# Project Assignment 2

—

# Multithreaded Programming using Pthreads

Performance Optimization, DV1463

Håkan Grahn, Emiliano Casalicchio
Blekinge Institute of Technology

December 1, 2016

## 1 Introduction

The task in this project assignment is to implement three parallel algorithms on a shared-memory computer with 8 cpus using Pthreads (POSIX threads).

The examination of this project is done by sending in the project, with complete source code and a report in PDF format, before the submission deadline. The submission deadline is December 30th, 2016 at 23:55. The reports and source code should be submitted on It's Learning.

## 2 Goals

This project assignment serves two purposes:

- Introduce you to basic Pthreads programming.
- Give some experiece of how work and data partitioning as well as synchronization impact the performance of a parallel application on a shared address-space computer.

The rationale behind these goals is that Pthreads is one of the most common approach to implement high-performance parallel applications on shared address-space computers. On such machines, data communication is implicit but may still impact the performance. Further, the synchronization of parallel threads and the protection of shared data may also have a significant impact on the performance.

## 3 Preconditions

### 3.1 Prerequisites

- You are supposed to have good programming experience and not be new to C programming.
- You are supposed to have basic knowledge about working in a Unix/Linux environment.
- Operating systems issues and concepts should not be unfamiliar to you.

### 3.2 Laboratory groups

You are encouraged to work in groups of two. Groups larger than two is not accepted.

Discussion and help between laboratory groups are encouraged. It is normally not a problem, *but* watch out so that you do not cross the border to cheating, see section 3.5 below.

### 3.3 Lecture support

There is one lecture on multiprocessors and one lecture on programming with pthreads in general, e.g., introducing the general concepts of shared-memory programming and an overview of the pthreads functions. In addition to this, you are expected to search for additional information on your own.

### 3.4 Examination

See section 4.4.

### 3.5 Cheating

All work that is not your own should be properly referenced. If not, it will be considered as cheating and reported as such to the university disciplinary board.

## 4 Project Tasks to Complete

In this project assignment you are going to implement parallel versions of three different algorithms:

1. Implement a parallel version of a Mandelbrot fractal calculation (described in Section 4.1).
2. Implement a parallel version of a blocked (2-dimensional) Matrix-Matrix multiplication (described in Section 4.2)
3. Implement a parallel version of the quicksort algorithm (described in Section 4.3).

The algorithms shall be implemented using pthreads, and compile and execute correctly on a Linux-based shared-memory machine with 8 cpus. We will use `kraken` to test and verify that your applications are correct.

### 4.1 Fractal calculation

#### 4.1.1 Problem Description

The first application that you shall parallelize is an application that calculates one of the most known fractals, i.e., the Mandelbrot set which is shown in Figure 1. The code for a sequential version of the Mandelbrot calculation is found in the file `fractal_seq.c`, and is also listed in Appendix A.

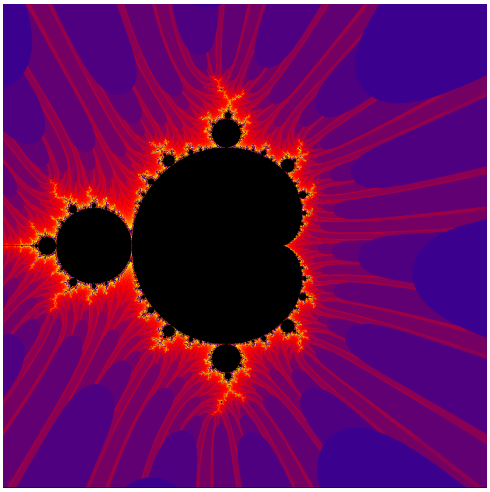

Figure 1: Picture showing the Mandelbrot set.

### 4.1.2 Tasks to Complete

You are supposed to do the following:

- Write a parallel version of the fractal calculation using pthreads.
- Measure and compare the execution times for (i) the sequential version given to you, (ii) your parallel version on 1 cpu/thread, and (iii) your parallel version on 8 cpus/threads. Further, calculate the speedup of your parallel application on 8 cpus.

The source code for the sequential version, `fractal_seq.c`, is found on the course home page on It's Learning.

## 4.2 Blocked Matrix-Matrix Multiplication

### 4.2.1 Problem Description

The sequential algorithm for performing Matrix-Matrix multiplication is shown in Figure 2. The sequential version of matrix-matrix multiplication written in C is found in Appendix B.

```
1.  procedure MAT_MULT (A, B, C)
2.  begin
3.      for i := 0 to n − 1 do
4.          for j := 0 to n − 1 do
5.              begin
6.                  C[i, j] := 0;
7.                  for k := 0 to n − 1 do
8.                      C[i, j] := C[i, j] + A[i, k] × B[k, j];
9.              endfor;
10. end MAT_MULT
```

**Algorithm 8.2** The conventional serial algorithm for multiplication of two $n \times n$ matrices.

Figure 2: Algorithm for matrix-matrix multiplication.

An exemple of an implementation of *row-wise* 1-dimensional Matrix-Matrix multiplication was done a laboratory 3. You are allowed to use the code as basis your own *blocked 2-dimensional* (checkerboard) implementation.

### 4.2.2 Tasks to Complete

You are supposed to do the following:

- Write a parallel implementation of *blocked* (2-dimensional!) Matrix-Matrix multiplication using pthreads.
- The parallel implementation shall execute correctly on 8 cores.
- Compare the performance (i.e., execution time) of your blocked version with the performance the row-wise parallel version that you developed at laboratory 3. The measurements should be done on 8 cores.

The source code for the sequential version of Matrix-Matrix multiplication is found on the course home page at It's Learning.

## 4.3 Parallel Quicksort implementation

### 4.3.1 Problem Description

The final application we shall parallelize is Quicksort. We saw in the laboratory exercise that Matrix multiplication is easy to parallelize using data decomposition and extracting loop parallelism. In contrast, Quicksort fits well for recursive decomposition. A sequential version of Quicksort is found in the

file `qsort_seq.c`. The code for the sequential implementation of the Quicksort algorithm is listed in Appendix C.

### 4.3.2 Tasks to Complete

You are supposed to do the following:

- Write a parallel version of Quicksort using pthreads.
- Measure and compare the execution times for (i) the sequential version given to you, (ii) your parallel version on 1 cpu/thread, and (iii) your parallel version on 8 cpus/threads. Further, calculate the speedup of your parallel application on 8 cpus. Your application shall have a speedup of at least 2 on 8 cpus.

The source code for the sequential implementation of Quicksort is found on the course home page at It's Learning.

## 4.4 Examination

Prepare and submit a `tar`-file (or `zip`-file) containing:

- **Source code:** The source-code for working solutions to the tasks in sections 4.1, 4.2, and 4.3, i.e., a listing of your well-commented source code for your parallel version of the applications.
- Corresponding `Makefile`(s), or a text-file describing how to compile the applications.
- **Measurements:** You should provide execution times for three cases for each application: (i) the sequential version of the application, (ii) the parallel version of the application running on one cpu/thread, and (iii) the parallel version of the application running on eight cpus. In addition, you shall calculate the speedup of your parallel application on 8 cpus.
- **Implementation:** A short, general description (one-two pages) of your parallel implementations, i.e., you should describe how you have partitioned the work between the cpus, how the data structures are organized, etc. The format of the report must be pdf.

All material (except the code given to you in this assignment) must be produced by the laboratory group alone.

The examiner may contact you within a week, if he needs some oral clarifications on your code or report. In this case, all group members must be present at that oral occasion.

# A Source code for Fractal

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>

// draws a mandelbrot fractal
// compile with gcc fractal.c -lm -std=c99 -o fractal

int pal[256] = {
    0xb2000a,0xb20009,0xb2000a,0xb1000a,0xb1000b,0xaf000d,0xaf000e,0xae000f,0xad0011,0xac0012,0xab0013,0xaa0015,
    0xa90016,0xa80018,0xa70019,0xa6001b,0xa4001d,0xa3001e,0xa2001f,0xa00022,0x9f0023,0x9e0025,0x9c0028,0x9b0029,
    0x9a002b,0x99002d,0x97002f,0x960031,0x940033,0x930035,0x910037,0x8f0039,0x8e003c,0x8d003d,0x8b0040,0x8a0042,
    0x870044,0x860047,0x840048,0x82004b,0x81004e,0x7f0050,0x7d0053,0x7b0055,0x7a0057,0x780059,0x76005c,0x74005e,
    0x730061,0x710063,0x6f0066,0x6e0068,0x6b006a,0x6a006d,0x680070,0x660072,0x640075,0x630077,0x60007a,0x5e007d,
    0x5c007f,0x5b0081,0x590084,0x570086,0x550089,0x54008c,0x52008f,0x500091,0x4e0093,0x4c0096,0x4a0099,0x49009b,
    0x46009e,0x4500a1,0x4300a2,0x4100a5,0x4000a8,0x3e00aa,0x3b00ac,0x3a00af,0x3900b1,0x3600b4,0x3500b6,0x3300b9,
    0x3100bb,0x2f00bd,0x2d00c0,0x2c00c2,0x2a00c4,0x2900c7,0x2700c9,0x2500cb,0x2400cd,0x2200d0,0x2100d2,0x2000d4,
    0x1e00d5,0x1c00d8,0x1b00da,0x1a00dc,0x1800dd,0x1700e0,0x1600e1,0x1400e4,0x1200e5,0x1200e7,0x1100e8,0xf00ea,
    0xd00ec,0xc00ee,0xc00ef,0xa00f1,0x900f2,0x900f4,0x700f6,0x600f6,0x600f8,0x400fa,0x300fb,0x200fb,0x100fd,
    0x100fe,0xff,0x1ff,0x2ff,0x3ff,0x5ff,0x6ff,0x8ff,0x9ff,0xaff,0xcff,0xdff,
    0xfff,0x10ff,0x12ff,0x13ff,0x15ff,0x17ff,0x19ff,0x1aff,0x1cff,0x1fff,0x20ff,0x22ff,
    0x24ff,0x26ff,0x29ff,0x2aff,0x2cff,0x2eff,0x31ff,0x34ff,0x35ff,0x38ff,0x3aff,0x3cff,
    0x3eff,0x41ff,0x43ff,0x46ff,0x49ff,0x4bff,0x4dff,0x50ff,0x52ff,0x54ff,0x57ff,0x59ff,
    0x5cff,0x5eff,0x61ff,0x64ff,0x66ff,0x69ff,0x6cff,0x6eff,0x71ff,0x74ff,0x76ff,0x79ff,
    0x7bff,0x7eff,0x81ff,0x83ff,0x86ff,0x89ff,0x8bff,0x8eff,0x91ff,0x93ff,0x96ff,0x98ff,
    0x9bff,0x9dff,0xa0ff,0xa2ff,0xa5ff,0xa7ff,0xa9ff,0xadff,0xafff,0xb1ff,0xb4ff,0xb6ff,
    0xb9ff,0xbcff,0xbdff,0xbfff,0xc2ff,0xc5ff,0xc7ff,0xc9ff,0xcbff,0xceff,0xcfff,
    0xd1ff,0xd3ff,0xd6ff,0xd7ff,0xd9ff,0xdcff,0xdeff,0xe0ff,0xe2ff,0xe4ff,0xe5ff,
    0xe7ff,0xe9ff,0xebff,0xecff,0xedff,0xefff,0xf1ff,0xf2ff,0xf3ff,0xf5ff,
    0xf6ff,0xf7ff,0xf8ff,0xfaff,0xfbff,0xfcff,0xfcff,0xfcff,0xfcff,0xfcff,0xfcff,
    0xfcff,0xfcff,0xfcff,0xfcff,0xfcff,0xfcff,0xfcff,0xfcff,0xfcff,0xfcff};

void mandelbrot(float width, float height, unsigned int *pixmap)
{
        int i, j;
        float xmin = -1.6f;
        float xmax = 1.6f;
        float ymin = -1.6f;
        float ymax = 1.6f;
        for (i = 0; i < height; i++) {
                for (j = 0; j < width; j++) {
                        float b = xmin + j * (xmax - xmin) / width;
                        float a = ymin + i * (ymax - ymin) / height;
                        float sx = 0.0f;
                        float sy = 0.0f;
                        int ii = 0;
                        while (sx + sy <= 64.0f) {
                                float xn = sx * sx - sy * sy + b;
                                float yn = 2 * sx * sy + a;
                                sx = xn;
                                sy = yn;
                                ii++;
                                if (ii == 1500) {
                                        break;
                                }
                        }
                        if (ii == 1500) {
                                pixmap[j+i*(int)width] = 0;
                        }
                        else {
                                int c = (int)((ii / 32.0f) * 256.0f);
                                pixmap[j + i *(int)width] = pal[c%256];
                        }
                }
        }
```

```
}

void writetga(unsigned int *pixmap, unsigned int width, unsigned int height, char *name)
{
	FILE *f;
	int i,j;
	char buffer[50];
	f = fopen(name,"wb");
	fwrite("\x00\x00\x02",sizeof(char),3,f);
	fwrite("\x00\x00\x00\x00\x00",sizeof(char),5,f);
	fwrite("\x00\x00",sizeof(char),2,f);
	fwrite("\x00\x00",sizeof(char),2,f);
	sprintf(buffer,"%c%c",(width & 0x00ff)%0xff,(width & 0xff00)%0xff);
	fwrite(buffer,sizeof(char),2,f);
	sprintf(buffer,"%c%c",(height & 0x00ff)%0xff,(height & 0xff00)%0xff);
	fwrite(buffer,sizeof(char),2,f);
	fwrite("\x18\x00",sizeof(char),2,f);
	for (i = height-1; i >= 0; i--) {
		for (j = 0; j < width; j++) {
			sprintf(buffer, "%c%c%c",
				(pixmap[j+(i)*width]>>16)&0x000000ff,
				(pixmap[j+i*width]>>8)&0x000000ff,
				(pixmap[j+i*width])&0x000000ff);
			fwrite(buffer,sizeof(char),3,f);
		}
	}
	fclose(f);
}


int main(int a, char *args[])
{
	int i, j;
	printf("fractal");
	unsigned int* pixmap = malloc(1024*1024*sizeof(int));
	mandelbrot(1024.0f, 1024.0f, pixmap);
	writetga(pixmap, 1024, 1024, "fracout.tga");
	free(pixmap);
	return 0;
}
```

# B Sequential matrix-matrix multiplication

```
/*****************************************************************************
 *
 * Sequential version of Matrix−Matrix multiplication
 *
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>

#define SIZE 1024

static double a[SIZE][SIZE];
static double b[SIZE][SIZE];
static double c[SIZE][SIZE];

static void
init_matrix(void)
{
    int i, j;

    for (i = 0; i < SIZE; i++)
        for (j = 0; j < SIZE; j++) {
          /* Simple initialization, which enables us to easy check
           * the correct answer. Each element in c will have the same
           * value as SIZE after the matmul operation.
           */
          a[i][j] = 1.0;
          b[i][j] = 1.0;
        }
}

static void
matmul_seq()
{
    int i, j, k;

    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            c[i][j] = 0.0;
            for (k = 0; k < SIZE; k++)
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}

static void
print_matrix(void)
{
    int i, j;

    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++)
          printf(" %7.2f", c[i][j]);
      printf("\n");
    }
}

int
main(int argc, char **argv)
{
    init_matrix();
    matmul_seq();
    //print_matrix();
}
```

# C Source code for Quicksort

```
/**********************************************************************
 *
 * Sequential version of Quick sort
 *
 **********************************************************************/

#include <stdio.h>
#include <stdlib.h>

#define KILO (1024)
#define MEGA (1024*1024)
#define MAX_ITEMS (64*MEGA)

#define swap(v, a, b) {unsigned tmp; tmp=v[a]; v[a]=v[b]; v[b]=tmp;}

static int *v;

static void
print_array(void)
{
    int i;

    for (i = 0; i < MAX_ITEMS; i++)
        printf("%d ", v[i]);
    printf("\n");
}

static void
init_array(void)
{
    int i;

    v = (int *) malloc(MAX_ITEMS*sizeof(int));
    for (i = 0; i < MAX_ITEMS; i++)
        v[i] = rand();
}

static unsigned
partition(int *v, unsigned low, unsigned high, unsigned pivot_index)
{
    /* move pivot to the bottom of the vector */
    if (pivot_index != low)
        swap(v, low, pivot_index);

    pivot_index = low;
    low++;

    /* invariant:
     * v[i] for i less than low are less than or equal to pivot
     * v[i] for i greater than high are greater than pivot
     */

    /* move elements into place */
    while (low <= high) {
        if (v[low] <= v[pivot_index])
            low++;
        else if (v[high] > v[pivot_index])
            high--;
        else
            swap(v, low, high);
    }

    /* put pivot back between two groups */
    if (high != pivot_index)
        swap(v, pivot_index, high);
    return high;
}
```

```
static void
quick_sort(int *v, unsigned low, unsigned high)
{
    unsigned pivot_index;

    /* no need to sort a vector of zero or one element */
    if (low >= high)
        return;

    /* select the pivot value */
    pivot_index = (low+high)/2;

    /* partition the vector */
    pivot_index = partition(v, low, high, pivot_index);

    /* sort the two sub arrays */
    if (low < pivot_index)
        quick_sort(v, low, pivot_index-1);
    if (pivot_index < high)
        quick_sort(v, pivot_index+1, high);
}

int
main(int argc, char **argv)
{
    init_array();
    //print_array();
    quick_sort(v, 0, MAX_ITEMS-1);
    //print_array();
}
```