Analysis and evaluation of techniques of parallelising a blockchain simulation

Svetlozar Georgiev

40203970@live.napier.ac.uk

Edinburgh Napier University - Concurrent and Parallel Systems (SET10108)

Abstract

The aim of this report is to describe attempts at improving the performance of a blockchain simulation using CPU-based concurrency methods. The techniques used were: manual multithreading, OpenMP and `std::future`s.

1 Introduction and Background

The provided sequential program works as follows:

1. A loop creates a set number of block objects. Certain information is stored in each block: its index, the time of its creation (to ensure the output of the program is always the same, this value is the same as the index), specific block data (a `string`) and the hash of the previous block in the chain.
2. A newly created block then starts "mining" a hash. This is achieved by incrementing a counter in a loop. Each time the counter is incremented, a string is created by joining all the information stored in the block and the counter. This string is then converted to a hash using the SHA256 encryption algorithm.
3. It is then evaluated whether this is the correct hash. This is based on a variable called *difficulty*. It determines how many zeroes have to be at the start of a hash.
4. If the correct hash is found, the program creates a new block and repeats the process. If the hash is incorrect, the program continues to increment the counter variable and generates new hashes until the correct one is found.

All of the tests described in the next sections were run on the same computer with specifications listed in **Table 1**.

Table 1: *PC specifications*

CPU	Intel Core i5-8300H
GPU	NVidia GTX 1050 Ti 4GB
RAM	8 GB
OS	Windows 10 Pro x64

Additionally, all test programs were compiled in Release mode with compiler optimisations using the *MSVC* compiler. All programs were run with 8 threads.

2 Initial Analysis

To analyse the initial program, the Visual Studio Profiler was used. Its CPU Sampling option provides detailed information about the resources each part of a program uses. The generated report can be used to easily spot the heaviest parts of the program. If they can be parallelised, the overall performance could be greatly improved.

Results

The results of the performance analysis showed that the most CPU intensive functions were:

1. `block::mine_block():` The total CPU time of this function is 84.72%. It includes a while loop which generates hashes until the correct hash is found. Each iteration of the loop is independent from the others so it could be a potential part of the program that could benefit from parallelisation. However, upon further investigation, it can be noticed that the call to `calculate_hash()` is actually the most intensive part of this function.

2. `block::calculate_hash():` This function returns a hash based on the information stored in the current block. All the information is concatenated into a single string and then passed to the `sha256()` function. The concatenation is achieved through a `stringstream` which can be optimised (not using parallel techniques). However, the most CPU-intensive part of this function is the call to `sha256()` (68.37% CPU time).

3. `sha256():` This function converts the passed string to a *SHA256 hash*. The use of `sprintf()` is highlighted as very CPU-intensive by the profiler (60% CPU time). Manual conversion can be implemented instead. Furthermore, the use of a for loop means *OpenMP* can potentially be used. Additionally, SIMD instructions seem suitable to optimise the hashing algorithm.

4. `main.cpp:` In the main file of the application, a loop creates a certain number of `block` objects. This could be a potential area that could be parallelised. However, every time a new block is created, it requires the hash of the previously mined block. This means there is a dependency between the elements of the blockchain. Parallelising such a data structure could be a difficult task. Sometimes this could result in the same performance as the initial sequential program as threads my have to wait for each other and the program essentially becomes sequential.

To test the effect the `difficulty` variable has on the speed of mining blocks, the program was modified to mine the first block with difficulty from 1 to 6. The results can be seen in **Figure 2**. As it turns out, increasing the difficulty, increases the number of iterations (i.e. how many hashes have to be calculated before the right one is found) exponentially.

To test the effect of the block data on performance, another modification was made to the program. A function that generates a random string of a given length [1] was added. A random alpha-numeric (including both lower and upper case characters) string containing special characters (`!£$%^&*()-=+'#~`) is generated. A pseudo-random generator is used (`rand()`) to produce the same data every time the program is run so that consistent results can be collected. The first ten blocks were mined with difficulty set to 3. The program was run 10 times and the length of the data was increased by 10 every run. The results are shown in **Figure 1**. As it can be seen, the length of the string does not seem to have a direct effect on the time it takes to mine the first 10 blocks. It is more likely that the specific letters or special characters affect the performance.

3 Methodology

3.1 Non-parallel

3.1.1 std::stringstream

As shown by the Visual Studio Profiler, the use of `std::stringstream` is quite inefficient. To find a suitable and faster replacement, a test program was written to compare the other string concatenation methods [2] [3]. The average results of 100 runs are shown in **Table 2**.

Table 2: *Results of string concatenation comparison.*

Concatenation method	Average time
operator+	6.02902 ms
operator+=	1.201189 ms
append()	1.18238 ms
std::stringstream	43.6603

As it can be seen, `operator+=` and `append()` are the fastest, `append()` being slightly faster.

3.1.2 sprintf()

Another CPU-intensive part of the program is the `sha256::sha256()` function. It uses the `SHA256 algorithm` to generate a hash based on an input `std::string`. `sprintf()` is used in it to convert an array of `unsigned chars` generated by the algorithm to their Hexadecimal string representation. This can be done in a faster and safer [4] method. One such method is the use of a look-up table and manual conversion [5] using a `Bitwise AND (&)` and a right shift. This change not only improves the performance of the sequential program, but it enables the parallelisation of the loop where the change was made with *OpenMP*.

3.2 Multithreading

Multithreading provides the ability for a single system to perform multiple independent activities in parallel, rather than sequentially, or one after the other [6].

After the initial analysis, it became obvious that the function `block::mine_block()` could be parallelised. The first attempt made was using multiple threads to generate a number of hashes simultaneously. This is an embarrassingly parallel problem as the hashes do not depend on each other. The only variable that could cause a problem was `_nonce` (the number of iterations) as each thread has to increment it. There are a number of possible solutions, such as using an `std::mutex` (or any of the mutex wrappers, e.g. `std::lock_guard`), atomics, etc. The selected solution was changing `_nonce` to an atomic variable. Atomic variables ensure well-defined behaviour when accessed from different threads simultaneously. For example, if one thread writes to an atomic object while another thread reads from it, no race conditions occur [7]. The next step was creating a lambda function to be run in each thread. The function contains a while loop which calculates a hash and assigns it to a temporary variable. It is then checked whether the hash contains the required number of leading zeroes (based on the `_difficulty` variable). If it does, a boolean variable (also `atomic`) is set to true. This is how the while loop is stopped. The same boolean variable is shared between all threads so they can stop working once one thread finds the correct hash.

3.3 OpenMP

`OpenMP` is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs [8]. It provides an easier way of parallelising parts of a program without having to manually set up and manage threads. It greatly speeds up development time and is much simpler than manually creating and managing threads.

`OpenMP` provides an extremely simple method of parallelising `for loops` by adding the `#pragma omp parallel for` directive before the creation of the loop. Using this method would require simply to convert the existing while loop to a for loop. However, for loops are range-based, i.e. it is required to know how many iterations are expected at the time of creation of the loop. Unfortunately, in this case it is impossible to know ahead of time how many iterations will be needed to find the right hash. It is possible to set the number of iterations to a large number and stop the loop when the correct hash is found. However, `OpenMP` does not allow the usage of the `break` statement inside parallel regions.

Instead, a `#pragma omp parallel` directive was used to parallelise the existing while loop. An `std::atomic_bool` was used to control when the loop stops. Similarly to the multithreaded version, the boolean is shared between the threads, so that when one thread sets it, all other threads stop.

`OpenMP` was also used to parallelise the for loop in the `sha256::ssha256()` function. Without the changes to the function described in the previous subsection, the output of the hashes was incorrect: only a part of them was being shown. A solution to this problem was using the schedule statement:

```
#pragma omp parallel for num_threads(8) schedule(↩
    static, 32)
```

However, after removing `sprintf()` (as described above), the output was correct. All changes made to the function were adding a

```
#pragma omp parallel for num_threads(8) schedule(↩
    static, 1)
```

directive before the loop initialisation.

3.4 std::future

Asynchronous functions are executed at the same time as other operations (potentially on a separate thread) and their results are returned as `std::futures` [9]. The use of futures enable a thread to `return` a value and stop running. Unlike the multithreading solution, in this case no manual thread management is required, however the automatic management is expected to be slower.

The changes made to implement this version were very similar to the multithreaded version. A number of `std::futures` is created (based on the available hardware concurrency) using the `std::async()` call. It is explicitly specified that a separate thread should be launched with the `std::launch` argument. Each future runs the same *lambda* function which generates hashes until the right one is found. A shared `std::atomic_bool` controls when the threads stop. If a thread is stopped before it find a hash, it simply returns an empty string. The resulting futures are stored in a vector. It can then be checked if the results stored in the futures are empty. If they are, they are ignored. However, one of the futures contains the correct hash.

4 Results and Discussions

To compare the effects of the optimisations described above, each version was run 10 times. The first 10 blocks were mined with difficulty set to 3, 4, 5 and 6. The times for each implementation were then summed. The results are shown in **Figure 3**. As it can be seen, the manually managed multithreaded implementation is the fastest. However, the difference between its total time and the total time of `OpenMP` version is very small - only 2 seconds. Futures are the next slowest implementation - 15 seconds slower. This is to be expected as they provide an extra layer of abstraction and do not allow any manual management. This introduces extra overhead and is less efficient. The slowest implementation, even slower than the sequential one, is the OpenMP optimisation in the `sha256()` function. This is also to be expected as the total number of iterations in this loop is very low and its CPU time is very low to begin with, meaning that no speedup can be expected. This is due to the fact that more overhead is introduced by the creation and management of threads by `OpenMP`. It should be noted that removing the `OpenMP` directive from this version yields much better results.

Additionally, comparing the data for each of the difficulties separately, instead of summing it, showed interesting results. The total time to mine 10 blocks for each difficulty is shown in **Figure 4**. As it can be seen, when difficulty is set to 3, 4 and 5, `OpenMP` is the fastest version. However, the results of the multithreaded version are very similar and it is only slightly slower. When difficulty is set to 6, the multithreaded version is 2 seconds faster. This is expected for the multithreaded version as the cost of creation of threads and context switching is offset by the amount of data to be processed. However, the speed of *OpenMP* at lower difficulties cannot be explained. A possible reason is the optimisations *OpenMP* does in the background.

Furthermore, the speedup and efficiency values were calculated for each of the test programs. They can be seen in **Figure 5**. As it can be expected from the previous results, the speedup and efficiency values of the `std::thread` version are the highest, followed closely by the *OpenMP* version.

5 Conclusion

To conclude, the blockchain simulation can definitely benefit from parallelisation. As it can be seen from the results of the tests, manual multithreading and `OpenMP` provide the best results - speedup values of over 5. If the aim were to achieve parallelisation in the least amount of development time, `OpenMP` appears to be the best solution. It allows the programmer to parallelise an existing program without having to make too many changes to existing code. If the aim is the best performance, manual multi-threading seems to provide the best results. With fine tuning, it can definitely provide even better speedup and efficiency. The results from using `std::futures` are satisfactory. Due to the automatic management, however,it is to be expected for them to be slower. Lastly, parallelising the for loop in the `sha256()` function did not improve the performance at all due to the low performance impact in the baseline application. On the other hand, changing the method of byte conversion provided noticeable results. It can also be concluded that this program benefits from parallel optimisations at any difficulty and block data.

Possible future work would include the use of `SIMD instructions` to optimise the sha256 hashing algorithm and investigating whether its possible to parallelise the creation of blocks.

References

[1] "How do I create a random alpha-numeric string in C++? - Stack Overflow." https://stackoverflow.com/questions/440133/how-do-i-create-a-random-alpha-numeric-string-in-c/12468109#12468109. (Accessed on 10/27/2018).

[2] "c++ - Most optimized way of concatenation in strings." https://stackoverflow.com/questions/18892281/most-optimized-way-of-concatenation-in-strings/18892355#18892355. [Accessed on 17/10/2018].

[3] "Modified program to compare speed of concatenation." https://gist.github.com/Ligh7bringer/c829b43eef89730e8a1dafe10dfb38af. [Accessed: 17/10/2018].

[4] "Buffer Overruns and Overflows." https://www.owasp.org/index.php/Buffer_Overruns_and_Overflows. [Accessed: 25/10/2018].

[5] "StackOverflow comment by Mat." https://codereview.stackexchange.com/a/78539. [Accessed: 25/10/2018].

[6] A. Williams, *C++ concurrency in action: practical multithreading*. Shelter Island, NY: Manning Publ., 2012.

[7] "std::atomic documentation." https://en.cppreference.com/w/cpp/atomic/atomic. [Accessed: 20/10/2018].

[8] "OpenMP." https://www.openmp.org/about/openmp-faq/#WhatIs. [Accessed: 21/10/2018].

[9] "std::async documentation." https://en.cppreference.com/w/cpp/thread/async. [Accessed: 21/10/2018].

Figure 1: *Relationship between length of data and total time to mine first 10 blocks.*

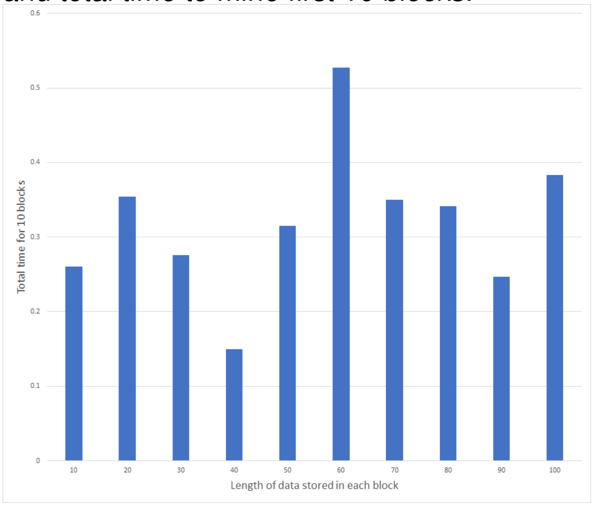


Figure 2: *Relationship between difficulty and number of iterations to mine the first block (in logarithmic scale, base 2)*

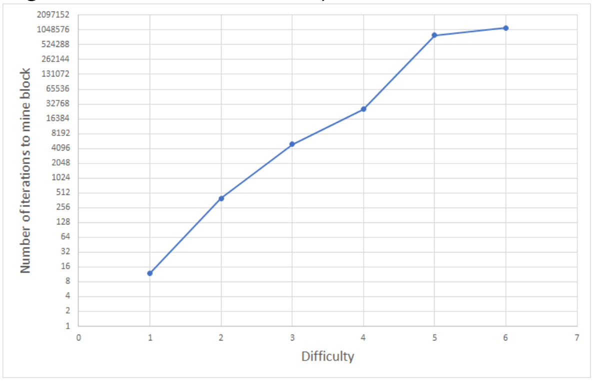


Figure 3: *Comparison of the total time (all difficulties summed) to mine 10 blocks of each version (time in logarithmic scale, base 2)*

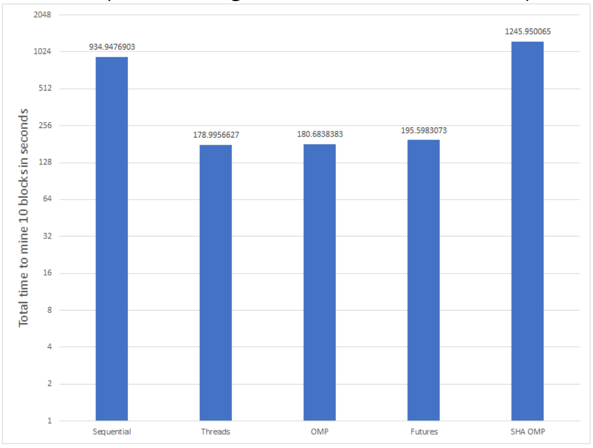


Figure 4: *Comparison of the total time to mine 10 blocks of each version per difficulty (time in logarithmic scale, base 2)*

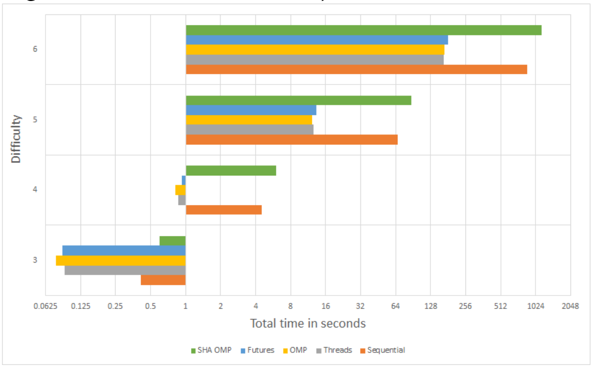
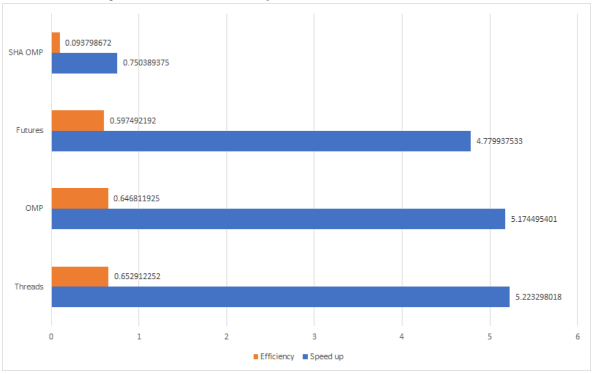


Figure 5: *Comparison of the speedup and efficiency of each implementation*


SHA OMP
Futures
OMP
Threads
Efficiency
Speed up