

Load_Image_Data:
This method is called from the main method in sobel.c. It loads an Image file of type “PGM” into a static two dimensional unsigned character array of Size 4128 * 4128. This method will also prompt the user to provide the name of the Input Image that needs to be processed.

Load_Image_File:
Same as that of Load_Image_Data, except that file name can be sent to the method as the input parameter.

Save_Image_Data:
Save the output from sobel filtering into an image file. The method will prompt the user to specify the name of the Image File to which the output needs to be saved.

Save_Image_File:
Same as the Save_Image_Data, except that name of the file can be sent as an input parameter to the method.

sobel.c - Source file for Sobel Filtering. Below are the methods in sobel.c

main:
Main method loads the image data into a static two dimensional array of unsigned characters by calling either Load_Image_Data or Load_Image_File function.

Once done sobel filter method is called which takes care of Image convolution with sobel Operators.
The output two dimensional array from the sobel_filtering method is then saved as an output Image file by calling either Save_Image_Data or Save_Image_File method.

sobel_filtering - Method where the Loaded Image is convolved with below 3 × 3 kernel for X and Y coordinates. Please refer

$$\mathbf{G_x} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G_y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & 2 & +1 \end{bmatrix} * \mathbf{A}$$

Where A is the Image Input, * denotes the convolution operation (Please refer figure 4, to check how convolution works), Gx - Convolved output using X coordinate kernel and Gy is the output from convolution using Y coordinate Kernel.

The convolved outputs are then combined using the below operation, Where G is the final output Image.

$$\mathbf{G} = \sqrt{\mathbf{G_x}^2 + \mathbf{G_y}^2}$$

For explanation and illustration purposes, below image will be used throughout the document. The application saves the below image in a two dimensional array and by applying convolution using sobel operators for x and y coordinates, the application outputs the Image in figure 7. The code does the X coordinate kernel convolution and Y coordinate kernel convolution and then calculates the root mean square value of both the results to save the final Image.
More test images and its results will be discussed in Performance Analysis Section of the report.

Figure 6: The Image which is used to test during different Implementation of the Parallelization.

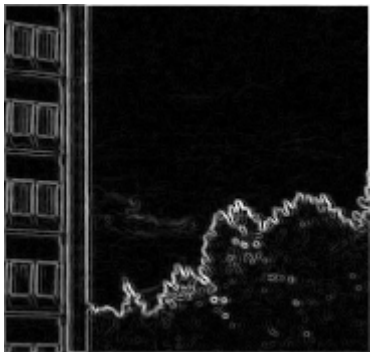

Figure 7: Output Image of Sequential program which will be used as a benchmark for correctness of different Parallel Program implementations.

Average time taken for the Sequential Program in Raspberry Pi single node for the above image is 0.12 seconds.

Initial implementation convolved the image starting from second row and finished the convolution with row above the last row. This was due to the fact that Edges of the image does not have enough pixels to convolve. This issue was later overcame by bloating the Input array with added rows and column to the sides. This resulted in Image size growing with 2 units on both the dimensions.

5. Symmetric MultiProcessing using openMP:

OpenMP compiler directives can be added to a program by simply including the header file “omp.h” in the program and “fopenmp” command along with compile command. Since, the major computation part of the Sobel Filtering lies in the “sobel_filtering” method in sobel.c, it was decided to parallelize the method.
The method consists of the two four level nested for loops, one for finding the Maximum and Minimum of output pixels and other for saving it after applying the convolution. For Normalization of the pixels at the end of filtering it was necessary to compute the Maximum and Minimum pixel size and then to convolute again to apply normalization. First of the two for loops is given below,

```
for (y = 1; y < y_size1 + 1; y++) {  //y_size1 – Input Image height
     for (x = 1; x < x_size1 + 1; x++) {   //x_size1 – Input Image Width

       pixel_value_x = 0.0; // Variable for X coordinate convolution output
       pixel_value_y = 0.0; // Variable for Y coordinate convolution output
       pixel_value = 0.0;   // Variable for final output

      // Convolution of a single pixel – below for loop reads pixel value
      //from all surrounding loops
      // for convolution

       for (j = -1; j <= 1; j++) {
         for (i = -1; i <= 1; i++) {
```

```
            // weigth_x denotes kernel matrix for x coordinate
            // weigth_y denotes kernel matrix for y coordinate

            pixel_value_x += weight_x[j + 1][i + 1] * image1[y + j][x + i];
            pixel_value_y += weight_y[j + 1][i + 1] * image1[y + j][x + i];

          }
        }

        //Root mean square value of X and Y coordinate output.

        pixel_value = sqrt((pixel_value_x * pixel_value_x) + (pixel_value_y *
pixel_value_y));

        // Max and Min pixels for normalization.
        if (pixel_value < min) min = pixel_value;
        if (pixel_value > max) max = pixel_value;
      }
    }
```

Sobel – Sequential: Four Level Nested For loop in sobel filtering method.

Any omp construct can be added by using a “pragma” compiler directives. Any section of the code which is to be parallelized among the thread will have “#pragma omp parallel” as default compiler directive. The directive hints the compiler that, the code section written inside “omp parallel” should be parallelized among the threads.

The first idea that came was to parallelize the inner loops simply using “*omp parallel for*” construct. In this way the pixel value all pixels involved in the convolution will sent to different processors. But the “omp parallel for” loop can be applied to only one loop, if there is a nested loop, then “collapse” clause can be added to the “omp parallel for” directive during the compilation of the program. It has a condition that not operations should be added between the two for loops. So, I added the below construct to the inner “for loops” assuming that each pixels involved in the convolution will be assigned to different processors.

```
#pragma omp parallel for collapse (2)
```

The above collapse clause in the omp construct instructs the compiler to collapse the first two for loops. The number is passed as an argument to the clause. It should be noted the for loops collapsed should not have any instruction or commands between them.

For compilation of program with openmp constructs it is necessary to add “–fopenmp” command along with gcc command.

When I ran the application, the output for Image in Figure 6 turned out to be as shown in Figure 8.

Figure 8: Wrong Output Image from Initial openMP Implementation.

The above Image is full of grains and not same as the output image from sequential programming. When parallelizing a code, the output should always match the output of sequential program which is not true in the case of above image.

I realized that the variables pixel_Value_x and pixel_value_y is being shared by all the threads and output of one thread is changing the output of another thread. This issue of some variables being required to be private for the threads whereas some needs to be shared can be resolved using openMP clauses called “Private” and “Shared”. Also, openMP also provides a clause called reduction, which makes thread to keep a private copy of the variables which are passed as its parameter and add the results of those private variables at the end of parallel section. Reduction can also be applied for multiplication and other Boolean functions.

```
#pragma omp parallel for collapse (2)\
reduction(+:pixel_value_x,pixel_value_y)
```

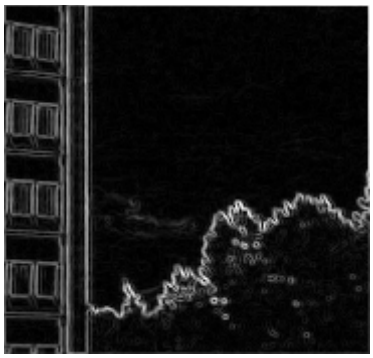

Figure 9: Output Image which matched the Sequential output after applying reduction.

The time taken by the Parallel Program with reduction applied to it was 0.38 seconds, which is around three (3.16) times the time taken for sequential program. This was opposite of what the parallelization should have achieved.

The analysis on what happened necessitated the correct understanding of sobel convolution, Processor and Cache architecture using which the program is running. Since the Inner two loops in the sobel filtering is reading all the surrounding pixel values and computing the output value, each processor will pick one of these pixel value to compute the final value. It should be

noted that the raspberry pi is using ARM cortex A7 Processor. Each core has its own L1 cache of Size 16 KB and each has a cache line of size 32 bytes. It should be also noted that, size of unsinged char being 1 byte, the cache line can hold the next 8 pixels considering 4 byte address space. By exploiting this spatial locality, sequential program is able to achieve 0.12 seconds. But, by dividing the convolving pixels to each processor, the above parallel program is making the processor to fetch the pixels every time from the main memory as the pixel will not be in the cache line as the previous pixel is fetched by a different processor. Higher physical memory access rate when compared to the sequential program made this program taking three times longer when compared to the Parallel Program. This study was an excellent example of how parallel programming can corrupt the cache consciousness of program.

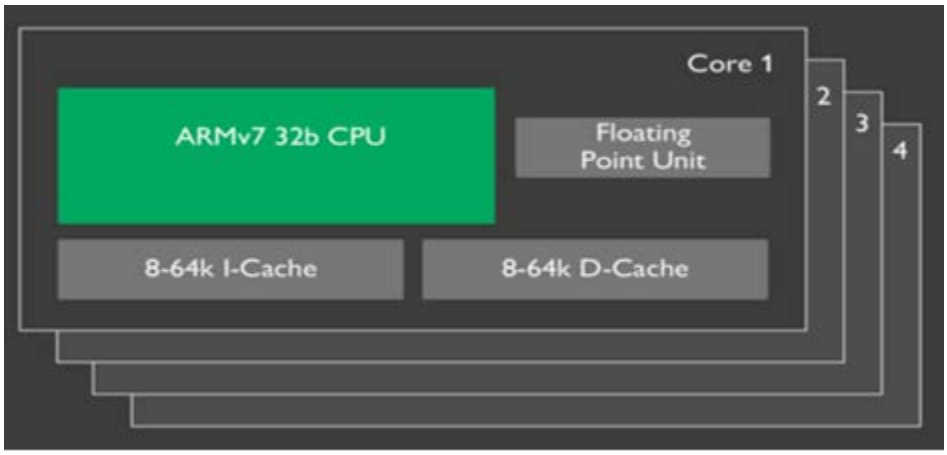

Figure 10: Cortex A7 Processor Architecture used in Raspberry Pi B 2 Models.

To avoid this problem of parallel program disturbing the cache consciousness of the underlying processor, it was decided to distribute the whole convolution itself rather than distributing each of convoluting pixels to different processor. This can be done by removing the “omp parallel for” from the inner loops and applying the same to the outer two loops.

It should be noted that the pixel_value_x and pixel_value_y and pixel_value_y should be private variables for each thread. If not private then the race condition may occur. Similarly the variables min and max needs to update the maximum and minimum pixels for every iteration. This data needs to be shared so that max and min values can be computed by checking the max and min updated by the other processors. openMP provides “private()” and “shared()” clauses that can be added to “omp for” construct where private and shared variables can be included. It should be noted that, the race conditions may occur as the variable are being shared. It is necessary to make the section where max and min are updated as “critical” section. After applying constructs as shown in Sobel OMP parallel code section, the application was able to achieve a time performance of 0.043 seconds. It took nearly three times (2.79) less time when compared to sequential program and also was able to achieve the resultant output same as that of sequential program. The expected speedup was not achieved owing to the overhead associated with openMP when compared to the processing time required for smaller size Image. Please refer Table 1 in section 7 where nearly four times speedup is achieved because of openMP overhead being small when compared to the processing time required for the image.

```
#pragma omp parallel for collapse(2)
private(i,j,pixel_value,pixel_value_x,pixel_value_y)\
                                        shared(min,max)
//private – variables that are private to each thread. Shared – variable
whose values can be\
//read and write by all the threads

   for (y = 1; y < y_size1 + 1; y++) {
     for (x = 1; x < x_size1 + 1; x++) {
       pixel_value_x = 0.0; //Output value for x coordinate convolution
       pixel_value_y = 0.0; //Output value for y coordinate convolution
             pixel_value = 0.0;   // Resultant output from the root mean
                                  // square of all the values

       // Convolution of a single pixel – below for loop reads pixel value
       // from all surrounding loops
       // for convolution

       for (j = -1; j <= 1; j++) {
         for (i = -1; i <= 1; i++) {

         // weigth_x denotes kernel matrix for x coordinate
         // weigth_y denotes kernel matrix for y coordinate

           pixel_value_x += weight_x[j + 1][i + 1] * image1[y + j][x + i];
           pixel_value_y += weight_y[j + 1][i + 1] * image1[y + j][x + i];
         }
       }
       pixel_value = sqrt((pixel_value_x * pixel_value_x) + (pixel_value_y *
pixel_value_y));

       //Since max and min are shared variable that can be written, the
thread need
       // to be “critical” (one thread at a time) synchronized
       #pragma omp critical
       {
           if (pixel_value < min) min = pixel_value;
           if (pixel_value > max) max = pixel_value; }  }  }
```

Sobel OMP Parallel: Four Level Nested for loop with necessary omp constructs applied.

To aid the Lab, a parallel program that makes use of only “omp parallel” construct was also built. Please check the appendix section for its source code. It makes use of thread id to avoid race condition among different thread. By doing that as an exercise the student will be able to easily identify the private and shared variable when parallelizing a code. Other implementation where Parallel Programming can go wrong was also created and added with this file, so that they can be used as a benchmark for lab exercise.

6. Asymmetric Multi Processing using MPI:

As mentioned in the Design Choices section, Message Passing Multicomputer model was chosen. So, the data will be split into sections with respect to the number of nodes available. Master node will take care of the process involved in splitting the data and sending them to different nodes, whereas the Slave nodes will process the data locally and sent them back to the master node.

When decided that the Image is going to be split, below are the design questions that arose,

1) How the two dimensional image is going to be split,
    a. It can split symmetrically along the y axis or x axis.
    b. It can be split symmetrically along both the axis.
    c. It can be split asymmetrically among the nodes.

Option 1.c was not a viable option as all the nodes are of same processing power it is necessary to split the data symmetrically among the nodes so that maximum performance can be achieved.

Option 1b is also not a viable option as data will be saved contiguously along x axis in memory, splitting the data along both the axis will result in non-contagious memory operations which is not desired and also difficult with the available MPI constructs.

So, option 1a was chosen and it was decided to split along the x axis as it has advantage of providing contagious memory chucks. This will help in great deal when splitting the array.

The Message with data can sent from Master to salve nodes and received back either using Point to Point Communication constructs (MPI_Send, MPI_Recv), or Collective Communication Constructs (MPI_Scatter, MPI_Gather, MPI_Scatterv and MPI_Gatherv).

Since, both are blocking communications, so care must be taken to avoid dead lock.

In our case, program used both collective and point to point communication in places wherever necessary.

To send the data from Master to slave nodes, Collective communication was used. Between MPI_Scatter and MPI_Scatterv, MPI_Scatterv is the correct option, because by using MPI_Scatter, one can only send contagious data. Overlapping of data cannot be achieved. Whereas with the help of MPI_Scatterv and MP_Gatherv we can achieve the overlapping of split data. The reason why overlapping is necessary because of the Edge case. When we are splitting the data symmetrically, Pixels in the edges of each chunk require the pixels from previous or next chunk for the purpose of convolution. So it is necessary to send the row immediately preceding the first row and the row immediately succeeding the last row of a chunk. This can be only achieved with the help of MPI_Scatterv and MPI_Gatherv.

Next is the challenge of finding the Upper and Lower bound in the Input data. It is necessary to calculate upper and lower bound for each of the chunk, so that it can be passed to MPI_Scatter function for splitting the data accordingly. But this can be avoided by using MPI Constructs, MPI_Type_subarry and MPI_Type_create_resized. These two constructs helps to create an MPI_Datatype for the size of chunk we will be sending and also to gather them back.

It should also be noted that, the dynamic array should be created on each of the nodes to save the data passed by the master nodes and it is necessary for all the nodes to have the knowledge of the size of the whole image, so that they can dynamically create the local array. This design choice was restricted to the assumption that the image is split symmetrically among the nodes.

So, before scattering the Image among the nodes, it is necessary to broadcast the dimension of the Image to all other slave nodes. This can be achieved using the MPI_Bcast construct.

Once Message passing design choices were made, it was applied to the sequential program to get an output image with each gathered chunk having different shades. This was due to the reason because of not sharing the maximum and minimum value which is computed in each node. Since, each nodes ended up using max and min values local to those nodes, the normalization output was different for each chunk resulting in output image of different shades for different chunks.

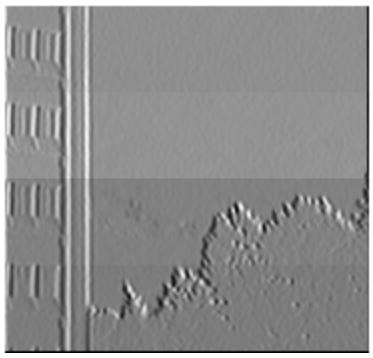

Figure 11: MPI parallel program output with different shades. The Image shows only the output of X coordinates, so that difference in shading will be easily visible.

This was resolved by making each slaves to send the max and min value computed by them to the Master node. The master node by receiving the max and min value, will compute the original max and min value and send them back to each nodes. Point to Point communication was used by all the slave nodes to send the max and min values.

After applying the solution, the output image was same as that of sequential image. The time taken by parallel program for same image used in section 2 was 0.05 seconds. It is slightly higher than the Symmetric Multi-Processing model, due to the overhead associated with passing data from one node to another. Please refer the appendix section for the MPI code.

7. Hybrid of SMP and ASMP using openMP and MPI:

As both SMP and ASMP programs were built successfully, a hybrid of both the models was built by adding openMP constructs to the existing ASMP program.

The Images which are divided symmetrically at the master node are sent to slave nodes for local processing. In each node being a Raspberry pi, which itself can act as a perfect SMP as described in section 5, openMP parallelization was applied locally.

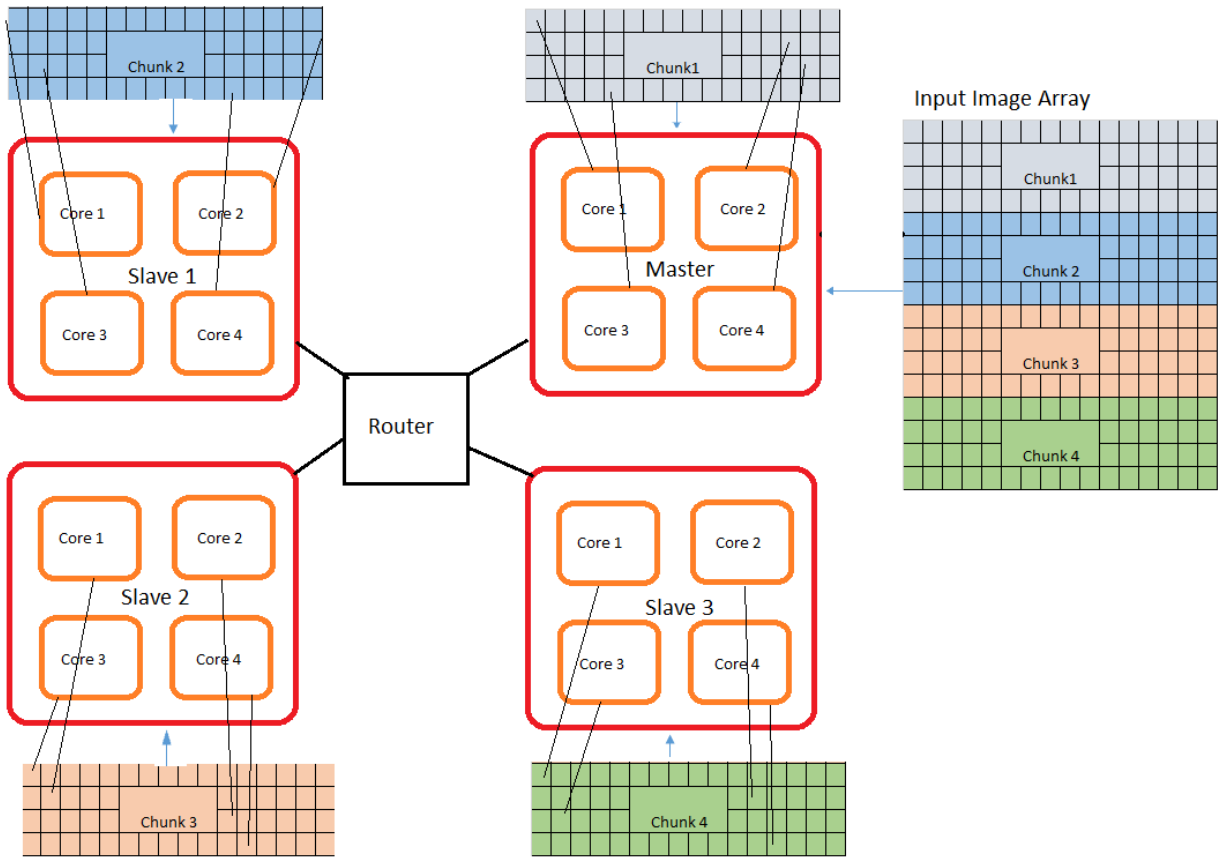

Figure 12: Showing Master node of ASMP cluster dividing an input image array into symmetric chunks and sharing it with all the node. Each node acting as a SMP applies openMP parallelization resulting in hybrid parallelization.

The reason why the SMP and ASMP setup is able to achieve high speedup is because, 16 Cores are processing different parts of the image concurrently, resulting in high speed up. Also, The Intercommunication Latency which is low due to 10/100 Mbps Ethernet connection between the Nodes helped to reduce the overhead associated with the Message Passing.

Since all the nodes involved in the Cluster uses same Processor and of same architecture, there was no difficulty or issues when implementing it. The time taken for the same image used for discussion in all above sections was 0.02 seconds, which is clearly a better performance than both SMP and ASMP programs.

8. Performance Analysis from different Programming Models:

From SMP, ASMP and Hybrid applications the performance time was noted for four different times and an average time was calculated to compare the performance of different implementation. Images of different size were selected to prove the point that large data can be processed efficiently by programming parallel instead of sequentially. This helps to exploit the availability of multiple cores in the system and also provides an opportunity to getting

performance in distributed system in case of Internet of Things. See the Table 2, for the comparison of time taken by each implementation for images of different size.

Once the satisfying performance is obtained and verified, the performance of the hybrid program was compared to the sequential program running in highly sophisticated machine with good cache configuration. The machine chosen was amdpool of Cornell's Computer Systems Laboratory. Amdpool is composed of pool of 8 [Dual CPU] AMD Opteron 8218 processors. Each of these processor is having 2.6 GHz clock frequency and cache of size 1024 KB. Each dual core processor costs around $1500. When the sequential program of sobel filter was ran in the amdpool for the Image of size 4096 × 4096 pixels, the time take was 7 seconds. Whereas Hybrid SMP and ASMP system built for this project achieved a run time of 5 seconds for the same Image. A raspberry Pi cluster with 4 Pis, each costing around $35 and with a total cost of around $150, was able to achieve better performance than the high speed processor costing around $1500. This proves the Cost wise efficiency of parallel programming when compared to the sequential programming.

Image Pixels	Sobel Sequential (time in Seconds)	Sobel OMP (time in Seconds)	Sobel MPI (time in Seconds)	Sobel Hybrid (time in Seconds)
256 × 256	0.2124	0.0851	0.0900	0.0500
	0.2271	0.0709	0.0900	0.0400
	0.2132	0.0772	0.0800	0.0500
	0.2170	0.0716	0.0700	0.0500
256 × 256 (Average)	**0.2174**	**0.0762**	**0.0825**	**0.0475**
512 × 512	0.8433	0.2169	0.2500	0.1000
	0.8223	0.2166	0.2600	0.1100
	0.8321	0.2318	0.2600	0.1000
	0.8343	0.2139	0.2600	0.1000
512 × 512 (Average)	**0.8330**	**0.2198**	**0.2575**	**0.1025**
1024 × 1024	3.2781	0.8463	1.0000	0.3800
	3.3100	0.8471	1.0000	0.3700
	3.2857	0.8451	1.0000	0.3700
	3.2786	0.8434	1.0000	0.3700
1024 × 1024 (Average)	**3.2881**	**0.8455**	**1.0000**	**0.3725**
2048 × 2048	13.1241	3.3563	3.9600	1.4500
	13.1234	3.3590	3.9700	1.4400
	13.1288	3.3586	3.9600	1.4400
	13.0123	3.3502	3.9600	1.4300
2048 × 2048 (Average)	**13.0972**	**3.3560**	**3.9625**	**1.4400**
4096 × 4096	52.3500	13.4159	15.8300	5.6800
	52.3288	13.4240	15.8500	5.6800
	52.3390	13.4246	15.8500	5.6800
	52.3298	13.4237	15.8000	5.6700
4096 × 4096 (Average)	**52.3369**	**13.4221**	**15.8325**	**5.6775**

Table 1: Time taken by all parallel programs designed in this project. Each Test Image was ran four times to come up with an average time.

Image Pixels	Sobel Sequential (time in Seconds)	Sobel OMP (time in Seconds)	Sobel MPI (time in Seconds)	Sobel Hybrid (time in Seconds)
256 × 256	**0.2174**	**0.0762**	**0.0825**	**0.0475**
512 × 512	**0.8330**	**0.2198**	**0.2575**	**0.1025**
1024 × 1024	**3.2881**	**0.8455**	**1.0000**	**0.3725**
2048 × 2048	**13.0972**	**3.3560**	**3.9625**	**1.4400**
4096 × 4096	**52.3369**	**13.4221**	**15.8325**	**5.6775**

Table 2 : Comparison of time taken by all parallel sobel programs with sequential program.

Image Pixels	Sobel Sequential (time in Seconds)	Sobel OMP (time in Seconds)	Sobel MPI (time in Seconds)	Sobel Hybrid (time in Seconds)
256 × 256	1	2.8	2.6	4.6
512 × 512	1	3.8	3.2	8.1
1024 × 1024	1	3.9	3.3	8.9
2048 × 2048	1	3.9	3.3	9.1
4096 × 4096	1	3.9	3.3	9.2

Table 3 : Comparison of speed up by parallel sobel programs with reference to sequential program.

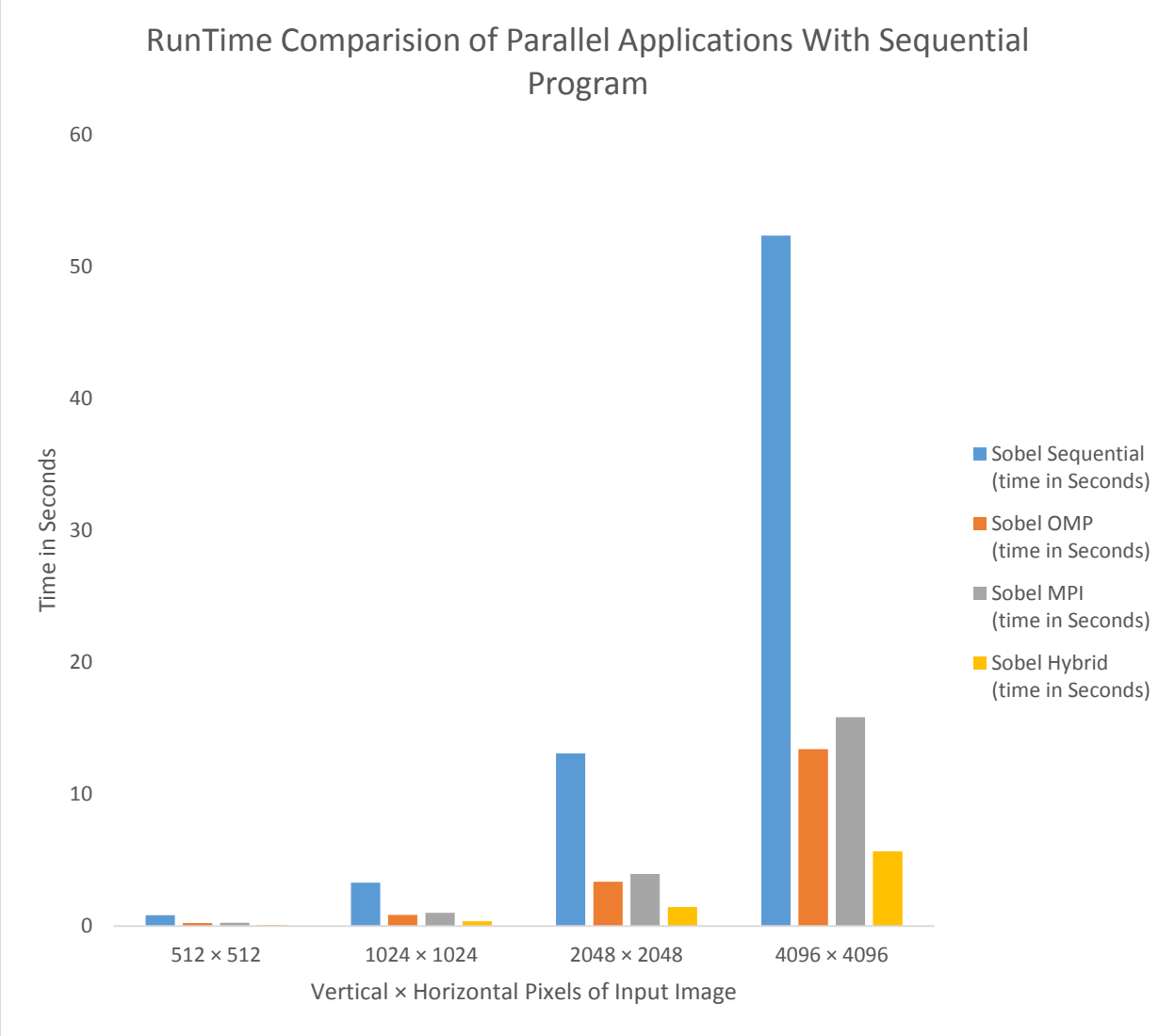

Chart 1: Graph comparing the Parallel and Sequential program performance based on Pixel size versus time taken.

9. Future Work:

The Project has great potential to test and research different and new methods in the field of parallel and distributed computing. Following are some of the options that can be considered,

- FPGA's can be connected to each raspberry pi node as accelerators. Tradeoffs between General purpose parallel programming and FPGA accelerators can be analyzed.
- Exploring different input data. For example, Voice commands, Sensor Data.
- Implementing a Fog Network, which is an architecture that uses one or a collaborative multitude of end-user clients or near-user edge devices to carry out a substantial amount of storage (rather than stored primarily in cloud data centers), communication (rather than routed over the internet backbone), and control, configuration, measurement and management (rather than controlled primarily by network gateways such as those in the LTE core network).
- A real time IOT application can be chosen and parallelized to analyze the performance.

Acknowledge:

I appreciate Cornell ECE department for providing the Raspberry Pis and Other required components. The course ECE 5990 – Embedded Operating System gave me an enough exposure and confident to work in Raspberry Pi and build a parallel cluster based on that.

I heartily appreciate the help from my MEng project advisor – Prof. Joe Skovira. With every design problems and issues I faced or with every wrong design choices I made, he provided a great guidance to resolve the issues. It was his enthusiasm to hear new ideas and his grace to spend time reasoning the design conflicts helped me emulate his methods of problem solving to complete this project successfully.

References:

1. MPI Installation:

https://www.mpich.org/static/downloads/3.2/mpich-3.2-installguide.pdf

2. MPI Constructs Reference:

http://www.mpich.org/static/docs/v3.2/www3/

3. OpenMP Reference:

https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG

http://openmp.org/mp-documents/omp-hands-on-SC08.pdf

4. Sobel Filtering:

https://en.wikipedia.org/wiki/Sobel_operator

5. The sequential program of Sobel Filter was based on below reference,

http://cis.k.hosei.ac.jp/~wakahara/sobel.c

6. openMP : https://en.wikipedia.org/wiki/OpenMP
7. MPI : https://en.wikipedia.org/wiki/Message_Passing_Interface
8. Raspberry PI : https://www.raspberrypi.org/
9. SSH key genrartion steps : https://www.jamescoyle.net/how-to/1215-create-ssh-key-authentication-between-nodes
10. Fog Computing : https://en.wikipedia.org/wiki/Fog_computing
11. Sobel Filter Image : https://blog.saush.com/2011/04/20/edge-detction-with-sobel-operator-in-ruby/

Appendix

Installation Steps

One of the raspberry pi was selected as a Master Node. Once booted with a Formatted SD card, NOOBS will prompt to install Different Operating System. Raspbian Linux which is a standard OS for Raspberry Pi was selected and installed.

`sudo apt-get update` was ran to update the linux to stable version.

Once update was complete, gcc availability was checked using “gcc --version” command and it was available as it comes along with raspbian linux.

GCC compiler is built inherently in Raspbian Linux. openMP api is also available as part of GCC compiler. So, just by installing raspbian linux, Raspberry Pi is turned into a Platform for SMP, where parallel programs can be built immediately after successful OS installation.

The parallel Programs should include, “omp.h” header for c programs.
gcc command should have “-fopenmp” to compile the openMP parallel programs. An example command is as follows,

```
gcc -fopenmp sobel.c -o sobel -g -Wall
```

-o : Output File
-g : produce debug information in native format
-Wall – Show All Warinings

A separate directory was created for mpich installation,

```
mkdir ~/mpich_stable
```

In newly created “mpich” directory, Ran the following command to download the tar file of stable version of mpich,

```
wget  http://www.mpich.org/static/downloads/3.2/mpich-3.2.tar.gz
```

Unpack the tar ball file by running the following command,

```
tar xfz mpich.tar.gz
```

Run the following command to create a program directory in “home/”

```
sudo mkdir /home/rpimpi/mpich-install
```

Create a build directory by running the following command,

```
sudo mkdir /home/pi/mpich-build
```

cd into the newly created “mpich-build” directory, Run the following command to configure the mpich stable version in the Installation directory created earlier.

```
“/home/pi/mpich_stable/mpich-3.2/configure -prefix=/home/rpimpi/mpich-install |& tee c.txt”
```

Once files are build, run the following make commands,

```
sudo make |& tee m.txt
```

```
sudo make VERBOSE=1 |& tee m.txt
```

```
sudo make install |& tee mi.txt
```

The path of MPICH can be added to “.profile” by running the following command. This allows us to run the MPICH programs without specifying their installation directory.

```
sudo vim /home/pi/.profile
```

Add the following lines to the end of the file,

```
PATH = “$PATH: /home/rpimpi/mpich-install/bin”
```

Now running “which mpicc” should show the path of mpicc application.

A file with all the ip address of participating nodes need to be created, For example, for things project a file with name “machinefile” was created and added with ip address of all other participating nodes,

To check whether the mpi installation was success, below command should return hostname of the node.

```
mpiexec –f machinefile –n 1 hostname
```
Where, -f : implementation-defined specification file
-n : Number of Process to use.

mpicc is the command to compile the MPI programs. An example mpicc command is as follows,

```
mpicc sobel_mpi.c -o sobel_mpi -g -Wall
```
-o : Output File

-g : produce debug information in native format
-Wall – Show All Warinings

The Programs should include “mpi.h” as a header.

With this all required configuration is finished master node. An Image of the SD card of Master Node was created using SD card Image writer of our choice and the Master Node image was written to the SD card of all other nodes. This helps to save the time of installing all the required applications in all the nodes separately.

Four raspberry Pi 2 Model B computers were connected using Ethernet Router. Category 5 E Lan cables were used to connect the nodes to the router.
Once connected to the router the Nodes gets assigned with the IP addresses by the router.

The hostname was changed to node1, node2, node3 and node4 by replacing the existing hostname in “/etc/hostname” file. This helps to identify the nodes with their hostname.
Slave nodes can be accessed from the master nodes by using “ssh”.
Once done Master Login was setup using the following commands,
SSH key was generated at the master node using the following command,

```
ssh-keygen –t rsa –C “pi@node1”
```

Where –C is for comment and we can add our own comments
Prompts for passphrase will be asked, it is upto the person to either add passphrase or not. If not, one can continue by simply pressing enter and rsa key will be generated.

Once ssh key was generated, the public key of master node can be added to auhroized key file of slave node by running the flowing command,

```
cat ~/.ssh/id_rsa.pub | ssh user@<ipaddressofslave> “mkdir .ssh;cat >>
.ssh/authorized_keys”
```

The above step can repeated for all the nodes by replacing the address and user of the slave nodes.

Once done, ssh each of the slave node and create the ssh key using the same command as in master node. Once ssh key was generated, the public key of each node needs to be added to the authorized key of Master node using the same command as in master node.

```
cat ~/.ssh/id_rsa.pub | ssh user@<ipaddressofmaster> “cat >>
.ssh/authorized_keys”
```

Output Images used in Analysis:

Image256.pgm (256×256 pixels)

Image256_par.pgm

Image1024.pgm (256×256 pixels)

Image1024_par.pgm

Image2048.pgm (2048×2048 pixels)

Image2048_par.pgm

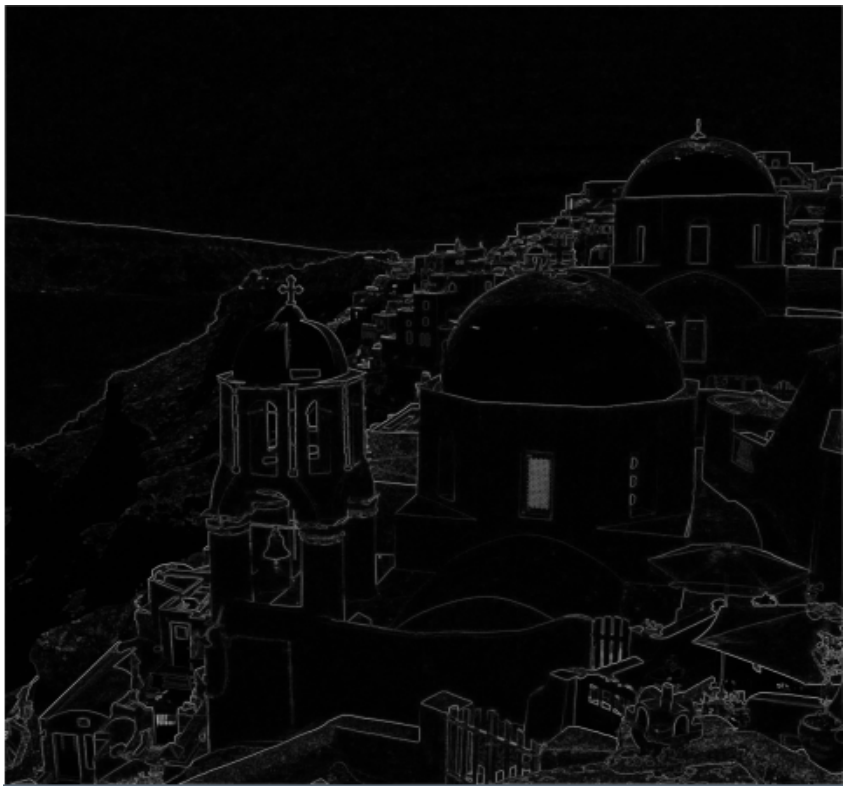

Image4096.pgm (4096×4096 pixels)

Image4096_par.pgm

Source code of the Implementations:

Sobel Sequential Program:

```
/* sobel.c  - Sobel filter sequential program*/
 #include <stdio.h>
 #include <stdlib.h>
 #include <float.h>
 #include "mypgm.h"
 #include <omp.h>
 #include <math.h>


 void sobel_filtering()
      /* Spatial filtering of image data */
      /* Sobel filter (horizontal differentiation */
      /* Input: image1[y][x] ---- Outout: image2[y][x] */
 {

   /* Definition of Sobel filter in horizontal direction */
   int weight_x[3][3] =   {{ -1,   0,   1 },
                           { -2,   0,   2 },
                           { -1,   0,   1 }};

   int weight_y[3][3] =   {{ -1,  -2,  -1 },
                           {  0,   0,   0 },
                           {  1,   2,   1 }};

   double pixel_value_x;
   double pixel_value_y;
   double pixel_value;
   double min, max;
   int x, y, i, j;  /* Loop variable */
   /* Maximum values calculation after filtering*/
   printf("Now, filtering of input image is performed\n\n");
   min = DBL_MAX;
   max = -DBL_MAX;
   for (y = 1; y < y_size1 + 1; y++) {
     for (x = 1; x < x_size1 + 1; x++) {
       pixel_value_x = 0.0;
       pixel_value_y = 0.0;
       pixel_value = 0.0;
       for (j = -1; j <= 1; j++) {
         for (i = -1; i <= 1; i++) {
           pixel_value_x += weight_x[j + 1][i + 1] * image1[y + j][x + i];
           pixel_value_y += weight_y[j + 1][i + 1] * image1[y + j][x + i];
         }
       }
       pixel_value = sqrt((pixel_value_x * pixel_value_x) + (pixel_value_y *
pixel_value_y));

       if (pixel_value < min) min = pixel_value;
       if (pixel_value > max) max = pixel_value;
     }
   }

   printf("the value of minimum  %f\n",min);
```

```
   printf("the value of maximum  %f\n",max);

   if ((int)(max - min) == 0) {
    printf("Nothing exists!!!\n\n");
     exit(1);
   }

   /* Initialization of image2[y][x] */
   x_size2 = x_size1;
   y_size2 = y_size1;
   for (y = 0; y < y_size2; y++) {
     for (x = 0; x < x_size2; x++) {
       image2[y][x] = 0;
     }
   }

   /* Generation of image2 after linear transformtion */
   for (y = 1; y < y_size1 + 1; y++) {
     for (x = 1; x < x_size1 + 1; x++) {
       pixel_value_x = 0.0;
       pixel_value_y = 0.0;
       pixel_value = 0.0;
       for (j = -1; j <= 1; j++) {
         for (i = -1; i <= 1; i++) {
           pixel_value_x += weight_x[j + 1][i + 1] * image1[y + j][x + i];
           pixel_value_y += weight_y[j + 1][i + 1] * image1[y + j][x + i];
         }
       }

       pixel_value = sqrt((pixel_value_x * pixel_value_x) + (pixel_value_y *
pixel_value_y));

       pixel_value = MAX_BRIGHTNESS * (pixel_value - min) / (max - min);

       image2[y][x] = (unsigned char)pixel_value;
     }
   }
 }

 main( )
 {
   double start_time, time;


   load_image_data( );   /* Input of image1 */
   start_time = omp_get_wtime();
   sobel_filtering();   /* Sobel filter is applied to image1 */
   printf("The total time for execution is : %f", omp_get_wtime() -
start_time );
   save_image_data( );   /* Output of image2 */


   return 0;
 }
```

Sobel Parallel Program using openMP:

```
/* sobel.c – Sobel Filtering with openMP
“Omp Parallel For” and “OMP critical” constructs are used*/
 #include <stdio.h>
 #include <stdlib.h>
 #include <float.h>
 #include "mypgm.h"
 #include <math.h>
 #include <omp.h>

 #define num_threads 4


 void sobel_filtering( )
      /* Spatial filtering of image data */
      /* Sobel filter (horizontal differentiation */
      /* Input: image1[y][x] ---- Outout: image2[y][x] */
 {
   /* Definition of Sobel filter in horizontal direction */
   int weight_x[3][3] =   {{ -1,   0,   1 },
                           { -2,   0,   2 },
                           { -1,   0,   1 }};

   /* Definition of Sobel filter in vertical direction */
   int weight_y[3][3] =   {{ -1,  -2,  -1 },
                           {  0,   0,   0 },
                           {  1,   2,   1 }};

   omp_set_num_threads(num_threads);


   double pixel_value_x;
   double pixel_value_y;
   double pixel_value;

   double min, max;
   int x, y, i, j;  /* Loop variable */

   /* Maximum values calculation after filtering*/
   printf("Now, filtering of input image is performed\n\n");
   min = DBL_MAX;
   max = -DBL_MAX;

   printf("The ysize and xsize is %d and %d", y_size1,x_size1 );
   pixel_value = 0.0;

   #pragma omp parallel for collapse(2)
private(i,j,pixel_value,pixel_value_x,pixel_value_y)\
                                        shared(min,max)

   for (y = 1; y < y_size1 + 1; y++) {
     for (x = 1; x < x_size1 + 1; x++) {
       pixel_value_x = 0.0;
       pixel_value_y = 0.0;
```

```
        pixel_value = 0.0;
        for (j = -1; j <= 1; j++) {
          for (i = -1; i <= 1; i++) {
            pixel_value_x += weight_x[j + 1][i + 1] * image1[y + j][x + i];
            pixel_value_y += weight_y[j + 1][i + 1] * image1[y + j][x + i];
          }
        }
        pixel_value = sqrt((pixel_value_x * pixel_value_x) + (pixel_value_y *
pixel_value_y));

        #pragma omp critical
        {
        if (pixel_value < min) min = pixel_value;
        if (pixel_value > max) max = pixel_value;
        }
      }
    }


    printf("The Value of Minimum  %f\n",min);
    printf("The Value of Maximum  %f\n",max);

    if ((int)(max - min) == 0) {
      printf("Nothing exists!!!\n\n");
      exit(1);
    }

    /* Initialization of image2[y][x] */
    x_size2 = x_size1;
    y_size2 = y_size1;

    #pragma omp parallel for collapse(2) private(x,y)\
                                         shared(image2)
    for (y = 0; y < y_size2; y++) {
      for (x = 0; x < x_size2; x++) {
        image2[y][x] = 0;
      }
    }


    /* Generation of image2 after linear transformtion */
    #pragma omp parallel for collapse(2)
private(i,j,pixel_value,pixel_value_x,pixel_value_y)\
                                         shared(image2)
    for (y = 1; y < y_size1 + 1; y++) {
      for (x = 1; x < x_size1 + 1; x++) {
        pixel_value_x = 0.0;
        pixel_value_y = 0.0;
        pixel_value = 0.0;
        for (j = -1; j <= 1; j++) {
          for (i = -1; i <= 1; i++) {
            pixel_value_x += weight_x[j + 1][i + 1] * image1[y + j][x + i];
            pixel_value_y += weight_y[j + 1][i + 1] * image1[y + j][x + i];
          }
        }

        pixel_value = sqrt((pixel_value_x * pixel_value_x) + (pixel_value_y *
pixel_value_y));
```

```
      pixel_value = MAX_BRIGHTNESS * (pixel_value - min) / (max - min);

      image2[y - 1][x - 1] = (unsigned char)pixel_value;
     }
  }
}

main( )
{
  load_image_data( );   /* Input of image1 */
  double start_time = omp_get_wtime();
  sobel_filtering( );   /* Sobel filter is applied to image1 */
  printf("The total time for filering is %f", omp_get_wtime() - start_time);
  save_image_data( );   /* Output of image2 */
  return 0;
}
```

Sobel Parallel Program using MPI:

```
/* sobel.c  - Chunks split by the Master node are saved locally
And Sobel filter is applied locally and final output is gathered at Master
node*/
 #include <stdio.h>
 #include <stdlib.h>
 #include <float.h>
 #include "mypgm.h"
 #include "mpi.h"
 #include <math.h>

 #define MASTER 0


 unsigned char** imageLocal;
 unsigned char** imageOpLocal;
 unsigned char** imageOpGlobal;
 int global_x_size, global_y_size;
 int size;



 void sobel_filtering(int rank, int size)
      /* Spatial filtering of image data */
      /* Sobel filter (horizontal differentiation */
      /* Input: image1[y][x] ---- Outout: image2[y][x] */
{
  /* Definition of Sobel filter in horizontal direction */
  int weight_x[3][3] =   {{ -1,   0,   1 },
                          { -2,   0,   2 },
                          { -1,   0,   1 }};


  /* Definition of Sobel filter in horizontal direction */
  int weight_y[3][3] =   {{ -1,  -2,  -1 },
                          {  0,   0,   0 },
                          {  1,   2,   1 }};


  double pixel_value_x;
```

```
   double pixel_value_y;
   double pixel_value;
   double min, max;
   int x, y, i, j,namelen;  /* Loop variable */
   char name[26];

   /* Maximum values calculation after filtering*/
   MPI_Get_processor_name(name, &namelen);
 //  printf("Now, filtering of input image is performed in Node %d
(%s)\n\n",rank,name);
   min = DBL_MAX;
   max = -DBL_MAX;

   for (y = 1; y < (((global_y_size-2)/size)+1); y++) {
     for (x = 1; x < (global_x_size-1); x++) {
       pixel_value_x = 0.0;
       pixel_value_y = 0.0;
       pixel_value = 0.0;
       for (j = -1; j <= 1; j++) {
        for (i = -1; i <= 1; i++) {
           pixel_value_x += weight_x[j + 1][i + 1] * imageLocal[y + j][x +
i];
           pixel_value_y += weight_y[j + 1][i + 1] * imageLocal[y + j][x +
i];
         }
       }
         pixel_value = sqrt((pixel_value_x * pixel_value_x) + (pixel_value_y
* pixel_value_y));

       if (pixel_value < min) min = pixel_value;
       if (pixel_value > max) max = pixel_value;
     }
   }

   MPI_Status status;

   if(rank != MASTER)
   {

     //Send Max and Min to Master
     MPI_Send(&min, 1, MPI_DOUBLE, MASTER, rank+100,MPI_COMM_WORLD);
     MPI_Send(&max, 1, MPI_DOUBLE, MASTER, rank+110,MPI_COMM_WORLD);

     //printf("Sent the max and min\n");
     //Recieve Max and Min from Nodes
     MPI_Recv(&min, 1, MPI_DOUBLE, MASTER, rank+200,MPI_COMM_WORLD, &status);
     MPI_Recv(&max, 1, MPI_DOUBLE, MASTER, rank+210,MPI_COMM_WORLD, &status);

   }



   if(rank == MASTER)
   {
     double minTemp,maxTemp;

     for(i = 1; i < size; i++)
     {
         //printf("Received the max and min\n");
```

```
            //Recieve Max and Min from Nodes
            MPI_Recv(&minTemp, 1, MPI_DOUBLE, i, i+100,MPI_COMM_WORLD, &status);
            MPI_Recv(&maxTemp, 1, MPI_DOUBLE, i, i+110,MPI_COMM_WORLD, &status);

            if (minTemp < min) min = minTemp;
            if (maxTemp > max) max = maxTemp;
        }

        for(i = 1; i < size; i++)
        {
            //Send Max and Min to Nodes
            MPI_Send(&min, 1, MPI_DOUBLE, i, i+200,MPI_COMM_WORLD);
            MPI_Send(&max, 1, MPI_DOUBLE, i, i+210,MPI_COMM_WORLD);
        }
    }
if ((int)(max - min) == 0) {
      printf("Nothing exists!!!\n\n");
      exit(1);
    }

    for (y = 0; y < ((global_y_size - 2)/size); y++) {
      for (x = 0; x < (global_x_size-2); x++) {
        imageOpLocal[y][x] = 0;
      }
    }

    /* Generation of image2 after linear transformtion */
    for (y = 1; y < (((global_y_size-2)/size) + 1); y++) {
      for (x = 1; x < (global_x_size-1); x++) {
        pixel_value_x = 0.0;
        pixel_value_y = 0.0;
        pixel_value = 0.0;
        for (j = -1; j <= 1; j++) {
          for (i = -1; i <= 1; i++) {
            pixel_value_x += weight_x[j + 1][i + 1] * imageLocal[y + j][x +
i];
            pixel_value_y += weight_y[j + 1][i + 1] * imageLocal[y + j][x +
i];
          }
        }

        pixel_value = sqrt((pixel_value_x * pixel_value_x) + (pixel_value_y *
pixel_value_y));

        pixel_value = MAX_BRIGHTNESS * (pixel_value - min) / (max - min);

        imageOpLocal[y-1][x-1] = (unsigned char)pixel_value;
      }
    }

 }

 int main(int argc, char** argv)
 {
    int i;
    int rank;
    unsigned char* ipImageptr = NULL;
    unsigned char* opImageptr = NULL;
    int temp;
```

```
   double timeStart, timeEnd;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   ipImageptr = NULL;

   if(rank == MASTER)
   {

     load_image_data();
    global_x_size = x_size1+2;
     global_y_size = y_size1+2;

     ipImageptr = &(image1[0][0]);

     timeStart = MPI_Wtime();

   }


   //Broadcast the image size to all the nodes

   MPI_Bcast(&global_x_size, 1, MPI_INT, MASTER, MPI_COMM_WORLD);
   MPI_Bcast(&global_y_size, 1, MPI_INT, MASTER, MPI_COMM_WORLD);

   //MPI_Barrier for Synchronisation
   MPI_Barrier(MPI_COMM_WORLD);

   printf("The value of split %d",((global_y_size-2)/size)+2);
   int ipImageDim[2] = {global_y_size, global_x_size};
   int scatterImageDim[2] = {((global_y_size-2)/size)+2, global_x_size};
   int scatterStart[2]   = {0,0};
   int sendcount[size];
   int displ[size];

   int opImageDim[2] = {(global_y_size), (global_x_size)};
   int gatherImageDim[2] = {(global_y_size-2)/size, (global_x_size-2)};
   int gatherStart[2]   = {0,0};
   int recvcount[size];
   int recvdispl[size];


   //Scatter dimensions for input image from Master process

   MPI_Datatype scatter_type, scatter_subarraytype;
   MPI_Type_create_subarray(2, ipImageDim, scatterImageDim, scatterStart,
MPI_ORDER_C, MPI_UNSIGNED_CHAR, &scatter_type);
   MPI_Type_create_resized(scatter_type, 0, (global_x_size)*sizeof(unsigned
char), &scatter_subarraytype);
   MPI_Type_commit(&scatter_subarraytype);
   printf("The Golbal X Size is %d\n",global_x_size);
   //Gather dimesions for output image from all the porcess

   MPI_Datatype gather_type, gather_subarraytype;
   MPI_Type_create_subarray(2, opImageDim, gatherImageDim, gatherStart,
MPI_ORDER_C, MPI_UNSIGNED_CHAR, &gather_type);
```

```
   MPI_Type_create_resized(gather_type, 0, (global_x_size)*sizeof(unsigned
char), &gather_subarraytype);
   MPI_Type_commit(&gather_subarraytype);

   malloc2dcharm(&imageLocal, ((global_y_size-2)/size)+2, (global_x_size));
   malloc2dcharm(&imageOpLocal, ((global_y_size-2)/size), (global_x_size-2));

   if(rank == MASTER)
   {
     temp = 0;
     for(i = 0; i < size; i++) {
         sendcount[i] = 1;
         recvcount[i] = 1;
         displ[i] = temp;
         recvdispl[i] = temp;
         temp += ((global_y_size-2)/size);
     }
     opImageptr = &(image2[0][0]);
 }

   //  printf("y size %d x size %d\n\n",global_y_size,global_x_size);

   MPI_Scatterv(ipImageptr, sendcount, displ, scatter_subarraytype,
&(imageLocal[0][0]), (((global_y_size-2)/size)+2)*global_x_size,
MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);


   sobel_filtering(rank, size);


   MPI_Gatherv(&(imageOpLocal[0][0]), ((global_y_size-
2)/size)*(global_x_size-2), MPI_UNSIGNED_CHAR, opImageptr, recvcount,
recvdispl, gather_subarraytype, 0, MPI_COMM_WORLD);

   if(rank == MASTER){
     timeEnd = MPI_Wtime();
   }

   free2dchar(&imageLocal);
   free2dchar(&imageOpLocal);

   MPI_Type_free(&scatter_subarraytype);
   MPI_Type_free(&gather_subarraytype);

   if(rank == MASTER){

     printf("The time taken %1.2f\n",(timeEnd - timeStart));
     x_size2 = global_x_size-2;
     y_size2 = global_y_size-2;

     save_image_data();   /* Output of image2 */

     free2dchar(&image2);
   }

   MPI_Finalize();
   return 0;
 }
```

Sobel Program Implemented using Hybrid of openMP and MPI:

```
/* sobel.c */
 #include <stdio.h>
 #include <stdlib.h>
 #include <float.h>
 #include "mypgm.h"
 #include "mpi.h"
 #include <math.h>
 #include <omp.h>

 #define MASTER 0


 unsigned char** imageLocal;
 unsigned char** imageOpLocal;
 unsigned char** imageOpGlobal;
 int global_x_size, global_y_size;
 int size;



 void sobel_filtering(int rank, int size)
      /* Spatial filtering of image data */
      /* Sobel filter (horizontal differentiation */
    /* Input: image1[y][x] ---- Outout: image2[y][x] */
 {
   /* Definition of Sobel filter in horizontal direction */
   int weight_x[3][3] =   {{ -1,   0,   1 },
                           { -2,   0,   2 },
                           { -1,   0,   1 }};


   /* Definition of Sobel filter in horizontal direction */
   int weight_y[3][3] =   {{ -1,  -2,  -1 },
                           {  0,   0,   0 },
                           {  1,   2,   1 }};


   double pixel_value_x;
   double pixel_value_y;
   double pixel_value;
   double min, max;
   int x, y, i, j,namelen;  /* Loop variable */
   char name[26];

   /* Maximum values calculation after filtering*/
   MPI_Get_processor_name(name, &namelen);
  min = DBL_MAX;
   max = -DBL_MAX;

   #pragma omp parallel for collapse(2)
private(x,y,i,j,pixel_value,pixel_value_x,pixel_value_y)\
                                        shared(min,max)
   for (y = 1; y < (((global_y_size-2)/size)+1); y++) {
```

```
    for (x = 1; x < (global_x_size-1); x++) {
      pixel_value_x = 0.0;
      pixel_value_y = 0.0;
      pixel_value = 0.0;
      for (j = -1; j <= 1; j++) {
        for (i = -1; i <= 1; i++) {
          pixel_value_x += weight_x[j + 1][i + 1] * imageLocal[y + j][x +
i];
          pixel_value_y += weight_y[j + 1][i + 1] * imageLocal[y + j][x +
i];
        }
      }
      pixel_value = sqrt((pixel_value_x * pixel_value_x) + (pixel_value_y *
pixel_value_y));

      if (pixel_value < min) min = pixel_value;
      if (pixel_value > max) max = pixel_value;
    }
  }
 MPI_Status status;

  if(rank != MASTER)
  {

    //Send Max and Min to Master
    MPI_Send(&min, 1, MPI_DOUBLE, MASTER, rank+100,MPI_COMM_WORLD);
    MPI_Send(&max, 1, MPI_DOUBLE, MASTER, rank+110,MPI_COMM_WORLD);

    //printf("Sent the max and min\n");
    //Recieve Max and Min from Nodes
    MPI_Recv(&min, 1, MPI_DOUBLE, MASTER, rank+200,MPI_COMM_WORLD, &status);
    MPI_Recv(&max, 1, MPI_DOUBLE, MASTER, rank+210,MPI_COMM_WORLD, &status);

  }



  if(rank == MASTER)
  {
    double minTemp,maxTemp;
   for(i = 1; i < size; i++)
    {
        //printf("Received the max and min\n");
        //Recieve Max and Min from Nodes
        MPI_Recv(&minTemp, 1, MPI_DOUBLE, i, i+100,MPI_COMM_WORLD, &status);
        MPI_Recv(&maxTemp, 1, MPI_DOUBLE, i, i+110,MPI_COMM_WORLD, &status);

        if (minTemp < min) min = minTemp;
        if (maxTemp > max) max = maxTemp;
    }

    for(i = 1; i < size; i++)
    {
        //Send Max and Min to Nodes
        MPI_Send(&min, 1, MPI_DOUBLE, i, i+200,MPI_COMM_WORLD);
        MPI_Send(&max, 1, MPI_DOUBLE, i, i+210,MPI_COMM_WORLD);
    }
  }
```

```
  if ((int)(max - min) == 0) {
     printf("Nothing exists!!!\n\n");
     exit(1);
   }

   #pragma omp parallel for collapse(2) private(x,y)\
                                        shared(image2)
   for (y = 0; y < ((global_y_size - 2)/size); y++) {
     for (x = 0; x < (global_x_size-2); x++) {
       imageOpLocal[y][x] = 0;
     }
   }

   /* Generation of image2 after linear transformtion */
   #pragma omp parallel for collapse(2)
private(x,y,i,j,pixel_value,pixel_value_x,pixel_value_y)\
                                        shared(image2)
   for (y = 1; y < (((global_y_size-2)/size) + 1); y++) {
     for (x = 1; x < (global_x_size-1); x++) {
       pixel_value_x = 0.0;
       pixel_value_y = 0.0;
       pixel_value = 0.0;
       for (j = -1; j <= 1; j++) {
         for (i = -1; i <= 1; i++) {
       pixel_value_x += weight_x[j + 1][i + 1] * imageLocal[y + j][x + i];
           pixel_value_y += weight_y[j + 1][i + 1] * imageLocal[y + j][x +
i];
         }
       }

       pixel_value = sqrt((pixel_value_x * pixel_value_x) + (pixel_value_y *
pixel_value_y));

       pixel_value = MAX_BRIGHTNESS * (pixel_value - min) / (max - min);

       imageOpLocal[y-1][x-1] = (unsigned char)pixel_value;
     }
   }

 }

 int main(int argc, char** argv)
 {
   int i;
   int rank;
   unsigned char* ipImageptr = NULL;
   unsigned char* opImageptr = NULL;
   int temp;
   double timeStart, timeEnd;
  MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   ipImageptr = NULL;

   if(rank == MASTER)
   {

     load_image_data();
```

```
      global_x_size = x_size1+2;
      global_y_size = y_size1+2;

      ipImageptr = &(image1[0][0]);

      //timeStart = MPI_Wtime();
      timeStart = omp_get_wtime();

    }


    //Broadcast the image size to all the nodes
MPI_Bcast(&global_x_size, 1, MPI_INT, MASTER, MPI_COMM_WORLD);
    MPI_Bcast(&global_y_size, 1, MPI_INT, MASTER, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);


    int ipImageDim[2] = {global_y_size, global_x_size};
    int scatterImageDim[2] = {((global_y_size-2)/size)+2, global_x_size};
    int scatterStart[2]   = {0,0};
    int sendcount[size];
    int displ[size];

    int opImageDim[2] = {(global_y_size), (global_x_size)};
    int gatherImageDim[2] = {(global_y_size-2)/size, (global_x_size-2)};
    int gatherStart[2]   = {0,0};
    int recvcount[size];
    int recvdispl[size];


    //Scatter dimensions for input image from Master process

    MPI_Datatype scatter_type, scatter_subarraytype;
    MPI_Type_create_subarray(2, ipImageDim, scatterImageDim, scatterStart,
MPI_ORDER_C, MPI_UNSIGNED_CHAR, &scatter_type);
  MPI_Type_create_resized(gather_type, 0, (global_x_size)*sizeof(unsigned
char), &gather_subarraytype);
    MPI_Type_commit(&gather_subarraytype);


    malloc2dcharm(&imageLocal, ((global_y_size-2)/size)+2, (global_x_size));
    malloc2dcharm(&imageOpLocal, ((global_y_size-2)/size), (global_x_size-2));

    if(rank == MASTER)
    {
      temp = 0;
      for(i = 0; i < size; i++) {
          sendcount[i] = 1;
          recvcount[i] = 1;
          displ[i] = temp;
          recvdispl[i] = temp;
          temp += ((global_y_size-2)/size);
      }
      opImageptr = &(image2[0][0]);
 }

    //  printf("y size %d x size %d\n\n",global_y_size,global_x_size);
```

```
 MPI_Scatterv(ipImageptr, sendcount, displ, scatter_subarraytype,
&(imageLocal[0][0]), (((global_y_size-2)/size)+2)*     global_x_size,
MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);


   sobel_filtering(rank, size);


   MPI_Gatherv(&(imageOpLocal[0][0]), ((global_y_size-
2)/size)*(global_x_size-2), MPI_UNSIGNED_CHAR, opImageptr,
recvcount, recvdispl, gather_subarraytype, 0, MPI_COMM_WORLD);

   if(rank == MASTER){
     //timeEnd = MPI_Wtime();
     timeEnd = omp_get_wtime();
   }

   free2dchar(&imageLocal);
   free2dchar(&imageOpLocal);

   MPI_Type_free(&scatter_subarraytype);
   MPI_Type_free(&gather_subarraytype);

   if(rank == MASTER){


     printf("The time taken %1.2f\n",(timeEnd - timeStart));
     x_size2 = global_x_size-2;
     y_size2 = global_y_size-2;

     save_image_data();   /* Output of image2 */

     free2dchar(&image2);
   }

   MPI_Finalize();
   return 0;
 }
```

Sobel Program Implemented only with “omp parallel” construct:

```
/* sobel.c – Only OMP parallel was applied. No other construct was used*/
 #include <stdio.h>
 #include <stdlib.h>
 #include <float.h>
 #include "mypgm.h"
 #include <math.h>
 #include <omp.h>

 #define num_threads 4

 void sobel_filtering( )
      /* Spatial filtering of image data */
      /* Sobel filter (horizontal differentiation */
      /* Input: image1[y][x] ---- Outout: image2[y][x] */
 {
   /* Definition of Sobel filter in horizontal direction */
```

```
    int weight_x[3][3] =   {{ -1,   0,   1 },
                            { -2,   0,   2 },
                            { -1,   0,   1 }};

    /* Definition of Sobel filter in vertical direction */
    int weight_y[3][3] =   {{ -1,  -2,  -1 },
                            {  0,   0,   0 },
                            {  1,   2,   1 }};
    double min[num_threads], max[num_threads];
    double min_value, max_value;
    /* Maximum values calculation after filtering*/
    printf("Now, filtering of input image is performed\n\n");
    min_value = DBL_MAX;
    max_value = -DBL_MAX;

    int v;
    for(v = 0; v < num_threads;v++)
    {
      min[v] = DBL_MAX;
      max[v] = -DBL_MAX;
    }

    #pragma omp parallel
    {
    double pixel_value;
    double pixel_value_x;
    double pixel_value_y;

    int x, y, i, j;  /* Loop variable */
    int thr_id = omp_get_thread_num ();

    for (y = (1+thr_id); y < y_size1 + 1; y = y + num_threads) {
    for (x = 1; x < x_size1 + 1; x++) {
        pixel_value_x = 0.0;
        pixel_value_y = 0.0;
        pixel_value = 0.0;

        for (j = -1; j <= 1; j++) {
          for (i = -1; i <= 1; i++) {

            pixel_value_x += weight_x[j + 1][i + 1] * image1[y + j][x + i];
            pixel_value_y += weight_y[j + 1][i + 1] * image1[y + j][x + i];

          }
        }

        pixel_value = sqrt((pixel_value_x * pixel_value_x) + (pixel_value_y *
pixel_value_y));

        if (pixel_value < min[thr_id]) min[thr_id] = pixel_value;
        if (pixel_value > max[thr_id]) max[thr_id] = pixel_value;
      }
    }
    }

    int m,p,q;

    for(m = 0;m<num_threads;m++)
    {
```

```
        if (min[m] < min_value) min_value = min[m];
        if (max[m] > max_value) max_value = max[m];
    }

    printf("the value of minimum  %f\n",min_value);
    printf("the value of maximum  %f\n",max_value);

    if ((int)(max - min) == 0) {
      printf("Nothing exists!!!\n\n");
      exit(1);
    }

    /* Initialization of image2[y][x] */
    x_size2 = x_size1;
    y_size2 = y_size1;
    for (p = 0; p < y_size2; p++) {
      for (q = 0; q < x_size2; q++) {
        image2[p][q] = 0;
      }
    }



    /* Generation of image2 after linear transformtion */
    #pragma omp parallel
    {
    double pixel_value;
    double pixel_value_x;
    double pixel_value_y;
    int x, y, i, j;  /* Loop variable */
    int thr_id = omp_get_thread_num ();

    for (y = (1+thr_id); y < y_size1 + 1; y = y + num_threads) {
    for (x = 1; x < x_size1 + 1; x++) {
      pixel_value_x = 0.0;
        pixel_value_y = 0.0;
        pixel_value = 0.0;

        for (j = -1; j <= 1; j++) {
          for (i = -1; i <= 1; i++) {
            pixel_value_x += weight_x[j + 1][i + 1] * image1[y + j][x + i];
            pixel_value_y += weight_y[j + 1][i + 1] * image1[y + j][x + i];
          }
        }

        pixel_value = sqrt((pixel_value_x * pixel_value_x) + (pixel_value_y *
pixel_value_y));

        pixel_value = MAX_BRIGHTNESS * (pixel_value - min_value) / (max_value
- min_value);
        image2[y][x] = (unsigned char)pixel_value;
          }
        }
     }
 }

 main( )
 {
    load_image_data( );   /* Input of image1 */
```

```
  double start_time = omp_get_wtime();
  sobel_filtering( );   /* Sobel filter is applied to image1 */
  printf("The total time for filering is %f", omp_get_wtime() - start_time);
  save_image_data( );   /* Output of image2 */
  return 0;
}
```

Bad Examples 1:

```
/* sobel.c – With Two level nested loop, thread id being added to
Both the loops results in pixels getting missed out of convolution*/
 #include <stdio.h>
 #include <stdlib.h>
 #include <float.h>
 #include "mypgm.h"
 #include <math.h>
 #include <omp.h>

 #define num_threads 4

 void sobel_filtering( )
      /* Spatial filtering of image data */
      /* Sobel filter (horizontal differentiation */
      /* Input: image1[y][x] ---- Outout: image2[y][x] */
 {
   /* Definition of Sobel filter in horizontal direction */
   int weight_x[3][3] =   {{ -1,   0,   1 },
                           { -2,   0,   2 },
                           { -1,   0,   1 }};

   /* Definition of Sobel filter in vertical direction */
   int weight_y[3][3] =   {{ -1,  -2,  -1 },
                           {  0,   0,   0 },
                          {  1,   2,   1 }};
   double min[num_threads], max[num_threads];
   double min_value, max_value;
   /* Maximum values calculation after filtering*/
   printf("Now, filtering of input image is performed\n\n");
   min_value = DBL_MAX;
   max_value = -DBL_MAX;

   int v;
   for(v = 0; v < num_threads;v++)
   {
     min[v] = DBL_MAX;
     max[v] = -DBL_MAX;
   }

   #pragma omp parallel
   {
   double pixel_value;
   double pixel_value_x;
   double pixel_value_y;

   int x, y, i, j;  /* Loop variable */
   int thr_id = omp_get_thread_num ();
```

```
  for (y = (1+thr_id); y < y_size1 + 1; y = y + num_threads) {
   for (x = (1+thr_id); x < x_size1 + 1; x = x + num_threads) {

        pixel_value_x = 0.0;
        pixel_value_y = 0.0;
        pixel_value = 0.0;

        for (j = -1; j <= 1; j++) {
          for (i = -1; i <= 1; i++) {

            pixel_value_x += weight_x[j + 1][i + 1] * image1[y + j][x + i];
            pixel_value_y += weight_y[j + 1][i + 1] * image1[y + j][x + i];

          }
        }

        pixel_value = sqrt((pixel_value_x * pixel_value_x) + (pixel_value_y *
pixel_value_y));

        if (pixel_value < min[thr_id]) min[thr_id] = pixel_value;
        if (pixel_value > max[thr_id]) max[thr_id] = pixel_value;
      }
    }
   }

    int m,p,q;

    for(m = 0;m<num_threads;m++)
    {
        if (min[m] < min_value) min_value = min[m];
        if (max[m] > max_value) max_value = max[m];
    }

    printf("the value of minimum  %f\n",min_value);
    printf("the value of maximum  %f\n",max_value);

    if ((int)(max - min) == 0) {
      printf("Nothing exists!!!\n\n");
      exit(1);
    }

    /* Initialization of image2[y][x] */
    x_size2 = x_size1;
    y_size2 = y_size1;
    for (p = 0; p < y_size2; p++) {
      for (q = 0; q < x_size2; q++) {
       image2[p][q] = 0;
      }
    }



    /* Generation of image2 after linear transformtion */
    #pragma omp parallel
    {
    double pixel_value;
    double pixel_value_x;
    double pixel_value_y;
    int x, y, i, j;  /* Loop variable */
```

```
   int thr_id = omp_get_thread_num ();

   for (y = (1+thr_id); y < y_size1 + 1; y = y + num_threads) {
   for (x = (1+thr_id); x < x_size1 + 1; x = x + num_threads) {
       pixel_value_x = 0.0;
       pixel_value_y = 0.0;
       pixel_value = 0.0;

       for (j = -1; j <= 1; j++) {
         for (i = -1; i <= 1; i++) {
          pixel_value_x += weight_x[j + 1][i + 1] * image1[y + j][x + i];
           pixel_value_y += weight_y[j + 1][i + 1] * image1[y + j][x + i];
         }
       }

       pixel_value = sqrt((pixel_value_x * pixel_value_x) + (pixel_value_y *
pixel_value_y));

       pixel_value = MAX_BRIGHTNESS * (pixel_value - min_value) / (max_value
-    min_value);
       image2[y][x] = (unsigned char)pixel_value;
         }
       }
     }
 }

 main( )
 {
   load_image_data( );   /* Input of image1 */
   double start_time = omp_get_wtime();
   sobel_filtering( );   /* Sobel filter is applied to image1 */
   printf("The total time for filering is %f", omp_get_wtime() - start_time);
   save_image_data( );   /* Output of image2 */
  return 0;
 }
```

Bad Examples 2:

```
/* sobel.c  - Bad example where the parallelizing the pixel for each
convolution resulting in high cache miss rate and high run time*/
 #include <stdio.h>
 #include <stdlib.h>
 #include <float.h>
 #include "mypgm.h"
 #include <math.h>
 #include <omp.h>

 #define num_threads 4


 void sobel_filtering( )
      /* Spatial filtering of image data */
      /* Sobel filter (horizontal differentiation */
      /* Input: image1[y][x] ---- Outout: image2[y][x] */
 {
   /* Definition of Sobel filter in horizontal direction */
   int weight_x[3][3] =   {{ -1,   0,   1 },
                          { -2,   0,   2 },
                          { -1,   0,   1 }};
```

```
  /* Definition of Sobel filter in vertical direction */
   int weight_y[3][3] =   {{ -1,  -2,  -1 },
                           {  0,   0,   0 },
                           {  1,   2,   1 }};

   omp_set_num_threads(num_threads);


   double pixel_value_x;
   double pixel_value_y;
   double pixel_value;

   double min, max;
   int x, y, i, j;  /* Loop variable */

   /* Maximum values calculation after filtering*/
   printf("Now, filtering of input image is performed\n\n");
   min = DBL_MAX;
   max = -DBL_MAX;

   printf("The ysize and xsize is %d and %d", y_size1,x_size1 );
   pixel_value = 0.0;


   for (y = 1; y < y_size1 + 1; y++) {
     for (x = 1; x < x_size1 + 1; x++) {
       pixel_value_x = 0.0;
       pixel_value_y = 0.0;
       pixel_value = 0.0;
 //      #pragma omp parallel for collapse(2) private(i,j) reduction(+:
pixel_value_x,pixel_value_y)
       #pragma omp parallel for collapse(2)  reduction(+:pixel_value_x,
pixel_value_y)
       for (j = -1; j <= 1; j++) {
         for (i = -1; i <= 1; i++) {
           pixel_value_x += weight_x[j + 1][i + 1] * image1[y + j][x + i];
           pixel_value_y += weight_y[j + 1][i + 1] * image1[y + j][x + i];
         }
       }
       pixel_value = sqrt((pixel_value_x * pixel_value_x) + (pixel_value_y *
pixel_value_y));

       if (pixel_value < min) min = pixel_value;
       if (pixel_value > max) max = pixel_value;
     }
   }
printf("The Value of Minimum  %f\n",min);
   printf("The Value of Maximum  %f\n",max);

   if ((int)(max - min) == 0) {
     printf("Nothing exists!!!\n\n");
     exit(1);
   }

   /* Initialization of image2[y][x] */
   x_size2 = x_size1;
   y_size2 = y_size1;

   #pragma omp parallel for collapse(2) private(x,y)\
```

```
                                        shared(image2)
   for (y = 0; y < y_size2; y++) {
     for (x = 0; x < x_size2; x++) {
       image2[y][x] = 0;
     }
   }


   /* Generation of image2 after linear transformtion */
   for (y = 1; y < y_size1 + 1; y++) {
    for (x = 1; x < x_size1 + 1; x++) {
       pixel_value_x = 0.0;
       pixel_value_y = 0.0;
       pixel_value = 0.0;
       //#pragma omp parallel for collapse(2) private(i,j) reduction(+:
pixel_value_x,pixel_value_y)
       #pragma omp parallel for collapse(2) reduction(+:pixel_value_x,
pixel_value_y)
       for (j = -1; j <= 1; j++) {
         for (i = -1; i <= 1; i++) {
           pixel_value_x += weight_x[j + 1][i + 1] * image1[y + j][x + i];
           pixel_value_y += weight_y[j + 1][i + 1] * image1[y + j][x + i];
         }
       }

       pixel_value = sqrt((pixel_value_x * pixel_value_x) + (pixel_value_y *
pixel_value_y));

       pixel_value = MAX_BRIGHTNESS * (pixel_value - min) / (max - min);

       image2[y - 1][x - 1] = (unsigned char)pixel_value;
     }
   }
 }

 main( )
 {
   load_image_data( );   /* Input of image1 */
   double start_time = omp_get_wtime();
   sobel_filtering( );   /* Sobel filter is applied to image1 */
   printf("The total time for filering is %f", omp_get_wtime() - start_time);
   save_image_data( );   /* Output of image2 */
   return 0;
 }
```