Ojani Eguia

287 Term Project

Term Project Report

For this project, I have created an image processor that applies a 'shade' to a base Image and produces the result in a ppm (Portable PixMap) image file. As of now, the program does not take input files, but can generate input information based on variable (More Details about this in the Readme). This processor has been implemented in C with Sequential, OpenMP, and MPI versions. The implementation of this processor is essentially a three- dimensional array for the result and both inputs, which depend on the vertical and horizontal length (in pixels) of the image to be generated, as well as the three color components (Red, Green, Blue) of each pixel. The priority for sequential implementation is simplicity and priority for OpenMP implementation is Speed increase, while priority for MPI implementation is increasing maximum size of output.

During testing, High Performance was gauged as how fast ('time') a program would run as the size of the image increases, and as working threads are added (If applicable). In addition, while memory ('space') used was not specifically measured, the results of the tests showed a clear difference between versions as to how memory is handled.

As stated above, the main characteristic measured was running time, but things like capability at larger file sizes were also considered. For all tests, the input files were set to random generation. Below are the tables with running times for each of the versions. First, Comparisons in seconds between Sequential and OpenMP Running Times.

Size of File	100*100	200*200	300*300	400*400
Sequential	0.016	0.021	0.038	0.065
1 Thread	0.014	0.02	0.038	0.066
2 Thread	0.009	0.022	0.047	0.066
3 Thread	0.027	0.036	0.063	0.095
4 Thread	0.052	0.057	0.104	0.122
5 Thread	0.013	0.021	0.041	0.064
6 Thread	0.009	0.022	0.038	0.069
7 Thread	0.011	0.025	0.049	0.072

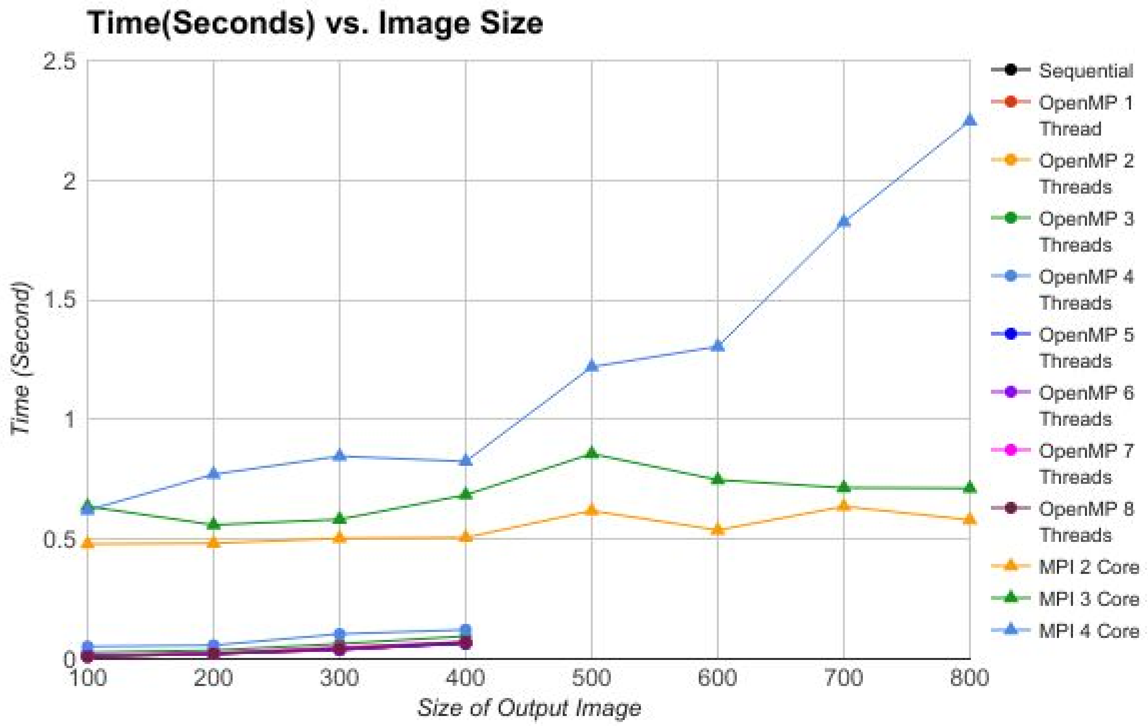
8 Thread	0.008	0.022	0.043	0.071

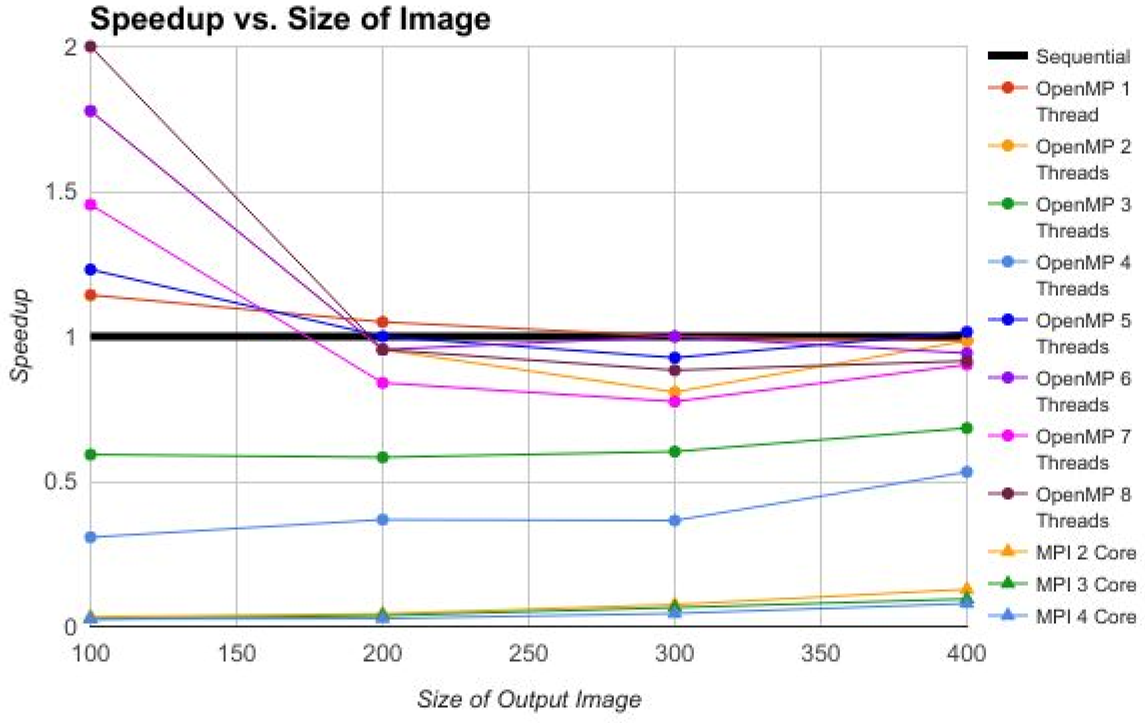
Below is a table comparing the running times, in seconds, between the sequential time and the MPI running times. There were Points of Interest regarding the implementation of MPI. First, I was unable to get MPI to run on multiple machines, so The tests below reflect MPI running numerous cores on a single machine. Since there are only 4 cores per machine, test could only be run on up to 4 cores. Second, The program is set so that the Master core does no work other than collecting the data and turning it into a ppm file. This is done to prevent unnecessary allocation of memory to arrays that will not be used. The Master only allocates space for a result array and the worker cores only allocate space for arrays used in calculation. For this reason, properly running MPI with one core is not possible.

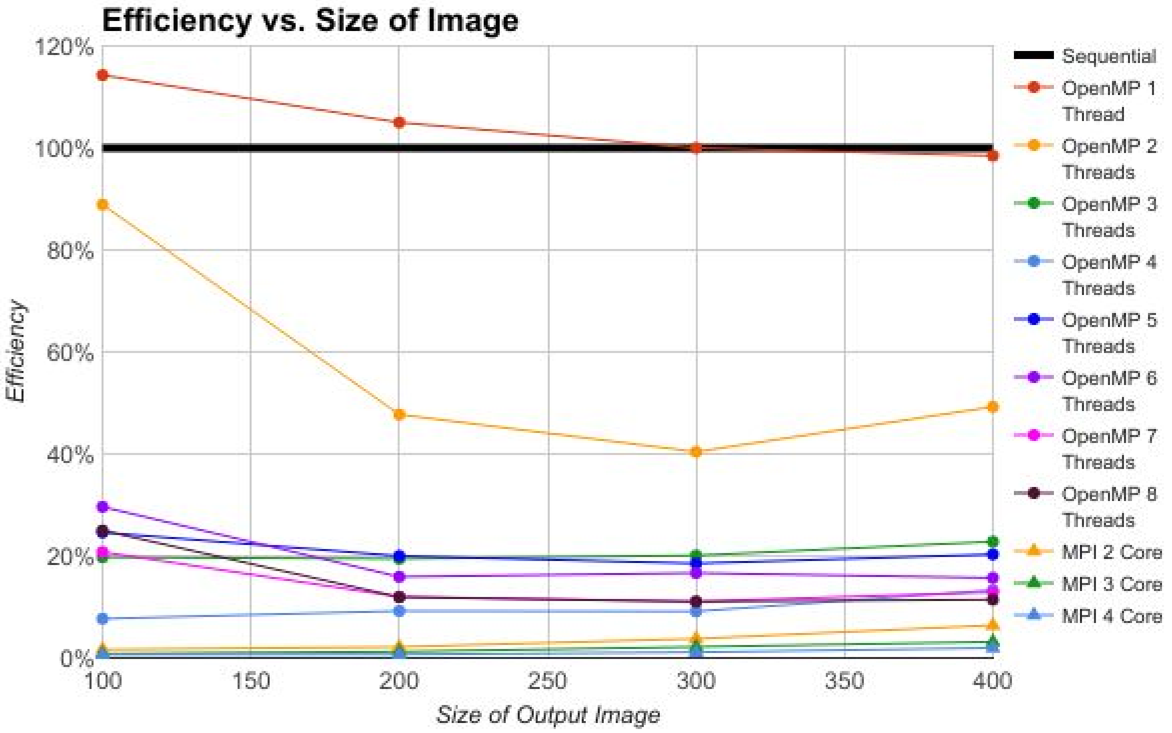
Note the lack of results for sequential after size 400: This is due to Segmentation fault, or being unable to carry out the program due to lack of space.

Size	100	200	300	400	500	600	700	800
Seq.	0.016	0.021	0.038	0.065				
2 Core	0.48	0.483	0.504	0.507	0.619	0.538	0.637	0.581
3 Core	0.637	0.56	0.583	0.685	0.857	0.748	0.715	0.713
4 Core	0.621	0.771	0.847	0.825	1.22	1.305	1.825	2.248

Below are the graphical representations for the time results, speedup, and efficiency for each test run. Speedup for each recorded time is defined as Speedup = Sq / RT, where Sq is the the Sequential running time, and RT is the running time for that number of processors. Efficiently is defined as Speedup divided by Number of processors. Note the lack of Speedup and efficiency values for arrays above 400; This is the result of lack of sequential results for these sizes.


Time(Seconds) vs. Image Size
Time (Second)
2.5
2
1.5
1
0.5
0
100
200
300
400
500
600
700
800
Size of Output Image
Sequential
OpenMP 1 Thread
OpenMP 2 Threads
OpenMP 3 Threads
OpenMP 4 Threads
OpenMP 5 Threads
OpenMP 6 Threads
OpenMP 7 Threads
OpenMP 8 Threads
MPI 2 Core
MPI 3 Core
MPI 4 Core


Speedup vs. Size of Image
Speedup
2
1.5
1
0.5
0
100
150
200
250
300
350
400
Size of Output Image
Sequential
OpenMP 1 Thread
OpenMP 2 Threads
OpenMP 3 Threads
OpenMP 4 Threads
OpenMP 5 Threads
OpenMP 6 Threads
OpenMP 7 Threads
OpenMP 8 Threads
MPI 2 Core
MPI 3 Core
MPI 4 Core



Add Graphs here

Initial observations were that for all tests, memory limits the size of project that can be run. Due to the size being limited to several hundred pixel size images, the comparatively large overhead for both OpenMP and MPI is so large that the use of them actually increases running times compared to sequential time. In addition, there is a decrease in running time between use of 4 and 5 threads in OpenMP, which could be the result of using both hardware threads on a core.

The results for this project are perhaps more interesting than results I have seen in other projects for this course. The running times for OpenMP were roughly similar but slightly higher than the comparable sequential running times. Theoretically OpenMP should comparatively have faster running times but same size capacity as sequential. However, due to Overhead, OpenMP seems to realistically have no positive or negative effects on running times over Sequential with this current implementation.

More interesting were the results of MPI, which at first were confusing. Even though the MPI version was implemented to maximize size capacity, I thought it would make no difference to running times since MPI in this test was limited to running on a single machine with 4 cores, which is what OpenMP was run on. As such, I expected similar results between OpenMP and

MPI: MPI would not improve size and increase running times even more than OpenMP since MPI involves direct message passing and this is the most costly part of the program. However, While the MPI running times are 8-9 times longer than sequential running times, MPI can produce images double the size of what Sequential and OpenMP can produce. Considering that Space requirements for the problem are 3 arrays of size (length)(length)(3), and doubling the size of the image quadruples the numbers of pixels in each image, MPI is at least quadratically more space efficient than Sequential implementation.

These results really reflect the beginning conversation of the course, in which we talked about the difference between aiming for faster running times as proposed by Amdahl and increasing problem size as proposed by Gustafson. This is a choice in approach that we have seen since the beginning of the course as we learned about the basics of OpenMP and MPI. In addition, we have considered the cost of overhead and communication between multiple cores, and the differences between shared memory and message passing shows in these results immensely. Lastly, the article I read for the journal project, *Parallel Graph Analysis*, made a point that rewriting code that increases complexity may be worth it if it is to improve parallel running times, and that is a point that I considered when implementing in the MPI version of this problem, and it improved the results beyond what I had expected.

Works Cited:

Source of converting array to ppm / source of cool Amazing Matrix file;

http://rosettacode.org/wiki/Bitmap/Write_a_PPM_file#C

How to generate single sheet of color ppm:

https://people.cs.clemson.edu/~levinej/courses/S15/1020/handouts/lec05/creatingPPM.pdf

Appendix :

1. Tables for Speedup and Efficiency

Speedup	Size	100	200	300	400
	Sequential	1	1	1	1
OpenMPI	1 Thread	1.142857143	1.05	1	0.9848484848
OpenMPI	2 Threads	1.777777778	0.9545454545	0.8085106383	0.9848484848
OpenMPI	3 Threads	0.5925925926	0.5833333333	0.6031746032	0.6842105263
OpenMPI	4 Threads	0.3076923077	0.3684210526	0.3653846154	0.5327868852
OpenMPI	5 Threads	1.230769231	1	0.9268292683	1.015625
OpenMPI	6 Threads	1.777777778	0.9545454545	1	0.9420289855
OpenMPI	7 Threads	1.454545455	0.84	0.7755102041	0.9027777778
OpenMPI	8 Threads	2	0.9545454545	0.8837209302	0.9154929577
MPI	2 Core	0.03333333333	0.04347826087	0.0753968254	0.1282051282
MPI	3 Core	0.0251177394	0.0375	0.06518010292	0.09489051095
MPI	4 Core	0.02576489533	0.02723735409	0.04486422668	0.07878787879

Efficiency	Size	100	200	300	400
	Sequential	1	1	1	1
OpenMPI	1 Thread	114.2857143	105	100	98.48484848
OpenMPI	2 Threads	88.88888889	47.72727273	40.42553191	49.24242424
OpenMPI	3 Threads	19.75308642	19.44444444	20.10582011	22.80701754
OpenMPI	4 Threads	7.692307692	9.210526316	9.134615385	13.31967213
OpenMPI	5 Threads	24.61538462	20	18.53658537	20.3125
OpenMPI	6 Threads	29.62962963	15.90909091	16.66666667	15.70048309
OpenMPI	7 Threads	20.77922078	12	11.0787172	12.8968254

OpenMPI	8 Threads	25	11.93181818	11.04651163	11.44366197
MPI	2 Core	1.666666667	2.173913043	3.76984127	6.41025641
MPI	3 Core	0.8372579801	1.25	2.172670097	3.163017032
MPI	4 Core	0.6441223833	0.6809338521	1.121605667	1.96969697

2. Code:

Sequential

```
/*
Ojani Eguia
287 Term Project
Image Generator / Processor
Sequential Version
*/
#include <stdbool.h>
#include <stdio.h>
#include <time.h>

void main(){
    int printing = 0;
    int dimention = 256;
    int i = 0,j = 0; // Indices
    int generation = 7;
    /*
    Instructions for Generate:

    For First Input:
    Even numbers start with awesome base,
    Odd numbers generate random fields.

    For Second Input:
    0/1 is no change
    2/3 is Shift to red
    4/5 is Shift to green
    6/7 is Shift to blue
    anything else is random


    NOTE: Cool array is thanks to
http://rosettacode.org/wiki/Bitmap/Write_a_PPM_file#C
```

```
    */
    srand(time(NULL));   // should only be called once

    int Input1[dimention][dimention][3];
    int Input2[dimention][dimention][3];
    int Result[dimention][dimention][3];


    for (i = 0; i < dimention; i++){//FIRST INPUT ARRAY
      for (j = 0; j < dimention; j++){
           int a, b, c;
           if((generation % 2) == 0){//cool array
                a = i % 256;    /* red */
                b = j % 256;    /* green */
                c = (i * j) % 256;    /* blue */
           }
           else{//random
                a = rand()%0x100;
                b = rand()%0x100;
                c = rand()%0x100;
           }
           Input1[i][j][0] = a;
           Input1[i][j][1] = b;
           Input1[i][j][2] = c;
      }
    }

    for (i = 0; i < dimention; i++){//SECOND INPUT ARRAY
      for (j = 0; j < dimention; j++){
           int a, b, c;
           if((generation == 1) || (generation == 0)){//no
change
                a = 0xff;
                b = 0xff;
                c = 0xff;
           }
           else if((generation == 2) || (generation == 3)){//red
                a = 0xff;
                b = 0x80;
                c = 0x80;
           }
           else if((generation == 4) || (generation ==
5)){//green
                a = 0x80;
                b = 0xff;
                c = 0x80;
           }
           else if((generation == 6) || (generation ==
7)){//blue
                a = 0x80;
                b = 0x80;
                c = 0xff;
           }
           else{//random
                a = rand()%0x100;
                b = rand()%0x100;
                c = rand()%0x100;
           }
           Input2[i][j][0] = a;
           Input2[i][j][1] = b;
           Input2[i][j][2] = c;
      }
    }

    if (printing == 1){
      printf("Here is Input 1 before Multiplication: \n");
      int a = 0, b = 0;// Indices
      for (a = 0; a < dimention; a++){
           for (b = 0; b < dimention; b++){
                printf("%.2x ,", Input1[a][b][0]);
                printf("%.2x ,", Input1[a][b][1]);
                printf("%.2x \t", Input1[a][b][2]);
           }
           printf("\n");
      }
    }

    if (printing == 1){
      printf("Here is Input 2 before Multiplication: \n");
      int a = 0, b = 0;// Indices
      for (a = 0; a < dimention; a++){
           for (b = 0; b < dimention; b++){
                printf("%.2x ,", Input2[a][b][0]);
                printf("%.2x ,", Input2[a][b][1]);
                printf("%.2x \t", Input2[a][b][2]);
           }
           printf("\n");
      }
    }
    //gonna overlap the files now

    int d = 0,e = 0, f = 0; // Indices
    for (d = 0; d < dimention; d++){
     for (e = 0; e < dimention; e++){
        for(f = 0; f < 3; f++){//Algorithm is based on
overlapping of colors: how much Input2 "affects' input a based
on intensity of that component in input2
          double intA = Input2[d][e][f];
          double percentage = (intA/0x100);
          int newValue = (Input1[d][e][f] * percentage);
          Result[d][e][f] = newValue+1;
        }
     }
    }

    if (printing == 1){
     printf("Here is Result after Multiplication: \n");
     int a = 0, b = 0;// Indices
     for (a = 0; a < dimention; a++){
        for (b = 0; b < dimention; b++){
          printf("%.2x ,", Result[a][b][0]);
          printf("%.2x ,", Result[a][b][1]);
          printf("%.2x \t", Result[a][b][2]);
        }
        printf("\n");
     }
    }
    //The below is adapted from
http://rosettacode.org/wiki/Bitmap/Write_a_PPM_file#C

    const int dimx = dimention, dimy = dimention;

    FILE *fp = fopen("first.ppm", "wb"); /* b - binary mode */
    (void) fprintf(fp, "P6\n%d %d\n255\n", dimx, dimy);
    for (j = 0; j < dimy; ++j){
     for (i = 0; i < dimx; ++i){
        static unsigned char color[3];
        color[0] = Result[i][j][0];  /* red */
        color[1] = Result[i][j][1];  /* green */
        color[2] = Result[i][j][2];  /* blue */
        (void) fwrite(color, 1, 3, fp);
     }
    }
    (void) fclose(fp);
```

```
}
```

OpenMP

```
/*
Ojani Eguia
287 Term Project
Image Generator / Processor
OpenMP Version
*/
#include <stdbool.h>
#include <stdio.h>
#include <time.h>

void main(){
    int printing = 0;
    int dimention = 256;
    int nthreads = 5;
    int i = 0,j = 0; // Indices
    srand(time(NULL));   // should only be called once
    int generation = 4;
    /*
    Instructions for Generate:

    For First Input:
    Even numbers start with awesome base,
    Odd numbers generate random fields.

    For Second Input:
    0/1 is no change
    2/3 is Shift to red
    4/5 is Shift to green
    6/7 is Shift to blue
    anything else is random


    NOTE: Cool array is thanks to
http://rosettacode.org/wiki/Bitmap/Write_a_PPM_file#C

    */

    int Input1[dimention][dimention][3];
    int Input2[dimention][dimention][3];
    int Result[dimention][dimention][3];


    for (i = 0; i < dimention; i++){//FIRST INPUT ARRAY
```

```
      for (j = 0; j < dimention; j++){
           int a, b, c;
           if((generation % 2) == 0){//cool array
                a = i % 256;    /* red */
                b = j % 256;    /* green */
                c = (i * j) % 256;    /* blue */
           }
           else{//random
                a = rand()%0x100;
                b = rand()%0x100;
                c = rand()%0x100;
           }
           Input1[i][j][0] = a;
           Input1[i][j][1] = b;
           Input1[i][j][2] = c;
      }
    }

    for (i = 0; i < dimention; i++){//SECOND INPUT ARRAY
      for (j = 0; j < dimention; j++){
           int a, b, c;
           if((generation == 1) || (generation == 0)){//no
change
                a = 0xff;
                b = 0xff;
                c = 0xff;
           }
           else if((generation == 2) || (generation == 3)){//red
                a = 0xff;
                b = 0x80;
                c = 0x80;
           }
           else if((generation == 4) || (generation ==
5)){//green
                a = 0x80;
                b = 0xff;
                c = 0x80;
           }
           else if((generation == 6) || (generation ==
7)){//blue
                a = 0x80;
                b = 0x80;
                c = 0xff;
           }
           else{//random
```

```
                a = rand()%0x100;
                b = rand()%0x100;
                c = rand()%0x100;
           }
           Input2[i][j][0] = a;
           Input2[i][j][1] = b;
           Input2[i][j][2] = c;
      }
    }

    if (printing == 1){
      printf("Here is Input 1 before Multiplication: \n");
      int a = 0, b = 0;// Indices
      for (a = 0; a < dimention; a++){
           for (b = 0; b < dimention; b++){
                printf("%.2x ,", Input1[a][b][0]);
                printf("%.2x ,", Input1[a][b][1]);
                printf("%.2x \t", Input1[a][b][2]);
           }
           printf("\n");
      }
    }

    if (printing == 1){
      printf("Here is Input 2 before Multiplication: \n");
      int a = 0, b = 0;// Indices
      for (a = 0; a < dimention; a++){
           for (b = 0; b < dimention; b++){
                printf("%.2x ,", Input2[a][b][0]);
                printf("%.2x ,", Input2[a][b][1]);
                printf("%.2x \t", Input2[a][b][2]);
           }
           printf("\n");
      }
    }

//gonna overlap the files now

    #pragma omp parallel num_threads(nthreads)
    {
    printf("this is thread %d \n", omp_get_thread_num());


    int d = 0,e = 0, f = 0; // Indices
    #pragma omp for
    for (d = 0; d < dimention; d++){
      for (e = 0; e < dimention; e++){
           for(f = 0; f < 3; f++){//Algorithm is based on
overlapping of colors: how much Input2 "affects' input a based
on intensity of that component in input2
                double intA = Input2[d][e][f];
                double percentage = (intA/0x100);
                int newValue = (Input1[d][e][f] * percentage);
                Result[d][e][f] = newValue+1;
           }
      }
    }
    }

    //Checking if there were leftover rows: if yes, this rank
will do them
    int leftovers = dimention % nthreads;
    if (leftovers != 0){
      int c = (dimention - leftovers),d = 0; // Indices
      for (c = (dimention - leftovers); c < dimention; c++){
           for (d = 0; d < dimention; d++){
//For now, simply turns the pixels white instead of being
garbage
                int a, b, c;
                a = 0xff;
                b = 0xff;
                c = 0xff;
                Result[c][d][0] = a;
                Result[c][d][1] = b;
                Result[c][d][2] = c;
           }
      }
    printf("%d rows were left behind, but Thread 0 cleaned them
up so it's okay :v \n", leftovers);
    }


    if (printing == 1){
      printf("Here is Result after Multiplication: \n");
      int a = 0, b = 0;// Indices
      for (a = 0; a < dimention; a++){
           for (b = 0; b < dimention; b++){
                printf("%.2x ,", Result[a][b][0]);
                printf("%.2x ,", Result[a][b][1]);
                printf("%.2x \t", Result[a][b][2]);
           }
           printf("\n");
      }
    }
    //The below is adapted from 
http://rosettacode.org/wiki/Bitmap/Write_a_PPM_file#C
    const int dimx = dimention, dimy = dimention;

    FILE *fp = fopen("first.ppm", "wb"); /* b - binary mode */
    (void) fprintf(fp, "P6\n%d %d\n255\n", dimx, dimy);
    for (j = 0; j < dimy; ++j){
      for (i = 0; i < dimx; ++i){
           static unsigned char color[3];
           color[0] = Result[i][j][0];  /* red */
           color[1] = Result[i][j][1];  /* green */
           color[2] = Result[i][j][2];  /* blue */
           (void) fwrite(color, 1, 3, fp);
      }
    }
    (void) fclose(fp);
}
```

MPI

```
/*
Ojani Eguia
287 Term Project
Image Generator / Processor
MPI Version
*/
#include <stdbool.h>
#include <stdio.h>
#include <time.h>
#include <mpi.h>   /* PROVIDES THE BASIC MPI DEFINITION AND 
TYPES */
//#include <string.h>
//#define SEED 35791247

int main(int argc, char **argv){

    int my_rank;
    int size;

    // to get node name, from 
http://www.cisl.ucar.edu/docs/lightning/examples/mpi.jsp
    int name_len;
    char nodename[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc, &argv); /*START MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); /*DETERMINE RANK OF
THIS PROCESSOR*/
    MPI_Comm_size(MPI_COMM_WORLD, &size); /*DETERMINE TOTAL
NUMBER OF PROCESSORS*/
    MPI_Get_processor_name(nodename, &name_len);

//    MPI Setup: Since this version generates values from
scratch, Threads have
//    Been set up to only send results back to main thread,
since there is no
//    need to send starting data.
//
//    -All threads (except main thread) generate numbers and
calculate answers
//    -these threads then send back calculated data back to main
thread
//    -main thread gathers data and puts it into the new ppm
file.

    int printing = 0;
    int dimention = 256;
    int workerThreads = size-1;
     int segment = dimention/workerThreads;
    int i = 0,j = 0; // Indices
    srand(time(NULL));   // should only be called once

    int generation = 2;
    /*
    Instructions for Generate:

    For First Input:
    Even numbers start with awesome base,
    Odd numbers generate random fields.

    For Second Input:
    0/1 is no change
    2/3 is Shift to red
    4/5 is Shift to green
    6/7 is Shift to blue
    anything else is random
    NOTE: Cool array is thanks to
http://rosettacode.org/wiki/Bitmap/Write_a_PPM_file#C

    */

    if(my_rank != 0){// for non master threads
      int Input1[segment][dimention][3];
      int Input2[segment][dimention][3];
      int SmallOut[segment][dimention][3];
      for (i = 0; i < segment; i++){
           for (j = 0; j < dimention; j++){
                int a, b, c;
                if((generation % 2) == 0){//cool array
                     a = (i + (segment*(my_rank-1))) % 256;
/* red */
                     b = j % 256;                       /* green
*/
                     c = ((i+ ((segment*(my_rank-1)))) * j) %
256;    /* blue */
                }
                else{//random
                     a = rand()%0x100;
                     b = rand()%0x100;
                     c = rand()%0x100;
                }
                Input1[i][j][0] = a;
                Input1[i][j][1] = b;
                Input1[i][j][2] = c;
           }
      }

      for (i = 0; i < segment; i++){
           for (j = 0; j < dimention; j++){
                int a, b, c;
                if((generation == 1) || (generation == 0)){//no
change
                     a = 0xff;
                     b = 0xff;
                     c = 0xff;
                }
                else if((generation == 2) || (generation ==
3)){//red
                     a = 0xff;
                     b = 0x80;
                     c = 0x80;
                }
                else if((generation == 4) || (generation ==
5)){//green
                     a = 0x80;
                     b = 0xff;
                     c = 0x80;
                }
                else if((generation == 6) || (generation ==
7)){//blue
                     a = 0x80;
                     b = 0x80;
                     c = 0xff;
                }
                else{//random
                     a = rand()%0x100;
                     b = rand()%0x100;
                     c = rand()%0x100;
                }
                Input2[i][j][0] = a;
                Input2[i][j][1] = b;
                Input2[i][j][2] = c;
           }
      }

      if (printing >= 3){
           sleep(2*my_rank); //easier to read print output this
way
           printf("Here is Input 1 before Multiplication: \n");
           int a = 0, b = 0;// Indices
           for (a = 0; a < segment; a++){
                for (b = 0; b < dimention; b++){
                     printf("%.2x ,", Input1[a][b][0]);
                     printf("%.2x ,", Input1[a][b][1]);
                     printf("%.2x \t", Input1[a][b][2]);
                }
                printf("\n");
           }
      }

      if (printing >= 3){
           sleep(2*my_rank); //easier to read print output this
way
           printf("Here is Input 2 before Multiplication: \n");
           int a = 0, b = 0;// Indices
           for (a = 0; a < segment; a++){
                for (b = 0; b < dimention; b++){
                     printf("%.2x ,", Input2[a][b][0]);
                     printf("%.2x ,", Input2[a][b][1]);
                     printf("%.2x \t", Input2[a][b][2]);
                }
                printf("\n");
           }
      }

      //gonna overlap the files now

      int d = 0,e = 0, f = 0; // Indices
      for (d = 0; d < segment; d++){
           for (e = 0; e < dimention; e++){
                for(f = 0; f < 3; f++){//Algorithm is based on
overlapping of colors: how much Input2 "affects' input a based
on intensity of that component in input2
                     double intA = Input2[d][e][f];
                     double percentage = (intA/0x100);
                     int newValue = (Input1[d][e][f] *
percentage);
                     SmallOut[d][e][f] = newValue+1;
                }
           }
      }

      if (printing >= 2){
           sleep(2*my_rank); //easier to read print output this
way
           printf("Rank %d of %d on %s Multiplied and got the
following:\n", my_rank, size, nodename);
           int a = 0, b = 0;// Indices
           for (a = 0; a < segment; a++){
                for (b = 0; b < dimention; b++){
                     printf("%.2x ,", SmallOut[a][b][0]);
                     printf("%.2x ,", SmallOut[a][b][1]);
                     printf("%.2x \t", SmallOut[a][b][2]);
                }
                printf("\n");
           }
      }

      //Sending out results of multiplication
      MPI_Send(&SmallOut[0][0][0], segment*dimention*3, MPI_INT,
0, 1, MPI_COMM_WORLD);
```

```
      printf("Rank %d of %d on %s Sent Results back to Rank 
0\n", my_rank, size, nodename);
      //Finished Sending

    }//end worker only segment;  begins master only segment

    if(my_rank == 0){//Master work only
      int Result[dimention][dimention][3];
      //Going to receive info from other ranks now
      int k = 1;
      for (k = 1; k <= workerThreads; k++){
           int g = k-1;
           int f = g*segment;
           MPI_Recv(&Result[f][0][0], segment*dimention*3, 
MPI_INT, k, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      }

      //Checking if there were leftover rows: if yes, this rank 
will do them
      int leftovers = dimention % workerThreads;
      if (leftovers != 0){
           int c = (dimention - leftovers),d = 0; // Indices
           for (c = (dimention - leftovers); c < dimention; 
c++){
                for (d = 0; d < dimention; d++){
//For now, simply turns the pixels white instead of being 
garbage
                     int a, b, c;
                     a = 0xff;
                     b = 0xff;
                     c = 0xff;
                     Result[c][d][0] = a;
                     Result[c][d][1] = b;
                     Result[c][d][2] = c;
                }
           }
      printf("%d rows were left behind, but Rank 0 cleaned them 
up so it's okay :v \n", leftovers);
      }
      //Finished Checking now

      //Finished receiving info
      //sleep(2*my_rank); //easier to read print output this way
      if (printing >= 1){
           printf("Here is Master Result: \n");
           int a = 0, b = 0;// Indices
           for (a = 0; a < dimention; a++){
                for (b = 0; b < dimention; b++){
                     printf("%.2x ,", Result[a][b][0]);
                     printf("%.2x ,", Result[a][b][1]);
                     printf("%.2x \t", Result[a][b][2]);
                }
           printf("\n");
           }
      }


      //The below is adapted from
http://rosettacode.org/wiki/Bitmap/Write_a_PPM_file#C
      const int dimx = dimention, dimy = dimention;

      FILE *fp = fopen("first.ppm", "wb"); /* b - binary mode */
      (void) fprintf(fp, "P6\n%d %d\n255\n", dimx, dimy);
      for (j = 0; j < dimy; ++j){
           for (i = 0; i < dimx; ++i){
                static unsigned char color[3];
                color[0] = Result[i][j][0];  /* red */
                color[1] = Result[i][j][1];  /* green */
                color[2] = Result[i][j][2];  /* blue */
                (void) fwrite(color, 1, 3, fp);
           }
      }
      (void) fclose(fp);
    }//end of master only commands

    MPI_Finalize();  /* EXIT MPI */
}
```