Concurrent & Parallel Systems - Coursework 1 Report

Glenn Wilkie-Sullivan - 40208762

October 26, 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

This project will be analysing, parallelising and optimising a block-chain simulator using the SHA-256 hashing algorithm. According to investopedia, block-chain is, "primarily used to verify transactions, within digital currencies though it is possible to digitize, code and insert practically any document into the blockchain". Each block contains a hash of the previous block, some form of timestamp and data associated to the task, and each block is connected together in a chain resembling a tree-structure. 'SHA, or **S**ecure **H**ashing **A**lgorithm', is a form of cryptographic hash function used to create unique signatures for a data file. In this instance, the SHA-256 hashing algorithm is assigning an almost-unique 256-bit (32-byte) signature for each block.

2 Initial Analysis

In my analysis of the application, I used the following processor:

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

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 ± 10 seconds. With difficulty 6, the application mined 5 blocks in about 382 ± 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

Function	CPU Unit Usage	CPU Usage %
block_chain::add_block()	60052	99.46%
block::mine_block()	60033	99.43%
block::calculate_hash()	54385	90.08%
sha256	32785	54.30%
sprintf	20667	34.23%

Table 1: Individual Function CPU Usage

difficulties. The results after running the application with difficulty 6 was as follows:

From these values, 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, the SHA-256 hashing function seems to be the main usage of CPU time. After using the diagnostic tools, the following CPU usage percentages were shown:

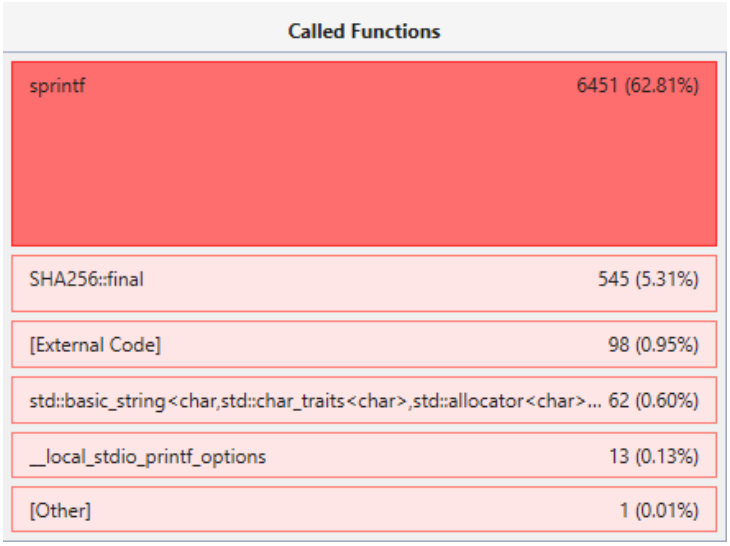

Figure 1: SHA256 Function CPU Usage

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	Iterations	% increase from previous difficulty
1	15,719	N/A
2	253,078	1,510%
3	3,989,869	1,476.5%
4	62,918,025	1,476.9%

Table 2: Hashing Iterations

From this, we can safely say that each step in difficulty will have roughly 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, the following CPU usage for each line can be seen,

Function	CPU Usage %
stringstream initialisation	$\approx 1.82\%$
stringstream concatenation	$\approx 20.00\%$
sha256 hashing	$\approx 64.13\%$
Other/Tertiary Processes	≈ 10 - 14%

Table 3: Individual Line 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. This is the main bottleneck of the application - no other function/method comes close to the amount of CPU resource that the hashing algorithm uses.

3 Methodology

Due to its overall effect on the application, we can safely assume that editing/optimising the stringstream initialisation and hashing algorithm should improve the efficiency and speedup in multiple areas of the program. Parallelising the algorithm should be the most

effective method of optimisation, and there are multiple string functions which are better suited, performance wise, than stringstreams - such as string::reserve and string::append. These functions can request a change in size to our hashcode, then append the necessary information respectively.
To start with, it would be beneficial to make small changes to the way the program operates - there are superfluous print statements, some values can be placed in the header file(s), and some functions can be edited slightly. In order to measure the speedup and efficiency, I'll have to serialise the times of the mined blocks, and see if any noticable change in time occurs before and after the changes are made. To do this, the time value can be placed into a CSV file for each block, and we can use R to draw comparison between two datasets. We can do this for all methods of optimisation in the future.
Next, we can use threads and OpenMP to parallelise the hashing algorithm, such that there are hashing processes running on 8 threads (relative to CPU) instead of running them sequentially. After examining the results of a successful or non-successful parallelisation, we must consider the other utilities offered from parallelisation. When creating threads for our parallisation, it may prove useful to use a mutex and atomics to ensure the consistency of our hashcodes. We can ensure that only one thread is operating within the hashcode generation/concatenation to avoid deadlock, and use atomic functions to make sure the hashcode isn't split during the process execution(s). Outside of the hashing algorithm, the application may benefit from parallel for loops as opposed to generic for/while loops. Since we either know or can approximate how many times a loop will run in this application, it would be a good idea to parallelise these processes. Likewise, we can use scheduling to divide work in our parallel loops - especially since we have a predetermined amount of blocks to mine.
Once these features are in place, we can measure efficiency and speedup using a table similar to this,

Feature	Time Before Implementation	Time After Implementation
string::append	x seconds	y seconds
Hashing Parallelisation	x seconds	y seconds
General Parallelisation	x seconds	y seconds

Table 4: Optimisation Approaches

4 Results and Discussion

In order to examine the speedup and efficiency for the application, there had to be some way to compare the mining time for different methodologies. With that in mind, there had to be separate Visual Studio projects with the original scripts, and every methodology

used for this project. However, all of these projects had to have functions which serialised the average time to mine blocks, as well as the difficulty. This can then be visualised and examined with a simple R script, which simply reads in the data, transforms it into a data frame and plots it using the ggplot2 package. Any results seen in this section won't be influenced by background processes or other programs - these were closed/stopped. To examine the application, the methods used were: thread pools, multithreading and OpenMP. The average speed for a block to be mined was compared against the speed of the original scripts, which produced the following graph:

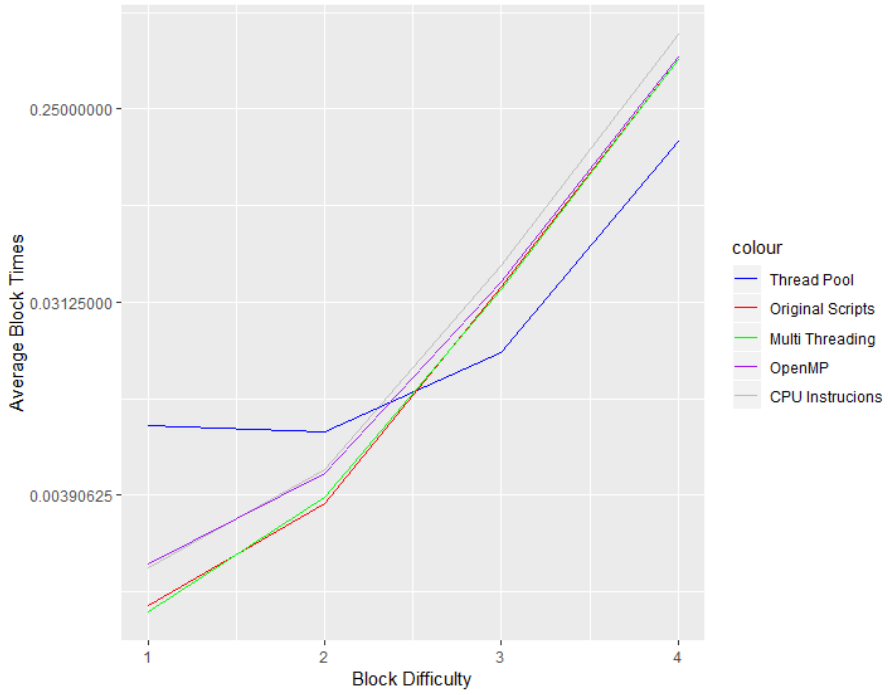

Figure 2: Average Block Mining Time Between Methods

To start with, we must examine the multithreading method. This entails creating n amount of threads, relative to the CPU, and having the threads work on facets of the application in parallel. In order to achieve this, we have to replace a costly, sequential function with a purely threaded method of execution - the chosen function to replace was the calculate hash function. As mentioned previously, this can be called millions, possibly even billions of times at higher difficulties. Having a threaded method to execute this would speed up the application considerably. Upon replacing the function, the following results were found:

Difficulty	Time Taken (Sequential)	Time Taken (Threaded)	Speedup
1	0.222963 seconds	0.743006 seconds	N/A
2	0.444432 seconds	0.751788 seconds	N/A
3	4.66663 seconds	1.80038 seconds	2.59202
4	57.9695 seconds	15.6473 seconds	3.70476
5	933.004 seconds	207.502 seconds	4.49636

Table 5: Performance Difference (Sequential vs. Threaded

From this table, we can see a 4.4 total speedup in favour of the multithreading technique, with an efficiency of 0.999987.

5 Conclusion

6 References