Lab 1
MPI

Martin Söderén
marso329, 9009291098

12 maj 2016

1 Introduction

The problem consists of implementing and parallelizing two image filters using OpenMPI. 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

Algorithm 1 Master thread blur filter

```
procedure MASTER
    Read file
    Broadcast size of image using MPI_Bcast
    Calculate which part of the image to work on
    for all slave threads do
        Calculate slave threads part to work on
        Send part of the image using MPI_Isend
    end for
    Calculate part of the image
    for all slave threads do
        Calculate slave threads part to work on
        Receive part of the image using MPI_Recv
        Store result in destination
    end for
    Store own result in destination
    Write destination to file
end procedure
```

Algorithm 2 Slave thread blur filter

```
procedure SLAVE
    Receive size of image using MPI_Bcast
    Calculate which part of the image to work on
    Receive part of the image using MPI_Recv
    Calculate part of the image
    Send part of the image using MPI_Send
end procedure
```

2.2 Threshold filter algorithm

Algorithm 3 Master thread threshold filter

```
procedure MASTER
    Read file
    Broadcast size of image using MPI_Bcast
    Calculate which part of the image to work on
    for all slave threads do
        Calculate slave threads part to work on
        Send part of the image using MPI_Isend
    end for
    Calculate part of the image
    for all slave threads do
        Calculate slave threads part to work on
        Receive threshold of that part using MPI_Recv
    end for
    Calculate threshold for complete image
    Broadcast threshold using MPI_Bcast
    Create part of new image
    for all slave threads do
        Calculate slave threads part to work on
        Receive part of the image using MPI_Recv
    end for
    Write image to file
end procedure
```

Algorithm 4 Slave thread threshold filter

```
procedure SLAVE
    Receive size of image using MPI_Bcast
    Calculate which part of the image to work on
    Receive part of the image using MPI_Recv
    Calculate threshold for that part
    Send threshold of the image using MPI_Send
    Receive complete threshold using MPI_Bcast
    Create part of the image
    Send part of image back using MPI_Send
end procedure
```

2.3 Design

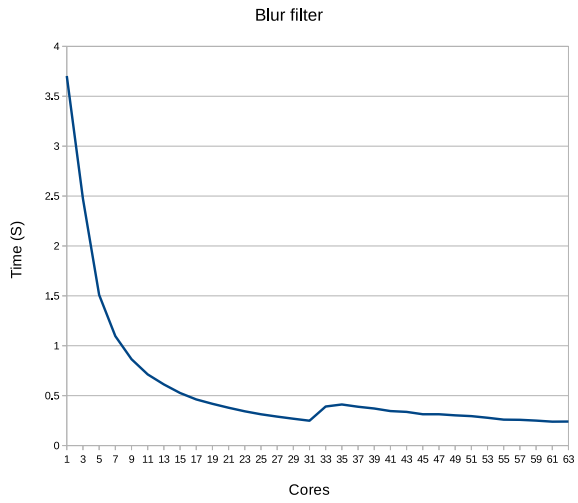
The image is split up by rows to utilize the space locality of the data and the data in both cases are accessed row wise for improved spatial locality. For example in the blur filter the pixelvalue is not calculated for each radius but for each row. If it was calculated for each radius that meant that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

3 Result

The blur filter scales well and hits the lower time limit at around 31 cores with an improvement compared

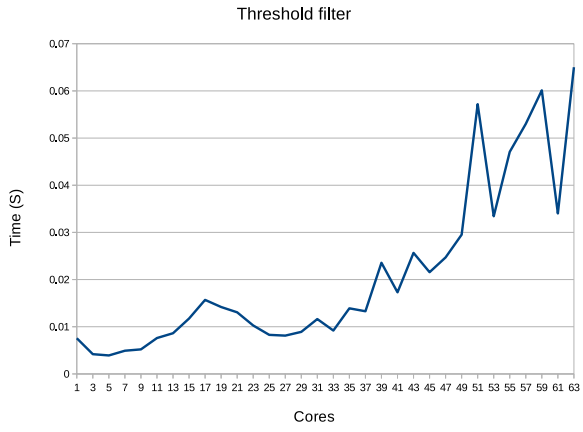
to a single core of 15 times faster. After 31 cores the calculation time starts to increase. This could be because each cores does to little work so much of the time is spent sending data to all cores.

The threshold filter scales rather poorly. The cause for this is probably because the calculation is not that intense. Each core just has to go through all pixels and add together their values which can be done very quickly so most of the time of the calculation is spent on sending data to the cores. This algorithm should be done on a single core.



Figur 1: Times for blur filter with radius 21

4 Result



Figur 2: Times for threshold filter

A blurmainmpi.c

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

//non-standard imports
#include "ppmio.h"
#include "blurfilter.h"
#include "gaussw.h"

//defines
#define MAX_RAD 1000

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

void blurfiltermpi(unsigned char *src, int xsize, int ysize, int start_y,
                int end_y, int 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 = src + (xsize * y + x) * 3;
                        *temp_pointer = red / n;
                        *(temp_pointer + 1) = green / n;
                        *(temp_pointer + 2) = blue / n;

                }

        }

}

int main(int argc, char ** argv) {
        //used to know where we are in the hierarchy
        int rank, size;

        //used to calc time
        double starttime, endtime;

        //used to know how blurry the image will be
        int radius;

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

        //store picture
        pixel src[MAX_PIXELS];

        MPI_Init(&argc, &argv); /* starts MPI */
        MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* get current process id */
        MPI_Comm_size(MPI_COMM_WORLD, &size); /* get number of processes */

        if (argc != 4) {
                fprintf(stderr, "Usage: %s radius infile outfile\n", argv[0]);
                exit(1);
        }
        radius = atoi(argv[1]);
        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);
        }

        if (rank == 0) {
                /* read file */
                if (read_ppm(argv[2], &xsize, &ysize, &colmax, (char *) src) != 0) {
                        exit(1);
                }
```

```
        if (colmax > 255) {
                fprintf(stderr, "Too large maximum color-component value\n");
                exit(1);
        }

}
starttime = MPI_Wtime();
//send problem size to slaves
MPI_Bcast(&xsize, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&ysize, 1, MPI_INT, 0, MPI_COMM_WORLD);
int start_y = rank * ysize / size;
int end_y = (rank + 1) * ysize / size;
if (rank == size - 1) {
        end_y = ysize;
}
//send data so slaves
int temp_start;
int temp_end;

if (rank == 0) {
        int number_of_elements;
        for (unsigned int i = 1; i < size; i++) {
                int temp_start = i * ysize / size - radius;
                if (temp_start < 0) {
                        temp_start = 0;
                }
                int temp_end = (i + 1) * ysize / size + radius;
                if (temp_end > ysize) {
                        temp_end = ysize;

                }
                if (i == size - 1) {
                        temp_end = ysize;
                }
                number_of_elements = (temp_end - temp_start) * 3 * xsize;
                temp_start = temp_start * 3 * xsize;


                MPI_Request temp_request;
                MPI_Isend((char *) src + temp_start, number_of_elements, MPI_CHAR,
                                i, 0, MPI_COMM_WORLD, &temp_request);
        }
} else {

        temp_start = rank * ysize / size - radius;
        if (temp_start < 0) {
                temp_start = 0;
        }
        temp_end = (rank + 1) * ysize / size + radius;
        if (temp_end > ysize) {
                temp_end = ysize;

        }
        if (rank == size - 1) {
```

```
                temp_end = ysize;
        }

        int number_of_elements = (temp_end - temp_start) * 3 * xsize;
        int start = temp_start * 3 * xsize;
        MPI_Status temp_status1;
        MPI_Recv((char *) src + start, number_of_elements, MPI_CHAR, 0, 0,
                        MPI_COMM_WORLD, &temp_status1);

}

//do works

temp_start = rank * ysize / size;
if (temp_start < 0) {
        temp_start = 0;
}
temp_end = (rank + 1) * ysize / size;
if (temp_end > ysize) {
        temp_end = ysize;

}
if (rank == size - 1) {
        temp_end = ysize;
}

blurfiltermpi((unsigned char *) src, xsize, ysize, temp_start, temp_end,
                radius);
if (rank == 0) {
        int temp_start;
        int temp_end;
        int number_of_elements;
        pixel dst[MAX_PIXELS];
        for (unsigned int i = 1; i < size; i++) {
                int temp_start = i * ysize / size;
                int temp_end = (i + 1) * ysize / size;
                if (i == size - 1) {
                        temp_end = ysize;
                }
                number_of_elements = (temp_end - temp_start) * 3 * xsize;
                temp_start = temp_start * 3 * xsize;

                MPI_Request temp_request;
                MPI_Status temp_status;
                MPI_Recv((char *) dst + temp_start, number_of_elements, MPI_CHAR, i
                                0, MPI_COMM_WORLD, &temp_status);
        }
        memcpy(dst, src, sizeof(char) * end_y * xsize * 3);
        write_ppm(argv[3], xsize, ysize, (char *) dst);
} else {
        int number_of_elements = (end_y - start_y) * 3 * xsize;
        int start = start_y * 3 * xsize;
        MPI_Send((char *) src + start, number_of_elements, MPI_CHAR, 0, 0,
                        MPI_COMM_WORLD);
```

```
        }
        endtime = MPI_Wtime();
        if (rank==0){
                printf("That took %f seconds\n",endtime-starttime);
        }

        MPI_Finalize();
}
```

B thresfiltermpi.c

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


//non-standard imports
#include "ppmio.h"
#include "blurfilter.h"
#include "gaussw.h"


//defines
#define MAX_PIXELS (1000*1000)


unsigned int calculcatethres(unsigned char *src, int xsize, int ysize, int start_y,
                int end_y) {
        unsigned char* temp_pointer;
        unsigned 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);
                }

        }
        return sum;

}

void thresfiltermpi(unsigned char *src, int xsize, int ysize, int start_y,
                int end_y, unsigned thres) {
        unsigned char* temp_pointer;
        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)>thres){
                                *temp_pointer = 255;
                                *(temp_pointer + 1) = 255;
                                *(temp_pointer + 2) = 255;
                        }
                        else{
                       *temp_pointer = 0;
                       *(temp_pointer + 1) = 2;
                       *(temp_pointer + 2) = 0;

                        }
                        }

        }

}

int main(int argc, char ** argv) {
        //used to know where we are in the hierarchy
        int rank, size;

        //used to know how blurry the image will be
        int radius;

         double starttime, endtime;

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

        //store picture
        pixel src[MAX_PIXELS];

        MPI_Init(&argc, &argv); /* starts MPI */
        MPI_Comm_rank(MPI_COMM_WORLD, &rank); /* get current process id */
        MPI_Comm_size(MPI_COMM_WORLD, &size); /* get number of processes */

        if (argc != 3) {
                fprintf(stderr, "Usage: %s infile outfile\n", argv[0]);
                exit(1);
        }
        radius = atoi(argv[1]);

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

        }
        starttime = MPI_Wtime();
```

```
//send problem size to slaves
MPI_Bcast(&xsize, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&ysize, 1, MPI_INT, 0, MPI_COMM_WORLD);
int start_y = rank * ysize / size;
int end_y = (rank + 1) * ysize / size;
if (rank == size - 1) {
        end_y = ysize;
}
//send data so slaves
int temp_start;
int temp_end;
if (rank == 0) {
        int number_of_elements;
        for (unsigned int i = 1; i < size; i++) {
                int temp_start = i * ysize / size - radius;
                if (temp_start < 0) {
                        temp_start = 0;
                }
                int temp_end = (i + 1) * ysize / size + radius;
                if (temp_end > ysize) {
                        temp_end = ysize;

                }
                if (i == size - 1) {
                        temp_end = ysize;
                }
                number_of_elements = (temp_end - temp_start) * 3 * xsize;
                temp_start = temp_start * 3 * xsize;

                MPI_Request temp_request;
                MPI_Isend((char *) src + temp_start, number_of_elements, MPI_CHAR,
                                i, 0, MPI_COMM_WORLD, &temp_request);
        }
} else {

        temp_start = rank * ysize / size - radius;
        if (temp_start < 0) {
                temp_start = 0;
        }
        temp_end = (rank + 1) * ysize / size + radius;
        if (temp_end > ysize) {
                temp_end = ysize;

        }
        if (rank == size - 1) {
                temp_end = ysize;
        }

        int number_of_elements = (temp_end - temp_start) * 3 * xsize;
        int start = temp_start * 3 * xsize;
        MPI_Status temp_status;
        MPI_Recv((char *) src + start, number_of_elements, MPI_CHAR, 0, 0,
                        MPI_COMM_WORLD, &temp_status);
```

```
}

//do works

temp_start = rank * ysize / size;
if (temp_start < 0) {
        temp_start = 0;
}
temp_end = (rank + 1) * ysize / size;
if (temp_end > ysize) {
        temp_end = ysize;

}
if (rank == size - 1) {
        temp_end = ysize;
}
unsigned int thres=calculcatethres((unsigned char *) src, xsize, ysize, temp_start,

if (rank==0){
        unsigned int temp=0;
        for (unsigned int i = 1; i < size; i++) {

                MPI_Request temp_request;
                MPI_Status temp_status1;
                MPI_Recv(&temp, 1, MPI_UNSIGNED, i,
                                0, MPI_COMM_WORLD, &temp_status1);
                thres+=temp;
        }
        thres=thres/(xsize*ysize);

}
else{
        MPI_Send(&thres, 1, MPI_UNSIGNED, 0, 0,
                        MPI_COMM_WORLD);

}
MPI_Bcast(&thres, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);

thresfiltermpi((unsigned char *)src, xsize,ysize,temp_start,temp_end,thres);

if (rank == 0) {
        int temp_start;
        int temp_end;
        int number_of_elements;
        pixel dst[MAX_PIXELS];
        for (unsigned int i = 1; i < size; i++) {
                int temp_start = i * ysize / size;
                int temp_end = (i + 1) * ysize / size;
                if (i == size - 1) {
                        temp_end = ysize;
                }
                number_of_elements = (temp_end - temp_start) * 3 * xsize;
                temp_start = temp_start * 3 * xsize;
```

```
                        MPI_Request temp_request;
                        MPI_Status temp_status2;
                        MPI_Recv((char *) dst + temp_start, number_of_elements, MPI_CHAR, i
                                        0, MPI_COMM_WORLD, &temp_status2);
                }
                memcpy(dst, src, sizeof(char) * end_y * xsize * 3);
                write_ppm(argv[2], xsize, ysize, (char *) dst);
        } else {
                int number_of_elements = (end_y - start_y) * 3 * xsize;
                int start = start_y * 3 * xsize;
                MPI_Send((char *) src + start, number_of_elements, MPI_CHAR, 0, 0,
                                MPI_COMM_WORLD);

        }
        endtime = MPI_Wtime();
        if (rank==0){
                printf("That took %f seconds\n",endtime-starttime);
        }

        MPI_Finalize();
}
```