CS4204 – Concurrency and Multi-core Architectures

Practical 2: Parallel Patterns

Chris Brown
cmb21@st-andrews.ac.uk

2021

Weighting: 50% of coursework

Deadline: 15th April 2021 at 21:00

Purpose

This practical will help develop your skills in:

- Implementing parallel patterns
- Implementing queues
- Use of pthreads to create parallelism
- Evaluation and experimentation of parallel execution

Overview and Background

You are a newly employed software developer of *ParallelSoft Ltd.*, who specialise in writing parallel software. You have been employed to develop a new parallel pattern library for C/C++ using pthreads called *para-pat*.

You are required to design the library so that it is easily callable by the user, with well-defined interfaces. The user must be able to use the library to parallelise their C code. This might require passing a function pointer to a worker function, for example. You will need to make use of pthreads, locking mechanisms and queues to implement the parallel pattern library, and provide a high-level interface so that the pattern is easily callable by passing in some parameters, as seen in the lecture slides on pattern implementation.

Providing an implementation that simply wraps an already existing parallel framework, such as FastFlow or TBB will not be permitted.

In *pseudo-code* a farm skeleton template/interface may look something like this:

```
output = Farm (4, Worker(), input);
```

Where `4` is the number of worker instantiations, and `Worker()` represents the worker function to be executed in parallel. `input` represents the input queue and `output` represents the output queue.

Similarly, a pipeline template/interface may look something like this:

```
Output = Pipe (Stage1(), Stage2(), Stage3(), input);
```

Here, `Stage1()`, `Stage2()`, `Stage3()` represent stages of the pipeline.

An example of nesting a farm inside a pipeline may have the following example interface/instantiation:

```
Output = Pipe( Stage1(), Farm(4, Stage2()), Stage3(), input);
```

Please note that these interfaces are just conceptual to offer an idea. They are not meant as an actual programmable interface. The pattern library should hide away the thread creation, task queues, locking, etc., from the user and provide a high-level skeletal interface. The choice of the design of the interface is left to you.

Examples

In the examples folder you will find two examples that you can use to test your parallel pattern library.

`example.c`

Compile with: `gcc example.c -o example`

This is a simple example that simulates a simple composition of functions. `payload1` and `payload2` are just functions that use Fibonacci to "burn" CPU cycles and simulate "payloads". Try using different parameters to fib to increase and decrease the granularity of the computations.

`convolution.cpp`

Compile with: `g++ convolution.cpp -lpng`

This is a simple convolution algorithm. It requires `libpng` to be installed in order to compile. There is a parallel region in the main function.

In the directory images, you can generate more images using the `create_inputs` script. For example, `create_inputs 20` will generate 20 images. You can modify convolution to process a different number of images, by modifying this line:

```
nr_images = 20 ;
```

For both of these examples, you must experiment with different configurations of skeletons and nesting of skeletons to explore the different performance characteristics.

Performance Analysis

Parallel programming is all about achieving good performance. In this practical, you will analyse the performance of your parallel implementation on the examples provided. To do this, you can use the `get_current_time()` method that is defined in `<sys/time.h>`:

```
#include <sys/time.h>

..

double beginning = get_current_time();

parallel region

double end = get_current_time();

cout << "Runtime is " << end - beginning << endl;
```

The performance analysis in the report should include graphs of runtimes of the examples on a different number of cores. You can vary the input size and plot different graphs to show the difference. The report must also show some runtime analysis of different input number/sizes to explore the performance characteristics of the granularity effect on the parallelism.

Tasks

1. (Easy) Implement the parallel farm pattern. The farm must allow the user to specify the number of workers and the worker function to be executed in parallel.
2. (Medium) Implement the parallel pipeline pattern. The pipeline must allow the user to declare a number of stages that themselves call user-defined pipeline functions.
3. (Hard) Implement nested parallel patterns. Modify your farm and pipeline patterns so that they can themselves take patterns. For example, a farm may take a pipeline as a worker; or, similarly, a pipeline may itself take instances of farms as workers.
4. Evaluate your parallel patterns against the supplied examples. Record your performance results for different configurations or nesting of patterns if applicable to the example. Explain in your report which configuration gives the best results and why.

Submission

Your implementation must be in C/C++ using pthreads.

You are required to hand in, on MMS, a zip file containing the sources of your *para-pat* implementations, your parallelised examples and a report of no more than 1500 words. The submission should contain clear and easy instructions on how to install, compile and run your implementations.

In your report, you must explain your key design decisions, explain how your library can be used, and how it fulfils the above tasks. The report must include an evaluation section, which shows the performance results of your implementations against the examples.

Marking

I am looking for:

- Good design and understandable code, which is well documented, with major design decisions explained;
- A study of the performance characteristics of your parallel pattern implementations;
- A report showing an evaluation section giving the trade-offs of different pattern configurations.

The standard mark descriptors in the School Student Handbook will apply:
https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors

Lateness

The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof):

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good Academic Practice

The University policy on Good Academic Practice applies:
https://www.st-andrews.ac.uk/students/rules/academicpractice/