4F03 Assignment 1
Parallel Image Processing

Jamie Counsell
Rakesh Mistry

Thursday, Jan 28, 2015

4F03 Assignment 1

This project is also available through GitHub. The URL will be private until the submission deadline.

Authors

Name	Student Number	Email	Website
Rakesh Mistry	1221428	mistryrn@mcmaster.ca	rakeshmistry.ca
Jamie Counsell	1054209	mistryrn@mcmaster.ca	jamiecounsell.me

Description

This C program provides two methods of image filtering to reduce noise in `ppm` formatted images. The two filter options are:

- Mean Filter
- Median Filter

Dependencies

- `openmpi`

Installation

- Install the dependencies
- Clone the repository

Operation

To run the program, first run make:

```
$ make clean
... [truncated]
$ make
mpicc -g -O2 -Wall -Wno-unused-variable   -c -o main.o main.c
mpicc -g -O2 -Wall -Wno-unused-variable   -c -o readwriteppm.o readwriteppm.c
mpicc -g -O2 -Wall -Wno-unused-variable   -c -o processimage.o processimage.c
mpicc -o ppmf main.o readwriteppm.o processimage.o
```

Then run the program as follows:

```
$ mpirun -np p ./ppmf input.ppm output.ppm N F
```

where:

• **p** is the number of processes
• **input.ppm** is the name of the input file
• **output.ppm** is the name of the output file
• **N** specifies the size of the window, that is N × N window, where N is an odd integer ≥ 3
• **F** is the type of filter. `A` for *mean filter*, and `M` for *median filter*

Work Breakdown

The work was broken down as follows:

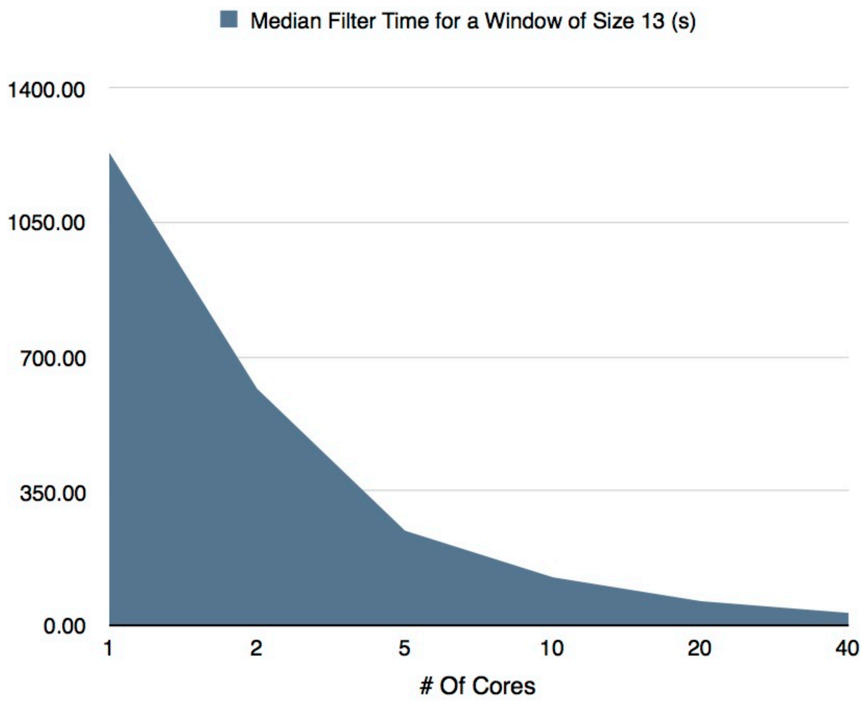
- Jamie wrote the mean filter
- Rakesh wrote the median filter
- Both authors wrote the rest of `main.c` and `processimage.c` together, in person

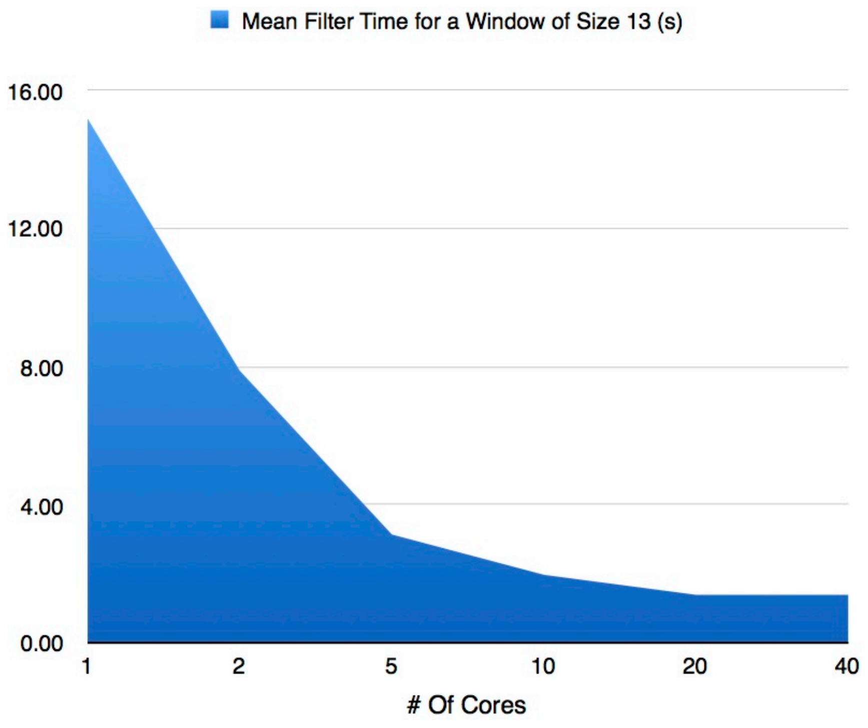
Notes

- Each process allocates and receives only the chunk of the image that it requires. Process 0 allocates and stores the entire source image, as it needs it to write back at the end of the filtering. Each other process uses only the pixels required to compute its ration of the original problem.
- This current implementation will suffer if an image loaded is larger than available memory. It is assumed that this is not the case for this assignment.

Results

The following are results from McMaster's MPIHost server.


Median Filter Time for a Window of Size 13 (s)
1400.00
1050.00
700.00
350.00
0.00
1
2
5
10
20
40
# Of Cores


Mean Filter Time for a Window of Size 13 (s)
16.00
12.00
8.00
4.00
0.00
1
2
5
10
20
40
# Of Cores

main.c

```c
#include "a1.h"
#include "mpi.h"
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

int main(int argc, char** argv)
{
  // Initialize variables
  RGB *image = NULL;
  int width = 0, height = 0, max = 0;
  int my_rank, p, i, start, size;
  clock_t begin = 0;
  double time_spent;
  int window = atoi(argv[3]);

  // Initialize MPI
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &p);

  if (my_rank == 0) {
    // Rank 0 read image from disk
    image = readPPM(argv[1], &width, &height, &max);
  }

  // Broadcast width and height to other processes
  MPI_Bcast(&width, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&height, 1, MPI_INT, 0, MPI_COMM_WORLD);

  if (my_rank == 0) {
    begin = clock();
    for (i=1; i<p; i++) {
      start = getStart(i, width, height, window, p);
      size = getSize(i, width, height, window, p);
      MPI_Send(image +start, size*sizeof(RGB), MPI_CHAR, i, 0, MPI_COMM_WORLD);
    }
  } else {
    start = getStart(my_rank, width, height, window, p);
    size = getSize(my_rank, width, height, window, p);
    image = (RGB*)malloc(size*sizeof(RGB));
    assert(image);
    MPI_Recv(image, size*sizeof(RGB), MPI_CHAR, 0, 0, MPI_COMM_WORLD, &status);
  }
  processImage(width, height, image, argc, argv);

  if (my_rank == 0) {
    time_spent = (double)(clock() - begin) / CLOCKS_PER_SEC;
    writePPM(argv[2], width, height, max, image);
    printf("done. processImage took %fs\n", time_spent);
  }

  // Cleanup
  free(image);
  MPI_Finalize();

  return(0);
}
```

processImage.c

```c
#include "a1.h"
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

void processImage(int width, int height, RGB *image, int argc, char** argv)
{
  // Initialize variables
  int i, my_rank, p, n, start, my_size, process_start, process_size;
  double a, b;

  // Initialize MPI
  MPI_Status status;
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  int source = 0, dest = 0, tag = 0;

  // Get total size of image
  int size = width*height;

  // Get number of pixels per process
  double h = size/p;

  // Get parameters from command line
  int window = atoi(argv[3]);
  char *filter = argv[4];

  // Print header
  if (my_rank == 0) {
    printf("Window size:     %dx%d\nFilter Type:     %s\n", window, window, filter);
    printf("Pixels in image: %d\nPixels/process:  %.0f\nTotal processes: %d\n", size, h, p);
    printf("----------------------\n");
  }

  /* create a type for struct RGB */
  const int nitems=3;
  int          blocklengths[3] = {sizeof(char),sizeof(char),sizeof(char)};
  MPI_Datatype types[3] = {MPI_CHAR, MPI_CHAR, MPI_CHAR};
  MPI_Datatype mpi_rgb_type;
  MPI_Aint     offsets[3];
  offsets[0] = offsetof(RGB, r);
  offsets[1] = offsetof(RGB, g);
  offsets[2] = offsetof(RGB, b);
  MPI_Type_create_struct(nitems, blocklengths, offsets, types, &mpi_rgb_type);
  MPI_Type_commit(&mpi_rgb_type);
  /* ------ */

  MPI_Barrier(MPI_COMM_WORLD);
  // Determine lower and upper values for pixel range
  start = getStart(my_rank, width, height, window, p);
  my_size = getSize(my_rank, width, height, window, p);
  if (my_rank == 0) {
    process_start = 0;
    process_size = my_size - (width * (window-1)/2);
  } else if (my_rank >= p-1) {
    process_start = (width * (window-1)/2);
    process_size = my_size;
  } else {
    process_start = (width * (window-1)/2);
    process_size = my_size;
  }

  // Run filtering on image
  printf("Process %d crunching pixels %d - %d...\n", my_rank, start, my_size);

  if ( *filter == 'A') { // mean filter
    meanFilter(my_size, width, image, window, process_start, process_size, my_rank);

  } else if ( *filter == 'M' ) { // median filter
    medianFilter(my_size, height, image, window, process_start, process_size, my_rank);

  } else { // Invalid input for filter type
    if (my_rank == 0){
      printf("Error: Invalid filter specified. Please use either 'A' for Mean, or 'M' for
Median.\n");
    }
  }
  if (my_rank != 0) {
    // Send this rank's image chunk to process zero
    MPI_Send(image + process_start, process_size, mpi_rgb_type, dest, tag, MPI_COMM_WORLD);
  } else {
    for (i=1; i < p; i ++) {
      start = size/p*i;
      process_size = getSize(i, width, height, window, p);
      // Wait to receive data from other processes
      MPI_Recv(image + start, process_size, mpi_rgb_type, i, tag, MPI_COMM_WORLD, &status);
    }
  }
}
```

median.c

```c
#include "a1.h"
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stdio.h>

float median(int n, int x[]);
float median2(int n, int arr[]);

void swap(int* a, int* b);

void medianFilter(int size, int width, RGB *image, int window, int start, int end, int rank){
  int thing = (window - 1)/2;
  int windowsq = window*window;
  int topleft, current;
  int pc, i, j, cpypx;

  // Storage for the R, G, and B values in the window
  int rvalues[windowsq];
  int gvalues[windowsq];
  int bvalues[windowsq];

  RGB *unmodified = (RGB*)malloc(size*sizeof(RGB));
  RGB *current_pixel;
  RGB *copydestpixel;
  RGB *copysrcpixel;
  RGB *pixel;


  // Deep copy unmodified <- img
  for (i=0; i < size; i++) {
    copydestpixel = unmodified + i;
    copysrcpixel = image + i;

    copydestpixel->r = copysrcpixel->r;
    copydestpixel->g = copysrcpixel->g;
    copydestpixel->b = copysrcpixel->b;
  }

  // For each pixel in this worker's quota
  for (pc = start; pc < end; pc ++) {
    // Current pixel of interest
    pixel = image + pc;

    // Pixel at top left of window
    topleft = pc - (thing * width) - thing;

    // Reset R, G, and B values to 0's
    memset(rvalues, 0, sizeof(int)*windowsq);
    memset(gvalues, 0, sizeof(int)*windowsq);
    memset(bvalues, 0, sizeof(int)*windowsq);

    int count = 1;

    // For each row in window
    for (i=0; i < window; i ++) {
      // For each column in window
      for (j=0; j < window; j ++) {
        // Determine the pixel we're looking at
        current = topleft + i * width - thing + j + 1;
        // If current pixel is outside range of window and image, skip it
        if (current < 0 || current > size -1 || (current % width) > (pc % width) + thing) {
          // Do nothing

        // If current pixel is in range of window and image
        } else {
          current_pixel = unmodified + current;
          rvalues[count - 1] = current_pixel->r;
          gvalues[count - 1] = current_pixel->g;
          bvalues[count - 1] = current_pixel->b;

          count = count + 1;
        }
      }
    }

    pixel->r = median2(count, rvalues);
    pixel->g = median2(count, gvalues);
    pixel->b = median2(count, bvalues);
  }
}

// Code courtesy of https://en.wikiversity.org/wiki/C_Source_Code/Find_the_median_and_mean
// Great for testing, but slow.
float median(int n, int x[]) {
  float temp;
  int i, j;
  // the following two loops sort the array x in ascending order
  for (i = 0; i < n-1; i++) {
    for (j = i+1; j < n; j++) {
      if (x[j] < x[i]) {
        // swap elements
        temp = x[i];
        x[i] = x[j];
        x[j] = temp;
      }
    }
  }

  if (n%2 == 0) {
    // if there is an even number of elements, return the mean of the two elements in the
middle
    return ((x[n/2] + x[n/2 - 1]) / 2.0);
  } else {
    // else return the element in the middle
    return x[n/2];
  }
}

void swap(int* a, int* b){
    int t = *a;
    *a = *b;
    *b = t;
}

// Code courtesy of http://stackoverflow.com/questions/1961173/median-function-in-c-math-
library
// Much faster than median().
float median2(int n, int arr[])
{
    int low, high ;
```

median.c

```c
    int median;
    int middle, ll, hh;
    low = 0 ; high = n-1 ; median = (low + high) / 2;
    for (;;) {
        if (high <= low) /* One element only */
            return arr[median] ;
        if (high == low + 1) { /* Two elements only */
            if (arr[low] > arr[high])
                swap(arr+low, arr+high) ;
                return arr[median] ;
        }
        /* Find median of low, middle and high items; swap into position low */
        middle = (low + high) / 2;
        if (arr[middle] > arr[high]){
            swap(arr + middle, arr +high);
        }
        if (arr[low] > arr[high]){
          swap(arr+low, arr+high);
        }
        if (arr[middle] > arr[low]){
            swap(arr+middle, arr+low);
        }

        /* Swap low item (now in position middle) into position (low+1) */
        swap(arr+middle, arr+low+1) ;
        /* Nibble from each end towards middle, swapping items when stuck */
        ll = low + 1;
        hh = high;

        for (;;) {
            do ll++; while (arr[low] > arr[ll]) ;
            do hh--; while (arr[hh] > arr[low]) ;
            if (hh < ll)
                break;
            swap(arr+ll, arr+hh) ;
        }

        /* Swap middle item (in position low) back into correct position */
        swap(arr+low, arr+hh) ;
        /* Re-set active partition */
        if (hh <= median)
            low = ll;
        if (hh >= median)
            high = hh - 1;
    }
    return arr[median];
}
```

mean.c

```c
#include "a1.h"
#include <stdlib.h>

void meanFilter(int size, int width, RGB *image, int window, int start, int end, int rank){
  int thing = (window - 1)/2;
  int topleft, current;
  int pc, i, j, cpypx;
  double sum[3];
  RGB *unmodified = (RGB*)malloc(size*sizeof(RGB));
  RGB *current_pixel;
  RGB *copydestpixel;
  RGB *copysrcpixel;
  RGB *pixel;


  // Deep copy to store unmodified values
  for (i=0; i < size; i++) {
    copydestpixel = unmodified + i;
    copysrcpixel = image + i;
    copydestpixel->r = copysrcpixel->r; // SEG FAULT HERE
    copydestpixel->g = copysrcpixel->g;
    copydestpixel->b = copysrcpixel->b;
  }

  // For each pixel in this worker's quota
  for (pc = start; pc < end; pc ++) {
    // Current pixel of interest
    pixel = image + pc;

    // Pixel at top left of window
    topleft = pc - (thing * width) - thing;

    sum[0] = 0; // Red values
    sum[1] = 0; // Green values
    sum[2] = 0; // Blue values
    int count = 0;
    //printf("sum: %0.1f   count: %d\n", sum, count);

    // For each row in window
    for (i=0; i < window; i ++) {
      // For each column in window
      for (j=0; j < window; j ++) {
        // Determine the pixel we're looking at
        current = topleft + i * width - thing + j + 1;

        // If current pixel is outside range of window and image, skip it
        if (current < 0 || current > size -1 || (current % width) > (pc % width) + thing) {
          // Do nothing

        // If current pixel is in range of window and image
        } else {
          current_pixel = unmodified + current;
          sum[0] = sum[0] + (current_pixel->r);
          sum[1] = sum[1] + (current_pixel->g);
          sum[2] = sum[2] + (current_pixel->b);

          count = count + 1;
        }
      }
    }
```

mean.c

```
    //printf("pc: %d    total:     %f\n", pc, sum);
    // new pixel rgb value is average of ^
    pixel->r = sum[0]/count;
    pixel->g = sum[1]/count;
    pixel->b = sum[2]/count;
  }
}
```

helpers.c

```c
// Get start position of chunk for process
int getStart(int rank, int width, int height, int window, int p) {
    int total = width * height;
    int ration = total/p;

    int start = ration * rank - ((window - 1)/2 * width);

    if (start < 0) {
        start = 0;
    }
    return start;
}

// Get size of chunk for process
int getSize(int rank, int width, int height, int window, int p) {
    int total = width * height;
    int ration = total/p;
    int size = ration + (width * (window-1));
    int start = getStart(rank, width, height, window, p);
    if (rank >= p-1 || size > total) {
        size = total - start;
    } else if (start == 0) {
        size = size - ((window-1)/2 * width);
    }
    return size;
}
```

readwriteppm.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "mpi.h"
#include "a1.h"

void writePPM(char* file, int width, int height, int max, const RGB *image)
{
  int i;
  printf("Writing result...  ");

  // open file for writing
  FILE *fd;
  fd = fopen(file, "w");

  // output the header
  fprintf(fd, "P3\n");
  fprintf(fd, "%d %d\n%d\n", width, height, max);
  // write the image
  for (i = 0; i < height*width; i++)
  {
    const RGB *p = image+i;
    fprintf(fd, "%d %d %d ", p->r, p->g, p->b);
  }
}


RGB * readPPM(char* file, int* width, int* height, int* max)
{
  /* Read a PPM P3 image from a file into a buffer.  Return the
     buffer, width, height, and the max value in the image. */

  FILE *fd;
  char c;
  int i,n, my_rank;

  // initialize MPI
  MPI_Status status;
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  int source, total, dest = 0, tag = 0;

  char b[100];
  int red, green, blue;

  // check if P3 ppm format
  fd = fopen(file, "r");
  n = fscanf(fd,"%[^\n] ",b);
  if(b[0]!='P'|| b[1] != '3')
    {
      if (my_rank == 0) {
        printf("%s is not a PPM file!\n",file);
      }
      exit(0);
    }
  if (my_rank == 0) {
    // printf("%s is a PPM file\n", file);
  }
  n = fscanf(fd, "%c",&c);
  while(c == '#')
    {
      n = fscanf(fd, "%[^\n] ", b);
```

readwriteppm.c

```
      if (my_rank == 0) {
        printf("%s\n",b);
      }
      n = fscanf(fd, "%c",&c);
    }
  ungetc(c,fd);
  n = fscanf(fd, "%d %d %d", width, height, max);
  assert(n==3);

  // size of image
  int size = *width*(*height);
  RGB *image = (RGB*)malloc(size*sizeof(RGB));
  assert(image);

  for(i=0; i < size; i++)
    {
      n =  fscanf(fd,"%d %d %d",&red, &green, &blue);
      assert(n==3);
      image[i].r = red;
      image[i].g = green;
      image[i].b = blue;
    }

  return image;
}
```

a1.h

```c
#ifndef INCLUDED_A1_H
#define INCLUDED_A1_H

typedef struct {
  unsigned char r,g,b;
} RGB;

RGB * readPPM(char* file, int* width, int* height, int* max);
void writePPM(char* file, int width, int height, int max, const RGB *image);

void meanFilter(int size, int width, RGB *image, int window, int start, int end, int rank);
void medianFilter(int size, int width, RGB *image, int window, int start, int end, int rank);

int getSize(int rank, int width, int height, int window, int p);
int getStart(int rank, int width, int height, int window, int p);

void processImage(int width, int height, RGB *image, int argc, char** argv);

#endif
```

makefile

```
CFLAGS=-g -O2 -Wall -Wno-unused-variable
CC=mpicc

PROGRAM_NAME= ppmf
OBJS = main.o readwriteppm.o processimage.o mean.o median.o helpers.o

$(PROGRAM_NAME): $(OBJS)
	$(CC) -o $@ $?

clean:
	rm  *.o $(PROGRAM_NAME) *~
```