Lab 2
Pthread

Martin Söderén
marso329, 9009291098

12 maj 2016

1 Introduction

The problem consists of implementing and parallelizing two image filters using Pthreads. The first filter is a blur-filter that for each pixel calculates a average colour for the surrounding pixels. The other is a threshold filter that calculates the average intensity of the whole picture and makes all pixels above the average black and those below white.

2 Method

2.1 Blur filter algorithm

No synchronization is needed here because all threads write to a new destination array.

Algorithm 1 Master thread blur filter

```
procedure MASTER
    Read file
    Calculate part of image to do
    for all slave threads do
        Calculate slave threads part to work on
        Create new thread using pthread_create
    end for
    Calculate part of the image
    for all slave threads do
        Join thread using pthread_join
    end for
    Write destination to file
end procedure
```

Algorithm 2 Slave thread blur filter

```
procedure SLAVE
    Calculate part of the image
end procedure
```

2.2 Threshold filter algorithm

Algorithm 3 Master thread threshold filter

```
procedure MASTER
    Read file
    Calculate which part of the image to work on
    for all slave threads do
        Calculate slave threads part to work on
        Create new thread using pthread_create
    end for
    Calculate threshold for part of image
    Add threshold to Mutex protected variable
    Wait for all threads to call pthread_barrier_wait
    Create part of image
    for all slave threads do
        Join thread using pthread_join
    end for
    Write destination to file
end procedure
```

Algorithm 4 Slave thread threshold filter

```
procedure SLAVE
    Calculate threshold for part of image
    Add threshold to Mutex protected variable
    Wait for all threads to call pthread_barrier_wait
    Create part of image
end procedure
```

2.3 Design

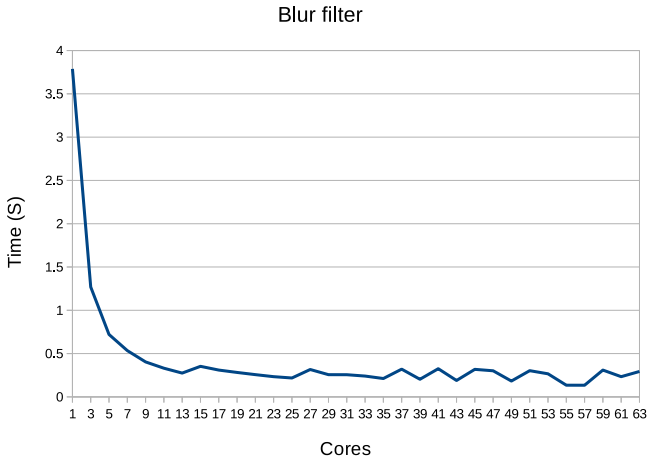
The image is split up by rows to utilize the space locality of the data and the data in both cases are accessed row wise for improved spatial locality. For example in the blur filter the pixelvalue is not calculated for each radius but for each row. If it was calculated for each radius that meant that we jump in the memory and the probability for cache misses is greater.

3 Result

The blur filter scales rather well and hits the lower time limit at around 13 cores with an improvement compared to a single core of 14 times faster. After 13 cores the calculation time remains the same. At that times the cost of starting a new thread is probably equal to the gain from that new thread.

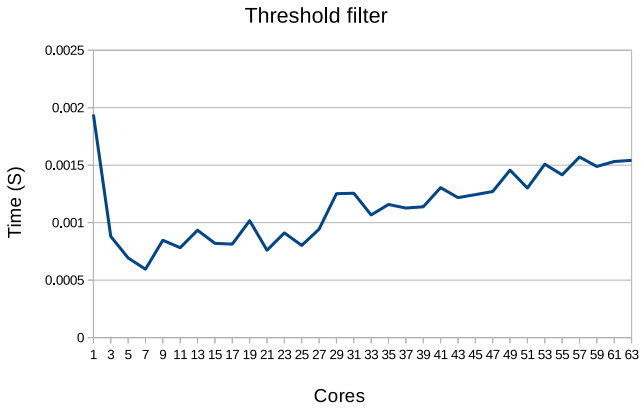
The threshold filter scales somewhat. It levels out at around 7 cores and after that it starts to increase. The calculation time with 7 cores is 3 times better than with a single core. The times are however very short in both cases. The reason for the increase of calculation time with

more cores is probably because of the mutex protected threshold variable which every core needs to access to write the local threshold.



Figur 1: Times for blur filter with radius 21

4 Result



Figur 2: Times for threshold filter

A blurthread.c

```
//standard imports
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <pthread.h>

#include "ppmio.h"
#include "gaussw.h"

//defines
#define MAX_RAD 1000
#define MAX_PIXELS (1000*1000)
#define MAX_THREADS 100
#define BILLION  1000000000L;

int min(int a, int b) {
        if (a < b) {
                return a;
        } else {
                return b;
        }
}

struct data_to_thread {
        unsigned char * src;
        unsigned char * dst;
        int xsize, ysize, start_y, end_y, radius;

} data_to_thread;

static void * blurfiltermpi(void *arguments) {
        struct data_to_thread* data = (struct data_to_thread *) arguments;
        unsigned char *src = data->src;
        unsigned char *dst = data->dst;
        int xsize = data->xsize;
        int ysize = data->ysize;
        int start_y = data->start_y;
        int end_y = data->end_y;
        int radius = data->radius;
        double w[MAX_RAD];
        get_gauss_weights(radius, w);
        double red, green, blue, n, temp_weight;
        register unsigned char* temp_pointer;
        //for each row
        for (int y = start_y; y < end_y; y++) {
                //for each column
                for (int x = 0; x < xsize; x++) {
                        red = 0.0;
                        green = 0.0;
                        blue = 0.0;
                        n = 0.0;
```

```
                        //for each sub row
                        for (int sub_y = y - radius; sub_y < y + radius; sub_y++) {
                                if (sub_y < 0 || sub_y > ysize) {
                                        continue;

                                }
                                //for each sub column
                                for (int sub_x = x - radius; sub_x < x + radius; sub_x++) {
                                        if (sub_x < 0 || sub_x > xsize) {
                                                continue;
                                        }
                                        temp_weight = w[min(abs(sub_x - x), abs(sub_y - y))
                                        temp_pointer = src + (xsize * sub_y + sub_x) * 3;
                                        red += temp_weight * (*temp_pointer);
                                        green += temp_weight * (*(temp_pointer + 1));
                                        blue += temp_weight * (*(temp_pointer + 2));
                                        n += temp_weight;
                                }

                        }
                        temp_pointer = dst + (xsize * y + x) * 3;
                        *temp_pointer = red / n;
                        *(temp_pointer + 1) = green / n;
                        *(temp_pointer + 2) = blue / n;

                }

        }
        return 0;
}

int main(int argc, char ** argv) {
        //used to know how blurry the image will be
        int radius;

        // information about the picture
        int xsize, ysize, colmax, number_of_threads;

        char* src = (char*) malloc(sizeof(unsigned char) * MAX_PIXELS * 3);
        char* dst = (char*) malloc(sizeof(unsigned char) * MAX_PIXELS * 3);

        struct timespec start, stop;
        double accum;

        if (argc != 5) {
                fprintf(stderr, "Usage: %s radius threads infile outfile\n", argv[0]);
                exit(1);
        }
        radius = atoi(argv[1]);
        number_of_threads = atoi(argv[2]);
        if ((radius > MAX_RAD) || (radius < 1)) {
                fprintf(stderr,
                                "Radius (%d) must be greater than zero and less then %d\n",
                                radius, MAX_RAD);
```

```
        exit(1);
}
/* read file */
if (read_ppm(argv[3], &xsize, &ysize, &colmax, (char *) src) != 0) {
        exit(1);
}
if (colmax > 255) {
        fprintf(stderr, "Too large maximum color-component value\n");
        exit(1);
}

int number_of_elements, temp_start, temp_end;

struct data_to_thread* arguments[MAX_THREADS];
pthread_t* threads[MAX_THREADS];
struct data_to_thread* temp_arguments;
 clock_gettime( CLOCK_REALTIME, &start);
for (unsigned int i = 1; i < number_of_threads; i++) {
        temp_start = i * ysize / number_of_threads;
        if (temp_start < 0) {
                temp_start = 0;
        }
        temp_end = (i + 1) * ysize / number_of_threads;
        if (temp_end > ysize) {
                temp_end = ysize;

        }
        if (i == number_of_threads - 1) {
                temp_end = ysize;
        }
        temp_arguments = (struct data_to_thread*) malloc(
                        sizeof(struct data_to_thread));
        temp_arguments->end_y = temp_end;
        temp_arguments->radius = radius;
        temp_arguments->src = src;
        temp_arguments->start_y = temp_start;
        temp_arguments->xsize = xsize;
        temp_arguments->ysize = ysize;
        temp_arguments->dst = dst;
        arguments[i] = temp_arguments;
        threads[i] = (pthread_t*) malloc(sizeof(pthread_t));
        pthread_create(threads[i], NULL, &blurfiltermpi, (void *) arguments[i]);
}
temp_arguments = (struct data_to_thread*) malloc(
                sizeof(struct data_to_thread));

temp_end = (1) * ysize / number_of_threads;
if (temp_end > ysize) {
        temp_end = ysize;

}
if (0 == number_of_threads - 1) {
        temp_end = ysize;
}
```

```
        temp_arguments->end_y = temp_end;
        temp_arguments->radius = radius;
        temp_arguments->src = src;
        temp_arguments->start_y = 0;
        temp_arguments->xsize = xsize;
        temp_arguments->ysize = ysize;
        temp_arguments->dst = dst;
        arguments[0] = temp_arguments;
        blurfiltermpi((void *) temp_arguments);
         clock_gettime( CLOCK_REALTIME, &stop);
        for (unsigned int i = 1; i < number_of_threads; i++) {
                pthread_join(*threads[i], NULL);
        }
    accum = ( (double)stop.tv_sec - (double)start.tv_sec )
          + ( (double)stop.tv_nsec - (double)start.tv_nsec )
            / BILLION;
    printf( "it took: %lf\n", accum );
        write_ppm(argv[4], xsize, ysize, dst);
        free(dst);
        free(src);
        for (int i = 0; i < number_of_threads; i++) {
                free(arguments[i]);
                if (i > 0) {
                        free(threads[i]);
                }
        }

        return 0;
}
```

B thresthread.c

```
//standard imports
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <pthread.h>

#include "ppmio.h"
#include "gaussw.h"

//defines
#define MAX_RAD 1000
#define MAX_PIXELS (1000*1000)
#define MAX_THREADS 100
//#define _POSIX_BARRIERS 1

#define BILLION  1000000000L;
int min(int a, int b) {
        if (a < b) {
                return a;
        } else {
                return b;
```

```
        }
}

struct data_to_thread {
        unsigned char * src;
        unsigned char * dst;
        int xsize, ysize, start_y, end_y, radius;
        pthread_barrier_t* barrier;
        pthread_mutex_t* lock;
        int* sum;

} data_to_thread;

static void * blurfiltermpi(void *arguments) {
        struct data_to_thread* data = (struct data_to_thread *) arguments;
        unsigned char *src = data->src;
        unsigned char *dst = data->dst;
        int xsize = data->xsize;
        int ysize = data->ysize;
        int start_y = data->start_y;
        int end_y = data->end_y;
        int radius = data->radius;
        double w[MAX_RAD];
        get_gauss_weights(radius, w);
        double red, green, blue, n, temp_weight;
        register unsigned char* temp_pointer;
        int sum=0;
        //for each row
        for (int y = start_y; y < end_y; y++) {
                //for each column
                for (int x = 0; x < xsize; x++) {
                        temp_pointer = src + (xsize * y + x) * 3;
                        sum+=*temp_pointer+ *(temp_pointer + 1)+*(temp_pointer + 2);

                }

        }

        pthread_mutex_lock(data->lock);
        *(data->sum)+=sum;
        pthread_mutex_unlock(data->lock);
        pthread_barrier_wait (data->barrier);
        sum=*(data->sum)/(xsize*ysize);
        for (int y = start_y; y < end_y; y++) {
                //for each column
                for (int x = 0; x < xsize; x++) {
                        temp_pointer = src + (xsize * y + x) * 3;
                        if(*temp_pointer+ *(temp_pointer + 1)+*(temp_pointer + 2)<sum){
                                temp_pointer = dst + (xsize * y + x) * 3;
                                *temp_pointer=0;
                                *(temp_pointer+1)=0;
                                *(temp_pointer+2)=0;
                        }
                        else{
```

```
                                temp_pointer = dst + (xsize * y + x) * 3;
                                                *temp_pointer=255;
                                                *(temp_pointer+1)=255;
                                                *(temp_pointer+2)=255;
                                }

                        }

        }
        return 0;
}

int main(int argc, char ** argv) {
        //used to know how blurry the image will be
        int radius;

        // information about the picture
        int xsize, ysize, colmax, number_of_threads;

    struct timespec start, stop;
    double accum;

        char* src = (char*) malloc(sizeof(unsigned char) * MAX_PIXELS * 3);
        char* dst = (char*) malloc(sizeof(unsigned char) * MAX_PIXELS * 3);
        unsigned* sum=(unsigned*)malloc(sizeof(unsigned));
        *sum=0;

        if (argc != 5) {
                fprintf(stderr, "Usage: %s radius threads infile outfile\n", argv[0]);
                exit(1);
        }
        radius = atoi(argv[1]);
        number_of_threads = atoi(argv[2]);
        if ((radius > MAX_RAD) || (radius < 1)) {
                fprintf(stderr,
                                "Radius (%d) must be greater than zero and less then %d\n",
                                radius, MAX_RAD);
                exit(1);
        }
        /* read file */
        if (read_ppm(argv[3], &xsize, &ysize, &colmax, (char *) src) != 0) {
                exit(1);
        }
        if (colmax > 255) {
                fprintf(stderr, "Too large maximum color-component value\n");
                exit(1);
        }

        pthread_barrier_t* barrier=(pthread_barrier_t*)malloc(sizeof(pthread_barrier_t));
        int  s = pthread_barrier_init(barrier, NULL, number_of_threads);

        pthread_mutex_t* lock=( pthread_mutex_t*)malloc(sizeof( pthread_mutex_t));
        pthread_mutex_init(lock, NULL);
```

```
int number_of_elements, temp_start, temp_end;

struct data_to_thread* arguments[MAX_THREADS];
pthread_t* threads[MAX_THREADS];
struct data_to_thread* temp_arguments;

 clock_gettime( CLOCK_REALTIME, &start);
for (unsigned int i = 1; i < number_of_threads; i++) {
        temp_start = i * ysize / number_of_threads;
        if (temp_start < 0) {
                temp_start = 0;
        }
        temp_end = (i + 1) * ysize / number_of_threads;
        if (temp_end > ysize) {
                temp_end = ysize;

        }
        if (i == number_of_threads - 1) {
                temp_end = ysize;
        }
        temp_arguments = (struct data_to_thread*) malloc(
                        sizeof(struct data_to_thread));
        temp_arguments->end_y = temp_end;
        temp_arguments->radius = radius;
        temp_arguments->src = src;
        temp_arguments->start_y = temp_start;
        temp_arguments->xsize = xsize;
        temp_arguments->ysize = ysize;
        temp_arguments->dst = dst;
        temp_arguments->barrier=barrier;
        temp_arguments->lock=lock;
        temp_arguments->sum=sum;
        arguments[i] = temp_arguments;
        threads[i] = (pthread_t*) malloc(sizeof(pthread_t));
        pthread_create(threads[i], NULL, &blurfiltermpi, (void *) arguments[i]);
}
temp_arguments = (struct data_to_thread*) malloc(
                sizeof(struct data_to_thread));

temp_end = (1) * ysize / number_of_threads;
if (temp_end > ysize) {
        temp_end = ysize;

}
if (0 == number_of_threads - 1) {
        temp_end = ysize;
}
temp_arguments->end_y = temp_end;
temp_arguments->radius = radius;
temp_arguments->src = src;
temp_arguments->start_y = 0;
temp_arguments->xsize = xsize;
temp_arguments->ysize = ysize;
temp_arguments->dst = dst;
```

```
        temp_arguments->barrier=barrier;
        temp_arguments->lock=lock;
        temp_arguments->sum=sum;
        arguments[0] = temp_arguments;
        blurfiltermpi((void *) temp_arguments);
        for (unsigned int i = 1; i < number_of_threads; i++) {
                pthread_join(*threads[i], NULL);
        }
         clock_gettime( CLOCK_REALTIME, &stop);
        write_ppm(argv[4], xsize, ysize, dst);

    accum = ( (double)stop.tv_sec - (double)start.tv_sec )
          + ( (double)stop.tv_nsec - (double)start.tv_nsec )
            / BILLION;
    printf( "it took: %lf\n", accum );
        free(dst);
        free(src);
        pthread_mutex_destroy(lock);
        free(lock);
        free(barrier);
        free(sum);
        for (int i = 0; i < number_of_threads; i++) {
                free(arguments[i]);
                if (i > 0) {
                        free(threads[i]);
                }
        }

        return 0;
}
```