6.338 Final Project:
Digital Image Processing in Parallel

Cory Zue

Introduction

Image processing is an important topic with a variety of both commercial and recreational applications. Adobe Photoshop software (or any similar photo editing tool) offers a variety of effects that can be applied to pictures for enhancing the quality or changing the ambiance. Matlab also has some built in image processing techniques. For almost all intents and purposes these tools are all that is necessary to meet the requirements of a typical photo-editing user.

Imagine, however, a satellite taking pictures of the ground on a daily basis. These pictures could be huge, and the number of pictures goes up quickly as the satellite attempts to cover a large area of space in significant detail. Now imagine that this satellite takes pictures of this area every day, or every hour, or every minute. Clearly the volume of data produced by this satellite is huge, and a person will not be able to look at every individual image for meaning. Automatic image processing could be an invaluable tool for tracking storms, movements of troops, or deforestation over several days.

This is where performance requirements begin to come into play, and it starts to make sense to develop high-performance image processing tools. To this end, I have chosen as my project the development of a basic image processing tool that works in parallel.

Image Processing Overview

Image processing offers a wide variety of tools and abilities, but is largely all the same basic operation. We start with the idea of a digital image as a bitmap. Once we know the height and width of the image in bits (usually found in a header), then we represent each pixel as a fixed number of bits. In a black and white image, this can be done by representing each pixel as an 8-bit number from 0-255. Black is represented as 0, while white is 255. Clearly everything in the middle is grey, with lower numbers being darker, and higher ones being lighter.

To represent a color image as a bitmap, the same can be done with each of three color components (red, green and blue) being represented by an 8-bit number at each pixel. By adding these three color components we can produce every color in the spectrum (000000 is black and FFFFFF is white, with everything else in between). Thus an image is basically a matrix of color components, and so we can perform matrix operations on it!

The majority of image processing relies on the relationship between a pixel and its neighbors. One nice way of applying filters in image processing (which is also handy in signal processing) is convolution. Convolution is a discrete processing technique that involves adding up a series of products of filter components and nearby data values to determine the new value of a point. Convolution can also be done in the continuous domain via integration, but this is not necessary for image processing.

In discrete signal processing, neighbors are usually chosen in the time domain, with the new value of a point being the result of a filter applied to the points immediately preceding the point, immediately following it, and the point itself. In image processing neighbors are chosen in the spatial domain, with the obvious choices being pixels immediately above, below, and to the side of a particular pixel. The result is a filter that is generally square that moves over the entire image.

Some filters are easy to understand the effects of, while some are more difficult. One simple filter is a blurring filter, which is basically a unity-summing matrix of identical values. A 3x3 blurring matrix is a 3x3 matrix with 1/9 in each position (see Figure 1). A 5x5 would have 1/25 in each position. The result of the convolution of this matrix with an image is that each pixel's new value will be 1/9 of each of the 9 pixels forming the 3x3 square with the pixel at the center. Thus the new value is the average of the nearby values, and the image gets blurred.

$$1/9 \left\{ \begin{array}{ccc} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{array} \right\}$$

Figure 1: The 3x3 Blurring Matrix

Similar reasoning reveals that the matrices of Figure 2 have a sharpening effect on images that increases the contrast between individual pixels. In the left filter, the pixel itself gets embellished while it's immediate neighbors above, below, and to the side, are subtracted. In the right filter all neighbors are subtracted equally, and the pixel itself is embellished even more.

$$\left\{ \begin{array}{ccc} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{array} \right\} \qquad \left\{ \begin{array}{ccc} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{array} \right\}$$

Figure 2: 3x3 Sharpening Matrices

There are several other image processing filters available, and they can grow in size well beyond 3x3 or 5x5. As the size of the filter grows, the computation time increases by a factor of N^2, so parallelism becomes important with large filters on large images.

Implementation

Although Matlab offers a variety of built in convolution packages, they are not readily adaptable to parallelism, and are not supported by StarP. For this reason I chose to do my implementation in MPI using C, with some basic image reading and writing steps done in Matlab.

The first step was to convert images of arbitrary formats to byte arrays that could be loaded into MPI. Fortunately, Matlab has functionality for loading images in as matrices, and using these it was relatively straightforward to write a function that would load an image and output it as a matrix text file. I knew that the project would also require me to convert from text back to images, and this was also most easily done in Matlab. The result was four Matlab functions, img_read, img_write, img_read_color, and img_write_color, for converting between images and text. These four scripts can be found in the Appendix. The color image functions create three text files, one for each color, or read from three text files to generate the full color image.

Once the images were storable and readable from text files, it was only a matter of performing parallel convolution on the image and filter. This was done in an MPI program, ImgProcess.c, which can also be found in the Appendix. The ImgProcess program takes three input arguments, representing 3 files: the input image, the filter, and the name to write the output image to. The input image is stored in a specific format which is a single line with the dimensions of the image, followed by the image itself with each column separated by spaces and each row separated by lines. The Matlab functions img_read and img_read_color automatically generate files in the correct format.

Once the text “images” are read into the MPI program, they are stored as a matrix of doubles and divided among the processors. Each processor computes n/np rows of the output matrix, and to do this requires giving it (n/np + f) rows of the input matrix (since convolution depends on local neighbors). Other algorithms for dividing up the matrix were considered, but giving each processor complete rows seemed to be the simplest. Each processor also receives a complete copy of the filter.

When each processor has all the data it needs, convolution between the image and the filter is done in parallel, and each processor reports back the new computed values to the root node. The root node then gathers all of the new values and writes the new image to a text file, in a format that can be interpreted by the img_write Matlab function.

Experiments

I also tested four filter sizes: 3x3, 5x5, 15x15, and 25x25. Blur filters were used for all four for performance measurements, but other filters were used to show interesting image processing results. Some new images generated from the small image are shown in Figure 5. The filters used to generate these images can also be found in the Appendix.

The various filters applied to the large image are shown in Figure 5. It should be noted that because of the size of the image, the effect of the 3x3 filters is not noticeable except at very high resolution, so only the more visible filters are shown in Figure 5.

Figure 3: The Small Image

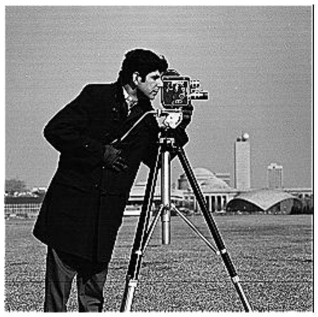
Figure 4: The Large Image

Original

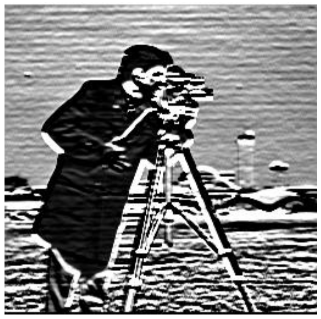
3x3 Blur

5x5 Blur

25x25 Blur



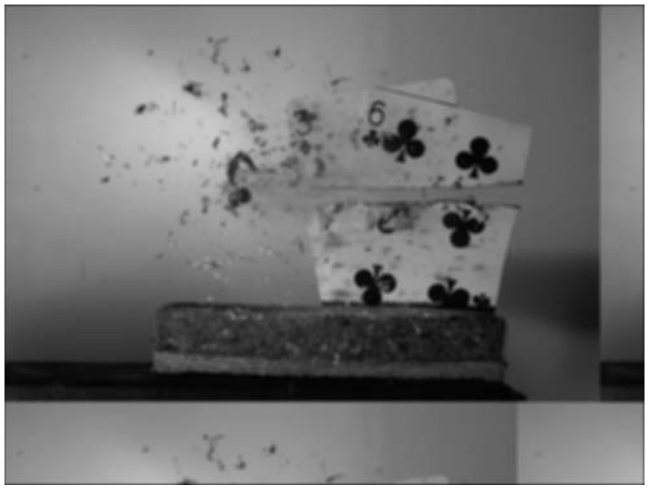
3x3 Sharpen



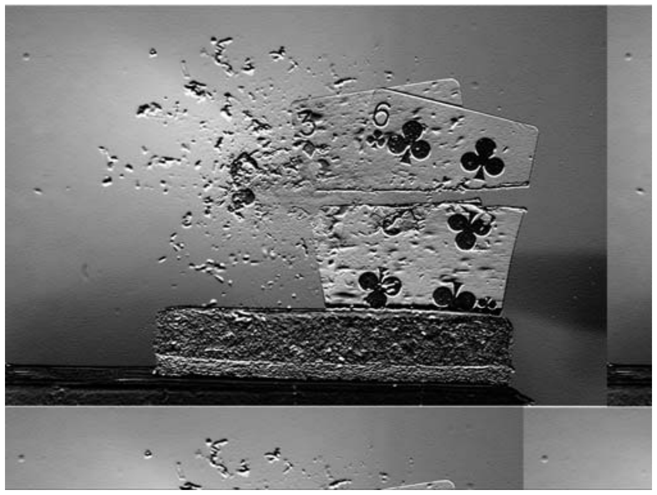
5x5 Embossing

Figure 4: Various Filters applied to the Small Image

Original Image



25x25 Blur



5x5 Emboss

Figure 5: Filters applied to the Large Image

Performance:

The time to perform each filtering operation was measured by MPI and the results were compiled in Excel. The first result was that for small filters, the overhead introduced by the communication steps outweighed the computational benefit introduced by parallelism as the number of processors was increased beyond 4. This behavior can be seen in Figure 6, which shows the performance of the 3x3 and 5x5 filter on the small image.

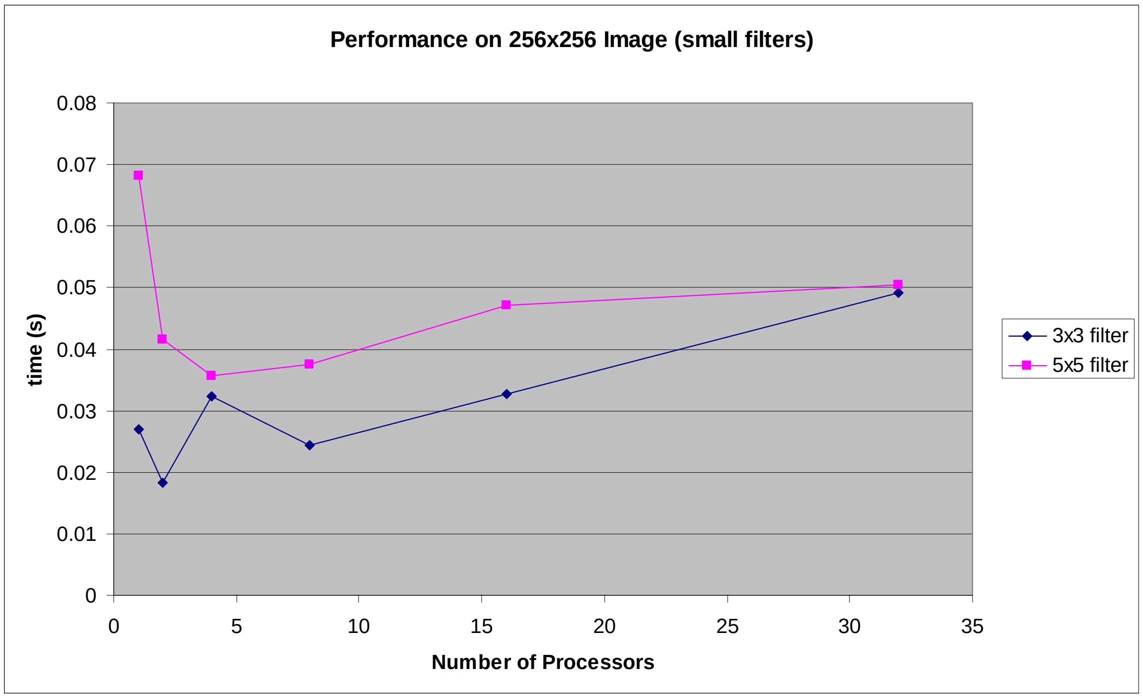


Figure 6: Performance of small filters on small image

Fortunately the performance of the larger filters was significantly better than with the small filters. For the larger filters performance improved all the way up to 32 processors (although 32 processors was a minimal performance gain over 16). These results can be seen in Figure 7. The performance of the small filters is also shown as a reference, and it can be seen that filter size plays a significant role in the time required to perform this operation. This was expected, as the size of the problem is O(m * n * f^2) for an m x n image and f x f filter.

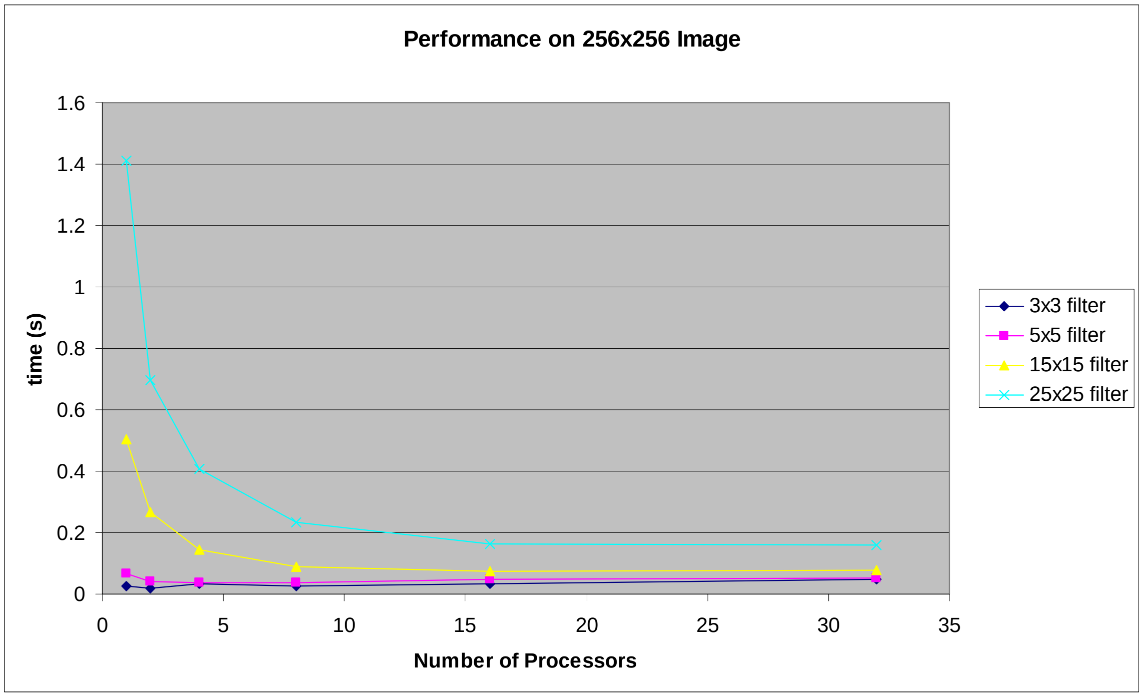


Figure 7: Performance of all Filters on Small Image

Similar performance was seen on the large image. Again, for the 3x3 and 5x5 filters, performance peaked at 4 processors, but for the larger problems performance gains increased all the way up to 32 processors. These results are summarized in Figures 8 and 9.

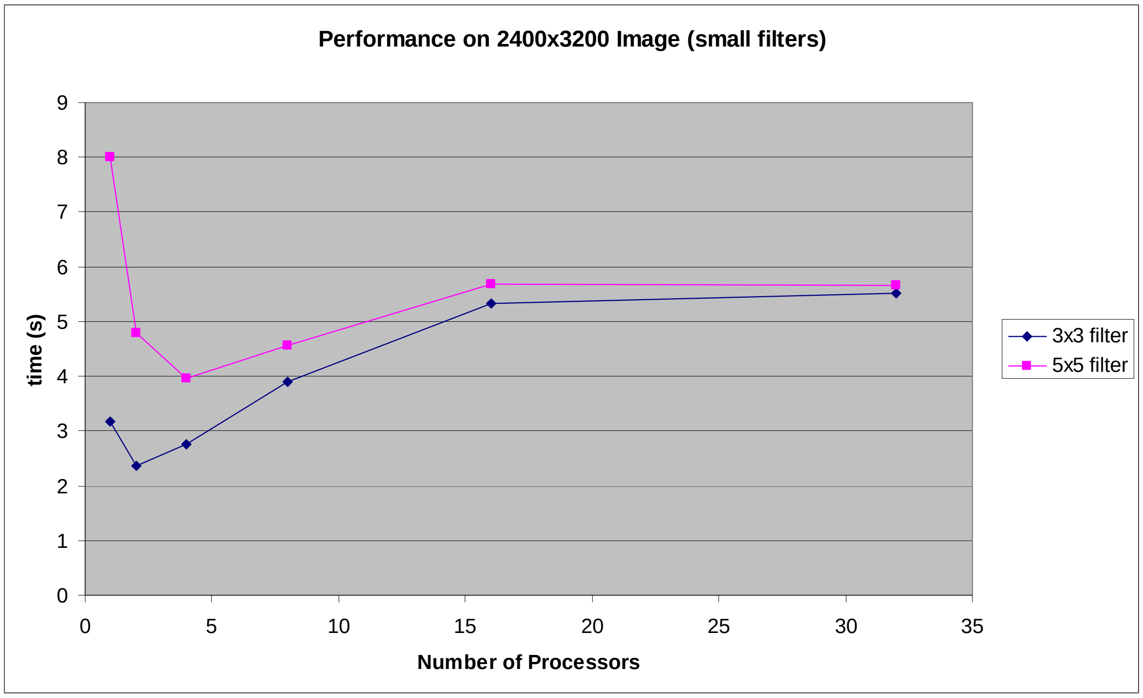


Figure 8: Performance of Small Filters on the Large Image

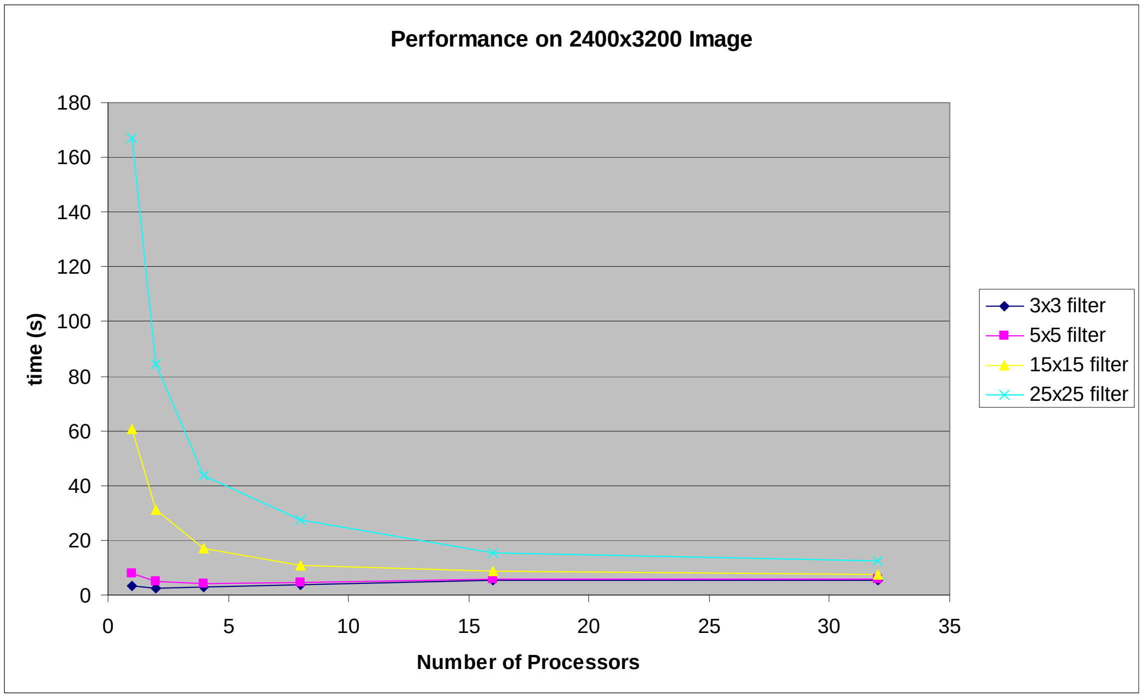


Figure 9: Performance of all Filters on Large Image

Finally we'll take a closer look at the performance gains from parallelism. Since it is known that the biggest computational problems benefit the most from parallelism (and this was seen in Figures 7 and 9), we'll focus on the large filter over the large image. To see how close to ideal this process is, we'll compare it to the linear best-case scenario. This is shown in Figures 10 and 11.

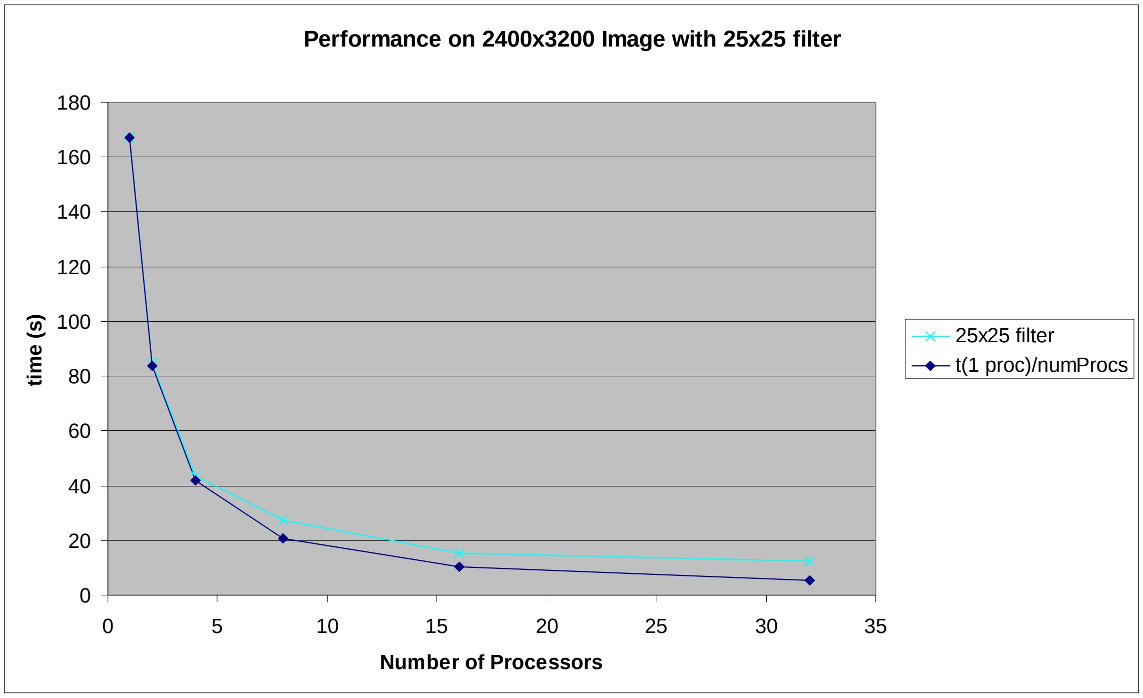


Figure 10: Run time of big filter and image versus ideal case

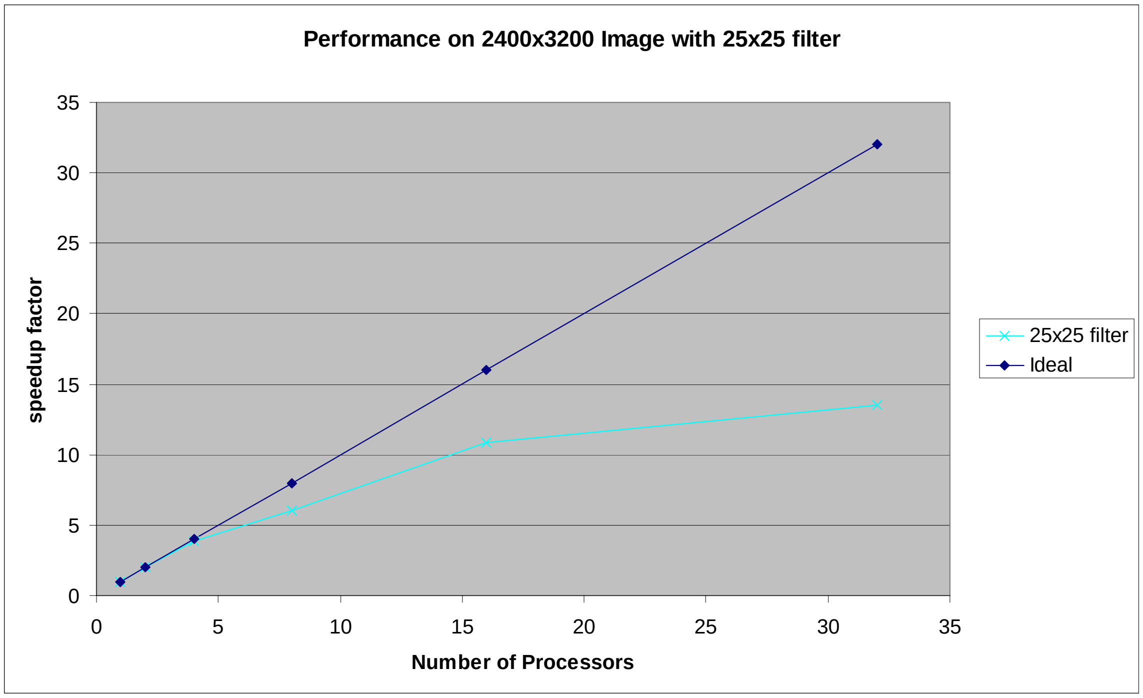


Figure 11: Speedup factor for large image and filter versus ideal case

In both Figures 10 and 11 the actual performance of the big problem is shown in light blue, while the ideal performance is shown in dark blue. Figure 10 shows the run time of the big problem versus the ideal case, which is calculated by dividing the serial runtime by the number of processors. It can be seen that the runtime for 2 and 4 processors is almost ideal, while it tapers off as the number of processors increases after that. This behavior can also be seen in Figure 11, which shows the speedup factor introduced by parallelism versus the ideal linear speedup factor. Again, it can be seen that for 2 and 4 processors, behavior is almost ideal, but the performance gains decrease as the number of processors grows beyond that.

Conclusion

It has been shown that image processing can greatly benefit from the introduction of parallelism. As the size of the problem grows, especially the filter, it makes sense to perform these operations in parallel and nearly linear performance gains can be seen for small numbers of processors. As the number of processors is increased, there are diminishing returns, but performance still improves for big problems. It appears that regardless of the size of the image, performance gains for 8 or more processors will only be seen for larger filters, and for small (3x3 or 5x5) filters, it is best to stick with 4 or less.

Appendix:

Filters used:

3x3 blur (other blurs are the same with 1/n^2 in each location)

.11111111	.11111111	.11111111
.11111111	.11111111	.11111111
.11111111	.11111111	.11111111

3x3 sharpen:

0	-1	0
-1	5	-1
0	-1	0

5x5 emboss:

-1	-1	-1	-1	-1
-2	-2	-2	-2	-2
0	0	1	0	0
2	2	2	2	2
1	1	1	1	1

ImgProcess.c

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

#ifndef MAX
#define MAX(a,b) ((a > b) ? a : b)
#endif

#define MAX_LENGTH 50000
#define MAX_PROCS 128

#define IND(m,n,i,j) ((i) * (n) + (j))
#define GET(m,n,i,j) ((m)[IND(m,n,i,j)])
#define SET(m,n,i,j,v) ((m)[IND(m,n,i,j)] = (v))

double * alloc_matrix(int m, int n)
{
  return (double *)malloc(sizeof(double) * m * n);
}

void print_matrix(FILE *fp, int m, int n, double *mat)
{
  int i, j;
  for(i = 0; i < m; i++)
    {
      for(j = 0; j < n; j++)
      {
        fprintf(fp, "%f ", GET(mat,n,i,j));
      }
      fprintf(fp, "\n");
    }
}

void read_matrix(FILE *fp, int m, int n, double *mat)
{

  char line[MAX_LENGTH];
  char *temp = NULL;
  int i, j;
  double value;

  for(i=0;i<m;i++) {
    if(fgets(line, MAX_LENGTH, fp) == NULL) break;
    temp = strtok(line, " \t\r\n");
      if (temp == NULL) {
            //printf("empty line...\n");
            fgets(line, MAX_LENGTH, fp);
            temp = strtok(line, " \t\r\n");
      }
      //printf("line is %s \n", line);

      for(j=0;j<n;j++) {
```

```
      value = atof(temp);
      SET(mat,n,i,j,value);
      if(!(temp = strtok(NULL, " \t\r\n"))) break;
    }
  }
}

/** Convolve an m x n image and f x f filter at location x, y,
returning the new value */

double convolve(double *img, double *filter, int x, int y, int m, int
n, int f_size) {
      int width = f_size / 2;
      int i, j, indx, indy;
      double val, fval, result;
      result = 0;
      for (i=0; i<f_size; i++) {
            indx = x - width + i;
            for (j=0; j<f_size; j++) {
                  indy = y - width + j;
                  if (indx < 0 || indy < 0 || indx >= m || indy >= n) {
                        val = 0;
                  } else {
                        val = GET(img, n, indx, indy);
                        fval = GET(filter, f_size, i, j);
                        result += val * fval;
                  }
            }
      }
      return result;
}

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

      char in_filename[MAX_LENGTH];
      char out_filename[MAX_LENGTH];
    char filter_filename[MAX_LENGTH];
      FILE* outFile;

      unsigned int myRank, numProcs;

      double tStart, tFinish;

      int m, n, f_size, f_size_tot, i_size_tot;
      double* img_in;
      double* img_out;
      double* filter;
    double* partial_out;

      double* tmp_result;
      double* tmp_mat;
      double* local_matrices;

      MPI_Status stat;
      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
      MPI_Comm_rank(MPI_COMM_WORLD, &myRank);


   // Fragment to process the command line parameters
   /**if(myRank == 0){
     if (argc < 3){
       fprintf(stderr,"%s inputfile power\n", argv[0]);
       MPI_Finalize();
       exit(-1);
     }
   */


      if (myRank == 0) {
          char first_line[MAX_LENGTH];
            char *tmp = NULL;




            // read in arguments
            sprintf(in_filename, "%s", argv[1]);
            sprintf(filter_filename, "%s", argv[2]);
            sprintf(out_filename, "%s", argv[3]);

            // read in image matrix
            FILE *in = fopen(in_filename, "r");
            fgets(first_line, MAX_LENGTH, in);
            tmp = strtok(first_line, " ");
            m = atoi(tmp);
            tmp = strtok(NULL, " \n\t");
            n = atoi(tmp);
            printf("input file is %s \n", in_filename);
            printf("reading a %i by %i image.\n", m, n);

            img_in = alloc_matrix(m, n);

            read_matrix(in, m, n, img_in);
            fclose(in);

            // read in filter
            FILE* filterFile = fopen(filter_filename, "r");
            fgets(first_line, MAX_LENGTH, in);
            tmp = strtok(first_line, " \n\t");
            f_size = atoi(tmp);
            if (f_size % 2 != 1) {
                  printf("Filter size of %i is invalid, must be odd\n",
f_size);

                  exit(-1);
            }
            printf("filter file is %s \n", filter_filename);
            printf("reading a %i by %i filter.\n", f_size, f_size);
            filter = alloc_matrix(f_size, f_size);
            read_matrix(filterFile, f_size, f_size, filter);
            //print_matrix(stdout, f_size, f_size, filter);
            outFile = fopen(out_filename,"w+");
      }

      tStart = MPI_Wtime();
    MPI_Bcast(&n, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
      MPI_Bcast(&m, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
      MPI_Bcast(&f_size, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
      i_size_tot = m * n;
    f_size_tot = f_size * f_size;
      if (myRank != 0) {
            img_in = alloc_matrix(m,n);
            filter = alloc_matrix(f_size, f_size);
      }
    MPI_Bcast(img_in, i_size_tot, MPI_DOUBLE, 0, MPI_COMM_WORLD);
      MPI_Bcast(filter, f_size_tot, MPI_DOUBLE, 0, MPI_COMM_WORLD);
      int nrows = m / numProcs;
      int nelts = n * nrows;

    partial_out = alloc_matrix(nrows, n);

      //print_matrix(outFile, m, n, img_out);
      //print_matrix(outFile, f_size, f_size, filter);

      int start = nrows * myRank;
      int end = nrows * (myRank + 1);
      printf("proccessor %i of %i starting at row %i and ending at row
%i of %i \n", myRank, numProcs, start, end, m);



      int i, j, xind;
      double val;
      for (i=0; i<nrows; i++) {
            for (j=0; j<n; j++) {
                  xind = i + start;
                  val = convolve(img_in, filter, xind, j, m, n,
f_size);
                  SET(partial_out, n, i, j, val);
            }
            //printf("done, row %i\n", xind);
      }
      //printf("breakpoint 1 \n");
      //print_matrix(stdout, nrows, n, partial_out);
      img_out = alloc_matrix(m,n);
    MPI_Gather(partial_out, nelts, MPI_DOUBLE, img_out, nelts,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
      tFinish = MPI_Wtime();
      double total_time = tFinish - tStart;
      if (myRank == 0) {
            //printf("breakpoint 2 \n");
            print_matrix(outFile, m, n, img_out);
            printf("procs\timg dim\tfilter\ttime\n");
            printf("%i\t%ix%i\t%ix%i\t%f\n\n", numProcs, m, n, f_size,
f_size, total_time);
            fflush(outFile);
            fclose(outFile);

      }
      printf("processor %i finished \n", myRank);
      free(img_out);
      free(img_in);
      free(filter);
      free(partial_out);

      MPI_Finalize();
      exit(0);
  }
```

************ img_read.m ************

```
function [ status ] = img_read( img_name, text_name )
%Read an image and save as a space delimited array of integers
%  Detailed explanation goes here

I = imread(img_name);
s = size(I);
dlmwrite(text_name, s, ' ');
dlmwrite(text_name, I, 'delimiter', ' ', '-append');

status = 1;
```

************ img_read_color.m ************

```
function [ status ] = img_read_color( img_name, text_name)
%Read an image and save as a space delimited array of integers
%  Detailed explanation goes here

I = imread(img_name);

I1 = I(:,:,1);
I2 = I(:,:,2);
I3 = I(:,:,3);
s = size(I1);

name1 = strcat(text_name, 'red.txt');
dlmwrite(name1, s, ' ');
dlmwrite(name1, I1, 'delimiter', ' ', '-append');

name2 = strcat(text_name, 'green.txt');
dlmwrite(name2, s, ' ');
dlmwrite(name2, I2, 'delimiter', ' ', '-append');

name3 = strcat(text_name, 'blue.txt');
dlmwrite(name3, s, ' ');
dlmwrite(name3, I3, 'delimiter', ' ', '-append');

status = 1;
```

************ img_write.m ************

```
function [ status ] = img_write( text_img, write_img )
%UNTITLED1 Summary of this function goes here
%  Detailed explanation goes here

I = dlmread(text_img);
[x, y] = size(I);
%I = I(2:x, :);
I = I(:, 1:(y-1));
I = I/256;
imwrite(I, write_img, 'jpg');
```

************ img_write_color.m ************

```
function [ status ] = img_write_color( text_img, write_img )
%UNTITLED1 Summary of this function goes here
%  Detailed explanation goes here

I1 = dlmread(strcat(text_img, 'red.txt'));
I2 = dlmread(strcat(text_img, 'green.txt'));
I3 = dlmread(strcat(text_img, 'blue.txt'));

[x, y] = size(I1);
I = zeros(x, y, 3);
I(:,:,1) = I1;
I(:,:,2) = I2;
I(:,:,3) = I3;
%I = I(2:x, :);
I = I(:, 1:(y-1), :);
I = I/256;
imwrite(I, write_img, 'jpg');
```