Thread pool-assisted image processing

Background

A thread pool is a facility where a number of threads continuously pull and execute work from a queue of client-supplied work items. Thread pools are both flexible and efficient: any desired number of worker threads can be used, and threads are constantly kept alive (idling when no work is available), cutting the cost for creation and destruction. Just like other "pool" patterns, such as memory pools and object pools, the thread pool pattern is generic with respect to the underlying object, i.e. work item signatures. Although thread pools have been around for long in C++, it was not until the introduction of C++11, and concepts such as variadic templates that came with it, that this generality became feasible in practice.

The application in this particular lab is image processing with filters, which is a problem well suited for parallelization. See Fig. 1.

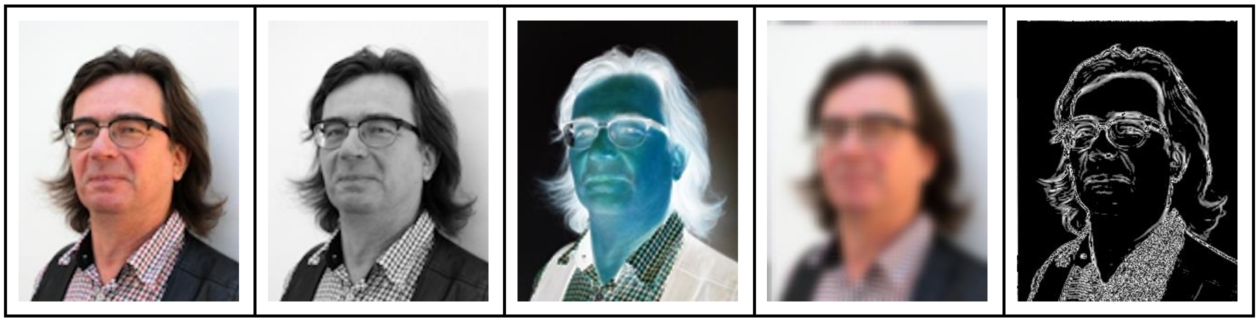

Fig. 1: Filters supported in the lab code (from the left): unfiltered, grayscale, invert, Gauss blur and Sobel. The blur and Sobel filters works by convolving a region around each pixel with a filter-specific kernel. Source image courtesy of Olle Lindeberg.

Instructions

Implement a **thread pool** in well formed C++11-notation. The thread pool should accept (enqueue) *work items* (functions, or general callables) with a **general signature**

with respect to arguments and return type. Results should be provided to the client via `std::future` objects.
The thread pool should work for (but not be limited to) the supplemented image processing example.

Thread pool sketch

A thread pool has the following basic structure:

Members

- Container with threads (`thread`)
- Queue with work items (`function`)
- Stuff needed to synchronize access to the work item queue.
- Stuff needed to handle state (stoppage).

Construction

- Launch a (user-provided) number of threads. A thread should be able to 1) pick a work item, or wait if either none is available or if the thread pool is stopped, and 2) execute the work item.

`enqueue(...)`
Variadic function that takes a work item and a parameter pack as arguments.

- The work item & parameter pack are wrapped in a `packaged_task` (see below for a hint on this) and added to the work item queue.
- `enqueue` returns a `future` object, obtained from the `packaged_task`.

Hint: The main work item wrapper is a `packaged_task`, however in order to have it stay alive long enough for results to be retrieved by the client, it has to be dynamically allocated. This is best done using a shared pointer, for example:

```
auto task = std::make_shared
     < std::packaged_task<return_type()> >
     (
          std::bind(std::forward<F>(f),
                    std::forward<Args>(args)...)
     );
```

.

Here, `return_type` is the type of the provided work function, defined by, for example, `F f` and `Args... args`. This pointer can be made into a callable work item via a lambda:

```
[task](){ (*task)(); }
```

Destruction
Stop and join all threads.

Preparations

- Preceding lectures
- Related readings in *Primer* (see instructions on preceding lectures).
- Online sources such as *cppreference.com* for details on new methods not covered in-depth in *Primer* or elsewhere, e.g. `packaged_task`, `future`, `mutex`, `condition_variable`, `lock_guard` and `unique_lock`.

You are expected to be familiar with basic concepts in concurrent programming.

Test program: Image processing

The supplemented test program utilizes a thread pool to apply a variety of filters to a PNG-image – see Fig. 1. Parallelization is made with respect to *tiles*, i.e. each work item is assigned a rectangular sub-area of the image. The test program includes code for image decoding & encoding, tile generation, filters etc – everything except the thread pool itself, which is your job to design.

The test program is configured via a number of precompiler definitions – see **config.h** for information about what they are and what they do. The first thing you need to do here is to enter paths to in- and output png images (the output image is generated).

Guidance

- The test program makes a few assumptions on how to use the thread pool (see the code). It is fine to make alterations here, i.e. alter the test program to fit your particular thread pool implementation, as long as the instructions made previously are followed.
- To make development easier:

    - Use the smaller image (PNG encoding alone can take several seconds on large images).
    - Build and run in Debug mode.
    - Use less expensive filters, such as grayscale and invert.
- When testing the thread pool's performance:
    - Use the 4K image (much more work to do).
    - Build and run in Release mode.
    - Also try the more expensive filters, i.e. Sobel and blur with a large kernel.

Experiments

These are broader questions not specifically about course topics. You are nevertheless encouraged to discuss them upon demonstration.

- How many cores does your current machine have? How many "hardware threads" are supported (this is queried by the test program)?
- Vary the number of threads and examine how performance is affected. How does performance scale with the number of threads? What is ideal?
- How is performance affected by tile size? Which tile size is optimal? Why?