Parallel Performance Analysis of a Block Chain Algorithm Implementation

Georgi Mirazchiyski

40208393

Concurrent and Parallel systems

SET10108

School of Computing

October 28, 2018

Abstract

The purpose of this report is to present and analyse multiple parallel implementations of a sequential program that simulates a `block-chain` algorithm. The methods covered in the scope of this project are naive `Multi-threading using Atomics`, `Thread-Pool using Futures` and the application programming interface - `OpenMP`.

1 Introduction and Background

The main idea of the project is to compare, analyse and discuss the parallelisation of the algorithm with every one of the mentioned methods with the use of helpful tools and techniques and such as profilers, scoped timers and plotting.
In general, the concept behind block-chain is that it has to operate as distributed database that maintains a continuously growing list of ordered records. It is a technology backing more than a few serious security-oriented projects (i.e. cryptocurrencies such as BitCoin). Intuitively the chain is divided into blacks as the name implies which are formed of at least:

- `Index` - Position of the block in the chain.
- `Timestamp` - Time of creation of the block.
- `Block data` - Data stored in the block.
- `Hash` - Unique signature of the block.
- `Previous Hash` - Hash code of the previous block in the chain.

The reason why we need to store previous hash is really important for the structure of the block-chain as it is used for the creation of a new unique hash for the current block being mined. Once a new hash is generated, the block creation will be finalized and then it will be added onto the chain as last in order.

The hashing algorithm in this program is the famous `SHA-256` that was used extensively for various secure applications in the past years. Nowadays it is still used for generating secure and (almost) fully unique 256-bit hashes for block-chain and cryptocurrency implementations.

2 Initial Analysis

During the very first debugging of the program, I could extract information about how does the algorithm perform with different settings by tuning the mining via a difficulty parameter. For the low difficulty levels the performance seemed reasonable. However, by raising the difficulty with just a little bit definitely makes the algorithm perform much slower and become almost unusable for a bigger chunk of blocks.

Basic debugging and naive analysis on performance result in a limited data to work with towards developing optimized solution, thus I had to do even better and analyse the `IR` (`instructions per call`) with some sort of a profiling tool because scoped timers would simply take very long and may not point out the non-obvious slow parts of the program. Therefore, I used a set of tools based on the `Valgrind` toolset such as `Callgrind` (a branch prediction profiler) integrated with `KCacheGrind` (a front-end for displaying the profiled data). This helped to easily identify the main bottlenecks of the application that are reasonably suitable for optimization through parallelism. `Figure 1` shows output of the intensive bits of the application captured by the profiler.

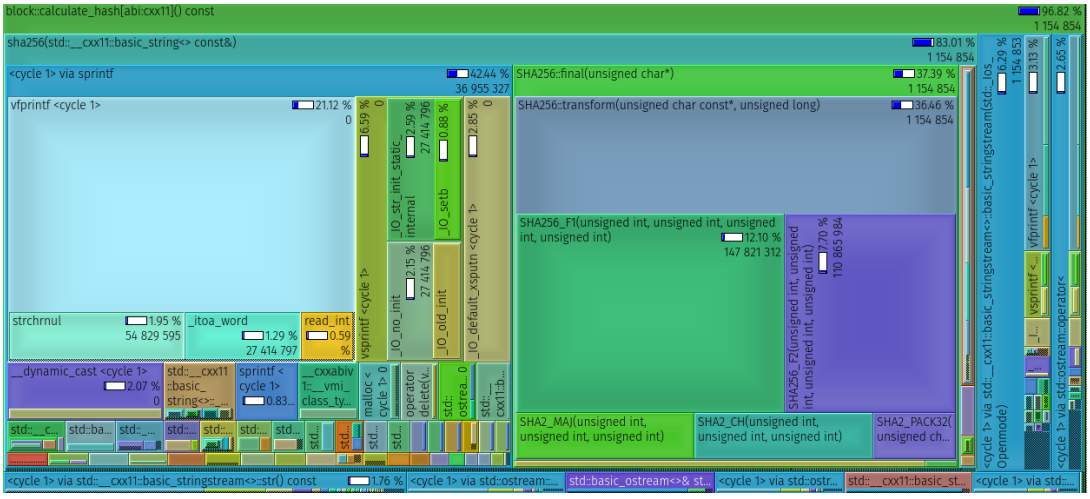

Figure 1: **KCacheGrind profiling tool.** Zoom into the `calculate_hash` function.

It turns out that 100% of the application's workload is just to mine a block and by analysing the figure above we can notice that *approx.* 97% of the time is spent inside the body of the function responsible for calculating the hash code, therefore it is the most intensive computation happening in that algorithm. We can also observe that writing the hash to a

buffer is an intensive operation but that will be too timely to optimised and not as efficient as desired. A good start would be to look into paralellising the `calculate_hash` function. After further investigating this particular function, it becomes apparent that reading and writing to the `_nonce` variable must be synchronized, so no memory access overlap can occur between multiple threads, which can lead to potential data race, thus output a wrong hash.

The baseline performance testing consisted of a variety of configurations of the difficulty parameter. The experiment was made by running 50 iterations for 50 blocks with difficulty level ranging from 1 to 5.

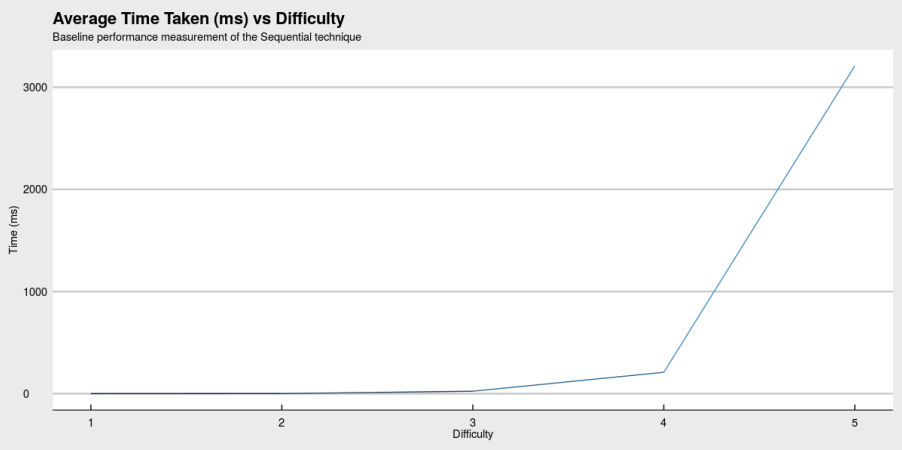

Figure 2: **Sequential benchmark - line graph.** Average Time Taken (ms) vs Difficulty.

By analysing the line curve in `Figure 2`, we can see that the bigger the difficulty is, the more aggressively time consuming becomes the task of mining a block.

Slightly more detailed statistical summary of the data gathered for each difficulty can be seen in the following `Table 1`.

Difficulty	Mean(Avg. Time Taken)	Standard Deviation	Standard Error
1	0.074	0.066	0.009
2	0.814	0.667	0.094
3	23.882	20.778	2.938
4	209.313	209.936	29.689
5	3208.798	2981.439	421.639

Table 1: Sequential Benchmark - `statistical summary`.

3 Methodology

The following specifications were used throughout all benchmarks for consistency.

PC configuration	-
CPU	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
Number of cores	4
Maximum processing units	8 (via Hyper-threading)
RAM	8 GB
Operating System	Linux (Ubuntu 16.04-x86-64)
(Preferred) Threading Library	Pthreads (POSIX threads)
Benchmark configuration	-
Range of difficulty	Levels: 1 to 5
Number of blocks	50
Number of iterations	50
Number of threads: Option 1	8 : max hardware threads
Number of threads: Option 2	2,4,7,7,7 : dynamic based on difficulty

Table 2: PC and Application / Benchmark configurations.

The listing shows how would adding parallelism to the application look like.

Listing 1: Pseudo-code of the new `mine_block()`

```
func mine_block(difficulty, max_difficulty) -> void
    // preferred number of threads : setup
    func thread_count = [difficulty, max_difficulty] -> unsigned int
        // determine the most optimal number
        return count;

    // parallel calculate hash function : definition
    func parallel_calculate_hash = [difficulty] -> void
        str_to_compare = string(difficulty, '0');
        while concurrent_modified_hash == false
            local_hash = calculate_hash_with_concurrent_nonce();
            if local_hash == str_to_compare
                && concurrent_modified_hash == false
                // store calculated value
                concurrent_modified_hash = true;
                hash = local_hash;

    // create threads and assign task : execute
    run(parallel_calculate_hash(thread_count));
```

3.1 OpenMP (Open Multi-Processing)

OpenMP seemed like the go-to technique to begin with since its API is perfectly abstract and allows for easy to achieve high-level parallelism. The idea was as simple as abstracting the `while loop` where the hash calculation happens in a lambda and run this lambda function in parallel. Additionally, the `_nonce` variable was protected with the `#omp critical directive` that is used to identify a section of code that must be executed by a single thread at a time in order to prevent data races. The `while loop` was slightly modified using a flag in order to ensure that the calculated value is correct.

3.2 Multi-threading with Atomics

Logically, the next step was to go slightly on a lower level and personally creating and managing threads to achieve parallelism within the same function. The `_nonce` is crucial in this calculation, thus I have protected it on atomic level as well as making the while flag that decides when a correct hash value has been computed atomic too. As for writing to the global hash of the block, I have further protected it with a `mutex` using scoped `unique_lock`. This implementation approach is based on following the logic of the higher level OpenMP approach but with a different tool set provided to work with, thus the results were satisfying.

3.3 Thread Pool with Futures

This final technique is a well known solution to achieving parallelism for many years. Thread pool consists of task queue, usually implemented via thread-safe queue data structure which I have done by creating a custom thread-safe wrapper for the needed functionality of `std::queue` via the use of `mutex` and `semaphore` to syncrhonise pushing and popping. Now the benefit that thread pool brings is that creating threads manually every time is expensive but having a `"pool"` of ready waiting and self-managing threads abstracted in a little helper class saves us from manually doing the same job repetitively. For the implementation, `packaged_task` and `variadic templates` to returing a future of the desired type when `pool.enqueue(task)` is called. When a task gets pushed to the task list, it notifies all threads that it is ready to be taken and the first available thread will get the signal and will execute it. Parallelising the hash calculation was really trivial, having almost the same function body as in the naive multi-threading approach we discussed above, excluding the need for locking a `mutex` before

writing to the hash as there is no need because of nature of how this technique operates.

4 Results and Discussion

We will discuss the results of the benchmarks with 2 `different` thread number setups.

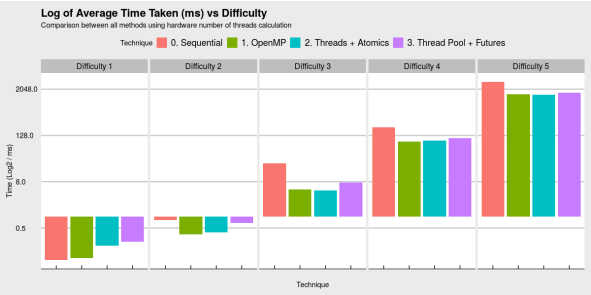

Figure 3: `Log2 Average Time (ms)` chart using `hardware threads`

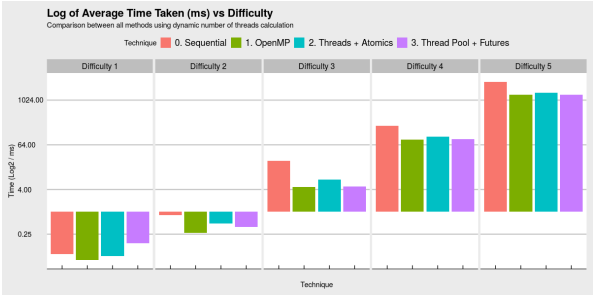

Figure 4: `Log2 Average Time (ms)` chart using `dynamic threads`

The charts above represent the average time taken to mine a single block for each setup.

Starting with `Figure 3`, it is apparent that for `difficulty 1`, none of the techniques are faster than the sequential one. There is no solid trend here, but we can define that for low difficulties `below 3`, the `Thread Pool` technique clearly outperforms the other methods. Whereas for higher ones, both `OpenMP` and `Threads + Atomics` perform similarly fast and also both outperform the `Thread Pool` method.

Now, there are two obvious trends in `Figure 4`. Before pointing them out, surprisingly now the sequential method is way slower in all difficulties and not just 2 and above. Back on defining the trends, `OpenMP` is now the fastest method for the lower difficulties, while from 3 to 5, the Thread Pool is the fastest by outperforming `OpenMP` only a little bit. It is also visible that the the `Threads + Atomics` method is faster than `Thread Pool` on the small difficulties but is the slowest on the bigger ones. Overall with this second setup, all of the tested techniques perform faster than with the first setup.

Next in line is speedup comparison and the following charts represent the speedup factor for each setup based on difficulty and technique used.

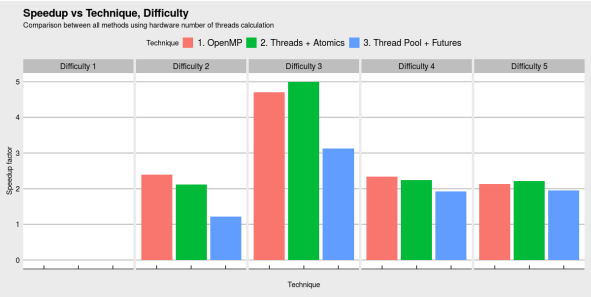

Figure 5: Speedup chart using `hardware threads`

Figure 6: Speedup chart using `dynamic threads`

Starting with `Figure 5`, we already know that there is no speed gains for difficulty 1 with this first thread setup for any of the techniques. This brings us to another factor which is the cost of thread creation that makes the speedup simply vanish for the lowest difficulty, even slow down the application a bit. That being said, the trend is that all methods quickly start catching up on speedup with higher difficulties and especially on `level 3`, which seems to the peaking point for all of the techniques. The highest speedup gain comes from the `Threads + Atomics` method at exactly `5x` on difficulty 3, followed by `OpenMP` at just above `4.7x` at the same difficulty. It seems that OpenMP default thread management operates quite smart and is almost on par with atomics, although this is just in the context of using maximum processing units. As for the `Thread Pool` technique, it is not at all impressively fast in this case.

Continuing with `Figure 6`, there is definitely a massive change in the picture. `OpenMP` and `Threads + Atomics` now have performance gain for difficulty of 1 but this is as far as it goes for the latter one because for all of the above difficulties its performance gains are minimal compared to the former technique and the Thread Pool one. Here `OpenMP` proves the fastest of all techniques for all setups on difficulty of 3 at `5.2x` speedup as well as the fastest on this particular setup for all difficulty levels. With this particular setup, the `Thread Pool` method benefits from a good speedup. It even matches the `5x` on difficulty 3 which was gained by the `Threads + Atomics` technique with the former thread setup experiment.

Figure 7: Efficiency chart using `hardware threads`

Figure 8: Efficiency chart using `dynamic threads`

It seems that none of the techniques is really efficient for difficulty as low as 1 on the setup in `Figure 7`. However, `Threads + Atomics` is the most efficient of all for difficulty 3 and is almost on par with `OpenMP` for the rest. Whereas, with the second setup in `Figure 8.`, `OpenMP` simply beats the rest for all difficulties, tightly followed by the `Thread Pool` technique on difficulties of 3 and above.

5 Conclusion

The investigation shows that the algorithm is highly parallelisable as the difficulty for calculating a hash increases. However, it is also examined that this does not mean that all processing units are needed to gain a performance speedup. In fact, it is just the opposite with the case of low difficulties which is most likely how this program should be running in a real-world environment. Therefore, the variety of conducted experiments prove that throwing more numbers of processors or threads is not really all it takes to achieve efficient parallelism. A lot of experience and tests are needed to find the exact configuration of parameters that can bring the most of a certain computer architecture. Every approach showed strengths under different setups but `OpenMP` is definitely the most consistent in terms of efficiency.

However, this is not a completely fair comparison against what `Thread Pool` can achieve if designed better, so that it does dynamic resizing depending on the needs of the program which is a technique to further investigate and develop if the project is to be continued. Additionally, `cooperative multi-tasking` with the use of `Fibers` which are considered to be lighter weight threads is another field for future investigation in performance increase as the overhead will be minimised because there will be no task interruptions.