Running a job:

To reproduce the different experiments that are described in this report, we provide multiple shell scripts that automate the compilation and submission of jobs. To run part1, one can compile using the command "**gcc -o sobel sobel.c**" and run with **./sobel**.

For simplicity, in part2, the script "**com_del_sub.sh**" is the one to run to submit a job. The script job-hw*.sh may also be updated to modify the execution parameters. Please upload all the **.sh** files to HiperGator as they are pre-requisite to run **com_del_sub.sh**.

Authors: Joel Mandebi Mbongue, Samantha-Jo Cunningham

Part 1: Simple Sobel Implementation

We implemented a custom Sobel edge detector. The C code is presented below:

```
#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <time.h>

#define N 3000

int input_image [N][N];
int result_image [N][N];

struct timespec begin, end;
double time_spent;

/*   ttype: type to use for representing time */
typedef double ttype;
ttype tdiff(struct timespec a, struct timespec b)
/* Find the time difference. */
{
  ttype dt = (( b.tv_sec - a.tv_sec ) + ( b.tv_nsec - a.tv_nsec ) / 1E9);
  return dt;
}



struct timespec now()
/* Return the current time. */
{
  struct timespec t;
  clock_gettime(CLOCK_REALTIME, &t);
  return t;
}

void printArray(int A[N][N]){
    int i, j;

    for(i=0;i<N;i++){
                for(j=0;j<N;j++)
                printf(" %d", A[i][j]);
                printf("\n");
    }
}


void load_image( ){
  FILE *fp;
  int i,j, num;

  fp = fopen("input.txt", "r");

  if(fp == NULL) {
                 printf("Error in opening file\n");
                 exit(1);
  }

  i = 0;
  j = 0;
  while( fscanf(fp, "%d", &input_image[i][j])!=EOF )
   {
                 //printf(" input_image[%d][%d] = %d \n", i,j,input_image[i][j]);
     j++;
     if(j==N){
                   j = 0;
                   i++;
     }
   }
   //printArray(input_image);
  fclose(fp);
}

void save_image( ){

  FILE *fp;
  int i,j, num;

  fp = fopen("output.txt", "w");

  if(fp == NULL) {
                 printf("Error in opening file\n");
                 exit(1);
  }

  for(i=0;i<N;i++){
                 for(j=0;j<N;j++)
                   if(j == 0)
                                 fprintf(fp, "%d", result_image[i][j]);
                   else
                 fprintf(fp, " %d", result_image[i][j]);
                 fprintf(fp, "\n");
  }

  fclose(fp);
}

void apply_sobel( ){
  int i,j;

  for(i=0;i<N;i++)
   for(j=0;j<N;j++){
     if(i == 0   || i == (N-1) || j==0 || j == (N-1)  )
                 result_image[i][j] = 0; //we set all the edges to 0
                 else
                                  result_image[i][j] = abs( (input_image[i-1][j-1]-input_image[i+1][j-1]) + 2*(input_image[i-1][j]-input_image[i+1][j]) +
(input_image[i-1][j+1]-input_image[i+1][j+1]) )
                                 +abs((input_image[i-1][j-1]-input_image[i-1][j+1]) + 2*(input_image[i][j-1]-input_image[i][j+1]) + (input_image[i+1][j-1]-
input_image[i+1][j+1]) );

   }
   //printArray(result_image);
}

int main()
{
  begin = now();
     load_image( );
     apply_sobel( );
     save_image( );
  end = now();

  time_spent = tdiff(begin, end);
                printf("\n  --> Total time: %.8f sec\n\n", time_spent);

  return 0;
}
```

On Matlab, we used the provided conversion functions (from image to text, and from text to image) below:

```
img = imread("image.jpg");
gray = rgb2gray(img);
resized_gray = imresize(gray, [3000, 3000]);
writematrix(resized_gray, 'input.txt', 'Delimiter', 'tab');
```

```
i=dlmread("output.txt");
i=uint8(i);
imshow(i);
```

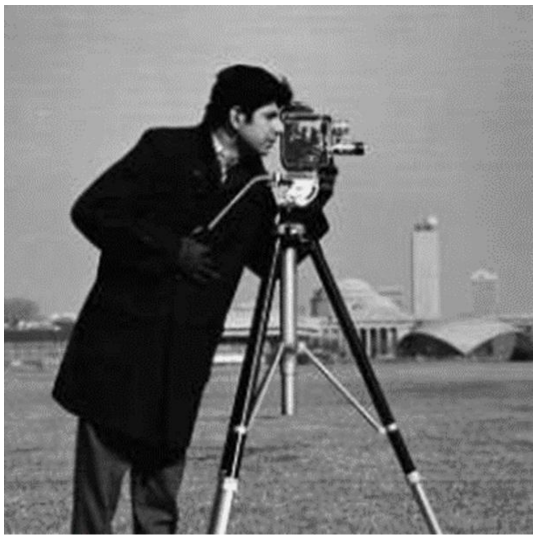


Fig1. Input image

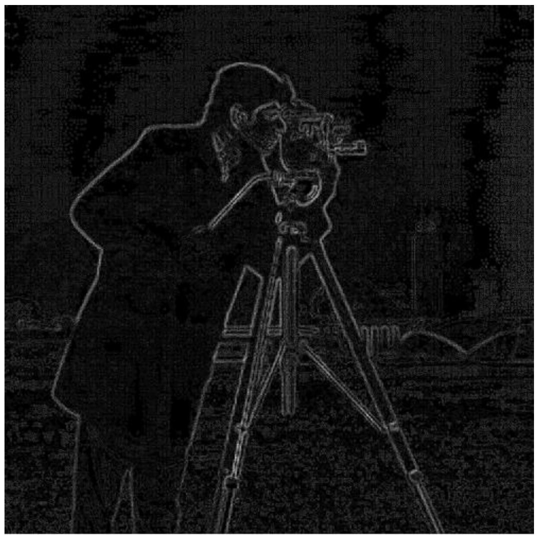
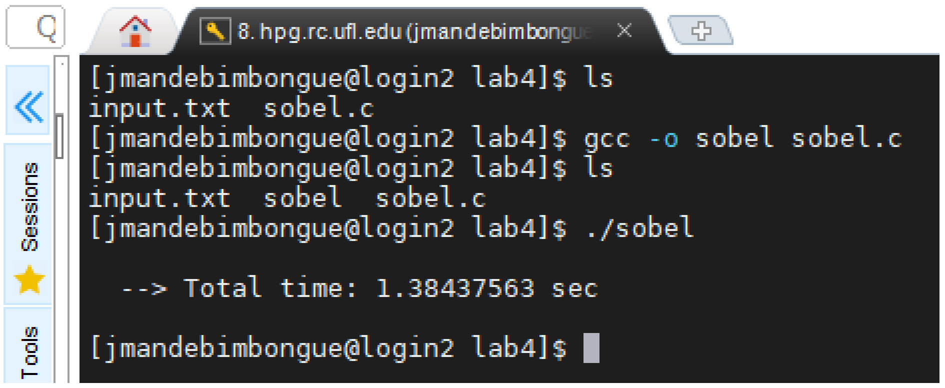


Fig2. Result Image

The program executes in **1.38437563 seconds** as illustrated in the figure below:



```
[jmandebimbongue@login2 lab4]$ ls
input.txt  sobel.c
[jmandebimbongue@login2 lab4]$ gcc -o sobel sobel.c
[jmandebimbongue@login2 lab4]$ ls
input.txt  sobel  sobel.c
[jmandebimbongue@login2 lab4]$ ./sobel

  --> Total time: 1.38437563 sec

[jmandebimbongue@login2 lab4]$
```

Fig3. Screenshot showing the compilation and execution of the simple Sobel detector

Part 2: Design Space Exploration

In this section we will be exploring strong scaling on the sobel filter implemented in part 1. Only strong scaling will be explored for this problem as the input image is of a fixed size (3000x3000). To begin we will first look at an implementation of the sobel filter with MPI only and compare the execution time as we increase the ranks. For further exploration we will look at the sobel filter using both mpi and omp and record the execution time as we vary the number of threads, ranks and nodes.

A. MPI Only

Number of Ranks (1 node)	**Execution Time (seconds)**	**Picture**
1	0.21213242	1rankMPI.png
2	0.18217222	2rankMPI.png
4	0.05896580	4ranksMPI.png
8	0.04538591	8rankMPI.png
16	0.07072464	16rankMPI.png
32	0.08077950	32rankMPI.png

Table 1: Sobel Filter with MPI only

In general, in Table1 we observe that as we increase the number of ranks, tends to slightly decrease. For instance, with 1 and 2 ranks we observe 0.21213242 seconds and then 0.18217222 seconds. However, after 8 ranks, we note that the execution time slightly increases while still remaining lower than with 1 and 2 ranks. The execution time increase observed with 16 and 32 ranks may just be a consequence of the communication overhead. Though sharing computations among the workers speeds up the computation time, the overall communication overhead mitigates the computation gain.

In the next section, we will explore the performance (execution time) achieved when introducing threads on each worker rank.

B. MPI + OpenMP

Number of Ranks	Number of Nodes	Number of Threads	Execution Time (s)	Picture
4	1	4	0.15103336	4ranks1node4Threads MPI_OMP.png
2	2	4	0.16336390	2ranks2node4Threads MPI_OMP.png
1	4	4	0.14760093	1ranks4node4Threads MPI_OMP.png

Table 2: Sobel Filter with MPI + OpenMP

Table2 records the execution time for 4 tasks spread across different configurations (1 node running 4 ranks, 2nodes, each running 2 ranks, and 4 nodes each running 1 rank). We observe that the execution time does not significantly differ in the different configurations. We also note that these configurations seem to be slower than the results observed in Table 1 (when using at least 4 ranks). In an attempt to improve the quality of result (QoR) when combining MPI and OpenMP, we run additional experiments. First, both scale the number of ranks and the number of threads. Next, we scale the number of threads and the number of nodes while off-loading the workload to another node. We want to assess if using multiple nodes can reduce the pressure on a CPU core, and eventually reduce the overall execution time. This choice is motivated by the fact that the lowest execution time in Table2 is recorded with the highest number of nodes.

Below we present the results of scaling the number of threads and ranks with the number of nodes:

Number of Ranks	Number of Nodes	Number of Threads	Execution Time (s)
1	1	1	0.16394847
2	1	2	0.17410919
4	1	4	0.07184341
8	1	8	0.22793713
16	1	16	0.29365027
32	1	32	0.40896401

Table 3: Sobel Filter with MPI + OpenMP varying ranks and threads

After running this experiment, it was observed that varying the number of ranks and threads on 1 node, there was a speedup until from 1 rank 1 thread to 4 ranks 4 threads. However, when increased to 8, 16 and 32 the execution time increased which means there was no speed up. This increase in execution time was caused by increased communication.

Number of Ranks per Nodes	**Number of Nodes**	**Number of Threads**	**Execution Time (s)**
1	2	2	0.08733923
2	2	4	0.05720718
4	2	8	0.07282508
8	2	16	0.03593145
16	2	32	0.06529058

Table 4: Sobel Filter with MPI + OpenMP varying ranks and threads with 2 Nodes

Table 4 shows the execution time when spreading the same workload of Table3 across 2 nodes. We directly note that the QoR is improved compared to the results from Table 3. In Table 3, only the execution time with 4 ranks and 4 threads was similar to that of Table 1 (4ranks - 32 ranks). However, when we increased the number of nodes to 2 there is significant speed up. We even obtain the best execution time of all the experiments carried out with 2 nodes, 8 ranks per node, and 16 threads per rank. This suggests that increasing the nodes can improve the QoR as we achieve about **2X** speed up when comparing the best QoR of Table3 and Table4.

Conclusion:

Throughout this experiment we realized that parallel computing increases performance. However, it can also suffer from high communication overhead. In the execution time of part 1, where there was no parallelism, the sobel filter took over **1.38 seconds** to compute a [3000]*[3000] image. While using OpenMP and MPI the computation took less than 1 second on each node configuration tested, with the best configuration coming from using 8 ranks per node, 2 nodes and 16 threads per rank, which achieves **39X** speedup compared to the serial implementation.