CS4204 – Concurrency and Multi-core Architectures

Practical 2: Divide & Conquer

Chris Brown & Susmit Sarkar
cmb21@st-andrews.ac.uk
Susmit.Sarkar@st-andrews.ac.uk

2020

Weighting: 50% of coursework

Deadline: 8th May 2020 at 21:00

You are expected to have read and understood all the information in this specification at least a week before the deadline. You must contact the lecturer(s) regarding any queries well in advance of the deadline.

Purpose

This practical will help develop your skills in:

- Programming divide-and-conquer parallelism;
- Task-based parallelism;
- Work-stealing;
- Load-balancing.

Overview and Background

Divide and Conquer is a classic task-based parallelism technique. A divide and conquer algorithm comprises three stages: a *divide stage*, a *solve stage*, and a *conquer stage*. The *divide* stage divides the input into N sub-tasks; the *conquer stage* applies the divide and conquer *recursively* to each of the N sub-tasks, in parallel; the *solve* stage applies some solve function, *f*, to each sub-task. Each divide stage must keep track of its sub-tasks, as some D&C instantiations require parent-tasks to be dependent on their (solved) sub-tasks. A Divide and Conquer skeleton is typically instantiated with 4 parameters.

A `Divide()` function, which takes a task, and divides it into several subtasks;
A `Combine()` function, which takes several subtasks, and combines them;
A `Base()` function, which returns a base value if some base condition is met;
A `Threshold()` function, which acts like a switch to turn the parallelism *off* if some threshold condition is met.

In *pseudo-code,* a Divide and Conquer skeleton template/interface may look something like this:

```
DC(
      Divide(),
      Combine(),
      Base(),
      Threshold()
)
```

If we look at the example of Fibonacci, we may define its sequential algorithm like the following:

```
unsigned int Fib( unsigned int n ) {

   if ( n < 2 ) return 1;

   return Fib(n-1) + Fib(n-2);

}
```

A Divide and Conquer equivalent may replace the sequential Fibonacci with a call to some `DC` skeleton (in pseudo-code); for `Threshold`, we simply call the sequential `Fib` if we drop below the threshold point, `T`:

```
DC (

      Divide(task){ return [task-1, task-2]; },

      Combine([task1, task2]) { return (task1 + task2); },

      Base(task) { if ( task < 2) return 1; },

      Threshold(task, T) { if (task < T) return (Fib(task));}

)
```

Tasks

Your task is to implement a parallel divide & conquer skeleton implementing a task-based parallelism model using pthreads, as seen in the lectures, which has load-balancing and random work-stealing. You will need to use the ideas in the lectures on task-based parallelism, (non-shared) task and worker pools, load balancing and work stealing. Tasks must keep track of their dependencies in a true Divide and Conquer fashion. Your solution must be implemented in C/C++ (no other languages are permitted).

Analyse and evaluate your implementation on the supplied set of benchmarks (pfib, mergesort, qsort), with a variety of different inputs (the choice/size of inputs, thresholding, task sizes, etc. are left up to you).

As possible **going further** tasks, you may want to think about comparisons with similar implementations using OpenMP, TBB, GrPPI, etc.

Submission

You should hand in the sources of your implementations, together with any recipes needed to build them. In your report, briefly discuss key design decisions, any problems or unexpected features you encountered, and the performance results of your implementations. Make a **zip archive** of all of the above and submit via MMS by the deadline.

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 divide & conquer implementations;
- A report giving a critical analysis of the design and applicability of the skeleton to the pfib, mergesort, and qsort examples, for a range of input sizes.

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
In particular, a very basic implementation of a parallel divide and conquer for one example with very basic load balancing and work stealing would fit the definition of a reasonable attempt achieving some of the required functionality, and could get marks up to 10. More than that would need work on analysis and critical thinking of the algorithms. As usual, marks of 19 or 20 would need an exceptional solution with demonstrated insight into the problem.

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/