1 Exercise #1: Basic Parallelism

In this first exercise you should explore different interfaces for parallel programming in native programming-languages. You will also analyze the runtime characteristics of your final programs. As a starting point you should use the example provided with the materials for this assignment. This program paints a graphical representation of the approximation of the *Mandelbrot set*[1]. For the purpose of this exercise you can ignore the mathematics behind it. The `mandel(x, y, max)` function calculates an integer value based on its arguments `x,y` up to a maximum value of `max`. With `getRGB()` a nice-looking color can be computed for a pair of the return value and `max`. The example program does this for a range of `x` and `y` values and prints out the final image in PPM[2] format.

1.1 Make It Parallel

Make yourself familiar with the code and identify the region where the iteration values of the *Mandelbrot set* are computed. Create multiple versions of this code and make them calculate the values in parallel. Use at least the following interfaces, one for each version of your program:

- Pthreads
- Cilk
- `executor_service` (see below)

Additional interfaces that you can also try:

- OpenMP
- `fork()`
- `clone()` C-library wrapper around the Linux system-call

1.2 Measure

Measure how your parallel solutions scale in performance with respect to the number of threads used. Create plots for each solution.
Measure each data point several times to compensate for fluctuations. Set the number of iterations high enough to run the computation for several seconds. Measure the whole-program running-time using the *time* command. Be sure to also increase the number of threads far beyond the number of CPUs on your test system.
With the help of *Amdahl's Law*, calculate the percentage of the serial and parallel portions of your programs based on the execution numbers for 1 and 2 CPUs. Compare the data and draw conclusions for which parallelization API you could achieve the highest performance. What effects can you see when the number of threads exceeds the number of CPUs? What are the differences between, for example, Pthreads and the executor service? What are the main differences between the APIs you tried for this assignment?

1.3 Executor Service

The executor service is a custom library for lightweight parallelization. It aims to support thousands of jobs processed in parallel. In the course of the semester, we will extend this module to a fully-featured thread library. For this assignment, you implement a basic variant of a thread pool executor service with just 3 functions:

- `executor_service *executor_service_new(unsigned int numthreads)` creates a new executor service. The function takes the number of worker threads as parameter. For convenience, the user can pass the special value 0 to let the implementation choose a suitable degree of parallelism.
- `int executor_service_run(executor_service *es, void (*func)(void *), void *arg)` submits the function call `func(arg)` to the executor service.
- `void exectuor_service_end(executor_service *es)` waits until all submitted jobs have completed, and frees up resources used by the executor service.

Besides, implement a test program for the executor service. Can the submitted job functions call pthread functions, such as `pthread_mutex_lock`? What happens when such a job function blocks?

Remarks:

- Submission deadline: 2018-11-05, 14:00

[1] `https://en.wikipedia.org/wiki/Mandelbrot_set`
[2] `https://en.wikipedia.org/wiki/Netpbm_format`