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
- LWT (see below)

Additional interfaces that would yield similar results as Pthreads (but certainly you can also try these):

- C++11 threads
- OpenMP
- `clone()` C-library wrapper around the Linux system-call

Be sure to also increase the number of threads far beyond the number of CPUs on your test system.

1.2 Measure

Measure how your solutions scale in performance with respect to the number of threads used. Create plots for each solution and 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. 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 LWT? What are the main differences between `fork`, MPI and the APIs you tried for this assignment?

1.3 LWT

Lightweight Threads (LWT) is a custom threading library. It aims to support thousands of threads running in parallel. In the course of this exercise you will extend this library. The LWT interface is kept in C but its implementation is in C++. For this assignment you get a very basic variant of LWT with just 3 functions:

- `int lwt_init()` must be called before all other LWT functions to initialise the library. A return value of 0 indicates success.
- `int lwt_run(void (*func)(void *arg1, void *arg2))` starts the function call `func(arg1, arg2)` as an asynchronous thread. A return value of 0 indicates success.
- `void lwt_destroy()` waits for all currently running threads to finish and frees up resources used by the library.

After comparing the different threading APIs and their implementations, make yourself familiar with the implementation of LWT. What kinds of threads are managed by the library? Try the small *TokenRing* test program provided with the materials, it implements a kind of token ring where an integer is passed between threads and is incremented at each thread. If a thread does not have the token, it waits using a Pthread semaphore. Why does this example deadlock? Would it also deadlock if ordinary Pthreads would be created in each `lwt_run()` call?

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