# Concurrent & Parallel Systems - Coursework 1 Report

Glenn Wilkie-Sullivan - 40208762

October 13, 2018

**Abstract**

This report will detail a project attempting to add concurrency and parallelism within a block-chain simulator in C++. The project in question will utilise methods such as multi-threading, algorithmic skeletons, CPU-level parallelism and OpenMP (Open Multi-Processing) to optimise the simulator as much as possible.

## 1 Introduction and Background

## 2 Initial Analysis

In my analysis of the application, I used a laptop with the following specifications:

- Intel Core i7-6700HQ @ 2.60GHz (4 cores, 8 threads)
- 8GB RAM (DDR4)

To analyse the performance and overall CPU usage of the blockchain application, I used the Visual Studio 2017 diagnostic tools - all within Release mode, x86. To start with, I ran the application to examine the overall performance, as well as any potential bottlenecks in the code. Unsurprisingly, the performance of the application hinges mainly on the 'difficulty' variable, which the programmer can control. After 5 runs, the application, with difficulty 5, mined 5 blocks in about $43 \pm 10$ seconds. With difficulty 6, the application mined 5 blocks in about $382 \pm 30$ seconds.
As for bottlenecking, the application has a few instances of it. The most effective method of finding the bottleneck was to use Visual Studio's debugging tools, between a range of difficulties. The results after running the application with difficulty 6 was as follows:

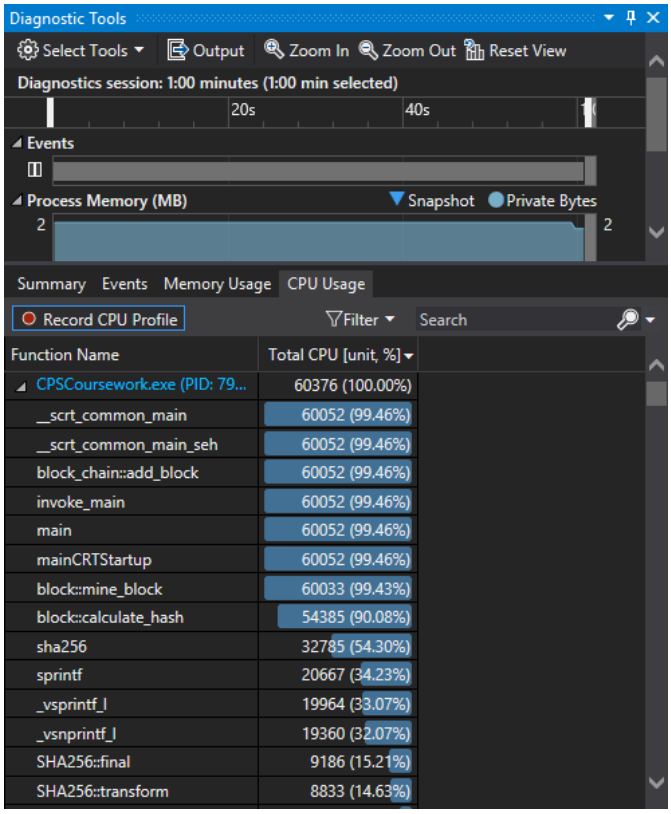


Figure 1: Difficulty 6 CPU Usage

From this image, I was able to delve into each individual method (add_block, mine_block, calculate_hash), which let me see the CPU usage for each line of code. The add_block method runs the mine_block method, which then runs the calculate_hash method. Within this, there is a hashing function which seems to be the main usage of CPU time, called 'SHA256'. After using the diagnostic tools, the following CPU usage percentages were shown:

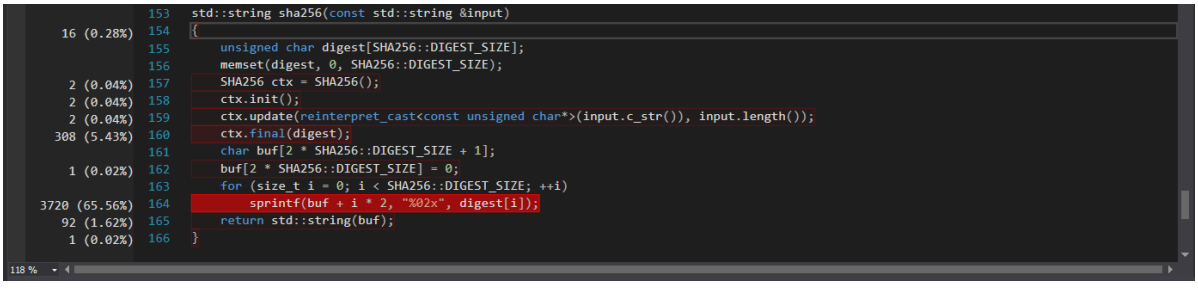


Figure 2: SHA256 Function CPU Usage

As it is called within the main while loop of calculate_hash, this line seems to be creating, appending and concatenating a hash-code until it is compared to a specific string, which is passed as a termination condition in the while loop. In order to understand how many times this hashing function is called, I simply added a value which will count the amount

of times the function runs. Given that 1000 blocks are mined, the application had to be ran at lower difficulties to test this. The following results were found:

- Difficulty 1 (base): 15,719 iterations,
- Difficulty 2: 253,078 iterations (1,510% increase from base),
- Difficulty 3: 3,989,869 iterations (1,476.5% increase from base),
- Difficulty 4: 62,918,025 iterations (1,476.9% increase from difficulty 3).

From this, we can safely say that each step in difficulty will have the same % increase. Based on these exponential values, it is clear why a bottleneck occurs within this function. Using this % increase, we can estimate a mining of 1000 blocks at difficulty 5 to have around 991,902,664 iterations. At difficulty 6, 15,637,345,497 iterations. Assuming this function can be ran multiple times concurrently, that theoretically should decrease execution time by a substantial amount - however, creating threads specifically for this function may be costly itself. Another line worth considering is the initialisation of the stringstream which is passed as a parameter to the hashing algorithm. As seen below,


Figure 3: stringstream Initialisation CPU Usage

The stringstream initialisation uses 20% of the CPU resource, as opposed to the hashing function's 64% - this is likely due to the amount of times the 'stringstream' object is being initialised. Moving the object constructor out of this method would likely increase the program speedup somewhat.