Name: Emil Ljung
Institution: Computer science
Educational program: Game programming
Course: DV1463 Performance optimization
Date: 2017-05-07

Project 2 - Multithreaded Programming using Pthreads

Note: All the files which were used for the assignments are included in the zip-file and are named so that it's easy to identify which file was used for which task. For instance, matmul_8threads.c was used for getting the execution time for blocked matrix multiplication with 8 threads.
All files are compiled with: **gcc -std=c99 {filename] -lm -lpthread -o {output name}**
Run the code with: **/usr/bin/time ./{output name}**

Fractal

Sequential: 3.21 seconds
1 thread: 3.12 seconds
8 threads: 1.32 seconds

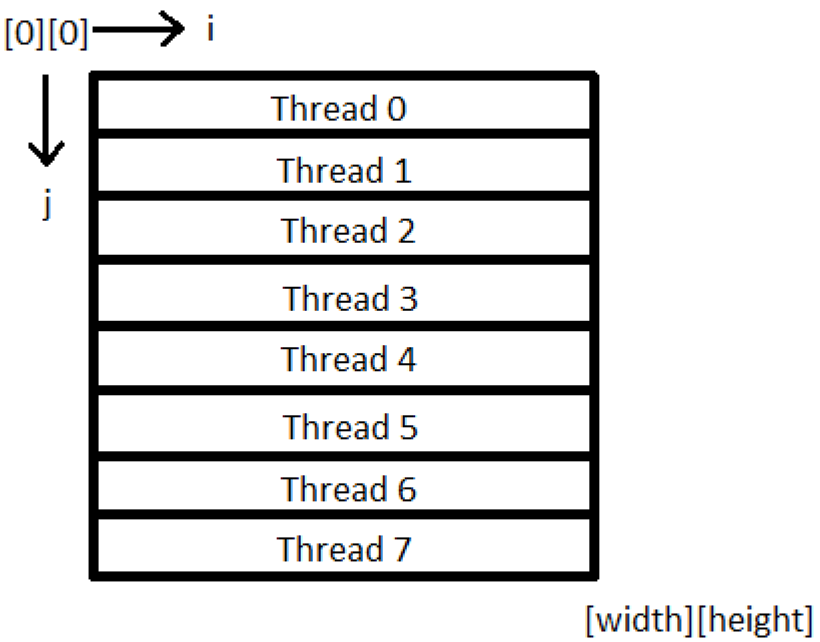
Speedup: 243.2%

The logic behind splitting up the work between the threads in the files fractal_1thread.c & fractal_8threads.c are the same. The only exception is that the #define variable NR_OF_THREADS equals 8 in fractal_8threads.c and 1 in fractal_1thread.c. All the threads which are going to be created exists in an array called threads. Each thread have a struct each which belongs to them (all the structs can be found in the array t_data). The struct is simply called thread_data and consists of 1 int pointer for the pixel map and 2 int values which tells from and to which j the threads shall work on. The 2 j values are based on NR_OF_THREADS and the #define variable width, which equals 1024. This way we can get a parallelization. In the figure below you can see how I imagined it was done. However, it is NOT what is actually happening in the code.

All threads are created in the function init_threads_mandelbrot. This is also where the structs get their values which will be used in the function mandelbrot, which is the function each thread will run. There you'll see the 2 for loops which are:

```
for (i = 0; i < width; i++) {
        for(j = data->start_j; j < data->end_j; j++) {
                ...
        ...
}
```

This code snippet is what made me treat the parallelization as the figure below shows since it made me think about a 2D array and thus treat it as a such.



How I imagined the partioning between each thread.

Blocked Matrix-Matrix multiplication

Row wise 8 threads: 11.69 seconds
Blocked 8 threads: 13.57 seconds

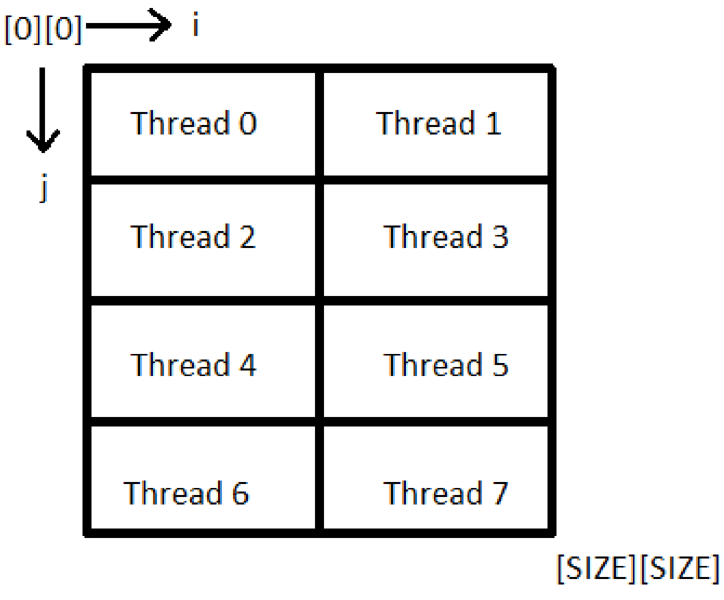
The way I solved this is similar to how the fractal was threaded. Basically, the difference is that in this solution the thread_data struct also have int i_start & i_end, which are used like the j counterparts found in the struct in the fractal assignment.

Since there was no requirement to implement a 1 thread version of this I only focused on the solution for 8 threads. There were also nothing that stated exactly how the partitioning was supposed to be between the threads, so I split up the work the way the figure below shows.

The code for splitting up the work looks like this:

```
for(i = 0; i < NR_OF_THREADS*0.5; i++) {
        int k = i*2;
        t_data[k].start_i = 0;
        t_data[k].end_i = SIZE*0.5;
        t_data[k].start_j = SIZE/(NR_OF_THREADS*0.5) * i;
        t_data[k].end_j = SIZE/(NR_OF_THREADS*0.5) * (i+1);
        int t0 = pthread_create(&threads[k], NULL, row_calculation_blocked, &t_data[k]);

        t_data[k+1].start_i = SIZE*0.5;
        t_data[k+1].end_i = SIZE;
        t_data[k+1].start_j = t_data[k].start_j;
        t_data[k+1].end_j = t_data[k].end_j;
        int t1 = pthread_create(&threads[k+1], NULL, row_calculation_blocked, &t_data[k+1]);
}
```



Quick Sort

Sequential: 30.61 seconds
1 thread: 31.53 seconds
8 threads: 11.63 seconds

Speedup: 263.2%

For this one I specifically went for the 8 threads solution first. What I did here was that I first created the root thread which pretty much just runs thread_quick_sort. This function is in several ways like the function quick_sort but when quick_sort isn't called a new thread which runs thread_quick_sort is called. After the parent thread has created its 2 child threads (1 for the values to the left of the pivot and 1 for the ones to the right in the array) it'll call pthread_join for both child threads, thus making the parent waiting for both of them to complete their tasks. This means that if I set NR_OF_THREADS to 8, which at first seems quite reasonable, then, if we look at the figure below, we'll see that the tree will have 4 leaf threads (0-7 are created). Since the parent threads are just waiting for the children we only have 4 ACTIVE threads. In order to create 8 active leaf threads NR_OF_THREADS must be set to 15!

Since I, in thread_quick_sort, have the if-statement if(childID > NR_OF_THREADS-1) which starts sorting if it's true I need to make sure that I create threads with unique IDs. Simply so I don't try to create a thread which already have been created. The thread_data consists of thread_number, low & high (all of which are int). The variable thread_number is used to set the child IDs, which are set to thread_number*2 + 1 for the child thread created to the left of pivot and thread_number*2 + 2 for the other child thread. By creating the threads and giving them thread_number*2 + 1 or + 2 I always end up with the same tree shown in the figure below and with the children at the same place!

This way of creating each child thread only works for binary trees and creates a special case if only 1 thread is to be created (unless some logic is to be changed).
If you would use the code which was used for 8 threads and then just set NR_OF_THREADS to 1 then 2 threads would be created in total anyway. In order to use 1 thread in a simple way I instead set NR_OF_THREADS = 0 and both the struct & thread array have the size 1 (instead of using the NR_OF_THREADS to set their size). This way only the root thread is created and in thread_quick_sort it goes directly into the quick_sort functions and starts sorting.