CS3210 Assignment 1

Aaron Sng (A0242334N)

27 September 2021

Machines Used

- soctf-pdc-014 - Intel i7-7700 with 4 physical cores and 8 hyperthreads
- soctf-pdc-021 - Intel i7-9700 with 8 physical cores and 8 hyperthreads
- soctf-pdc-006 - Intel Xeon Silver 4114 with 10 physical cores and 20 hyperthreads
- soctf-pdc-019 - Intel Dual Socket Xeon Silver 4114 with 20 physical cores and 40 hyperthreads

Program Design

Parallel Programming Pattern

The program submitted was based on the master-worker parallel programming pattern. In essence, upon creation of the simulation, one master thread and n worker threads are spawned. The master thread coordinates the results from the worker threads, and the worker threads prioritise the computation of the simulation.

Design Considerations

Data Parallelism

The sequential implementation shows an inherent lack of data dependencies in intragenerational computation, specifically when determining subsequent states. The various forms of data dependencies exist in the computation of next states, where the calculation of individual elements of the matrix are dependent on the surrounding cells and their previous state. In contrast, the program shows apparent task dependencies, with each generation requiring knowledge of invasions before determining the subsequent states. Figure 1 illustrates this with an overview of the program's task dependencies. Hence, this presents a stronger case for the master-worker implementation, as it involves an implicit data parallelism construct.

Overhead Minimisation

Another consideration is to minimise the amount of context switches. Overall, the computation of each generation requires multiple context switches. Parallel programming patterns such as parbegin-parend and fork-join implies the presence of repeated termination of each thread after the computation of each generation. This creates large amounts of overhead for the program and was the initial approach used to parallelise this problem. It was observed that not only did it result in greater overhead, but a slowdown of the program compared to the original sequential implementation. A master-worker programming pattern on the other hand allows a longer lifespan of threads, thereby minimising unnecessary thread termination and effectively reducing overhead.

Improving Spatial Locality

The master thread first divides the program for each individual worker thread to work with on a row basis. Row basis was used to reduce the likelihood of cache misses for each computation, due to the increased spatial locality of the program. Hence, for K rows with n threads, each thread will perform computation for approximately K/n rows. After all the threads are done with their allocated rows, the master thread will then allow the next generation of the simulation to be computed.

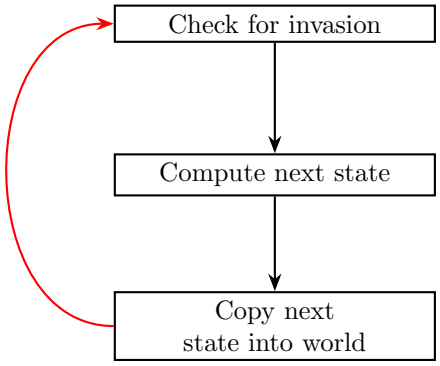

Figure 1: Intragenerational Task Dependency Graph

Eliminating the use of Mutual Exclusions

Mutual exclusions (MUTEX) incur some parallelism lost, specifically when updating the current world and as well as other variables such as death toll. It also makes the program sequential. To replace the use of MUTEX, other synchronisations constructs such as barriers will be used. An array to store the death toll will also be used, with the index corresponding to the thread IDs to reduce data contention.

Potential Downside of Implementation

Having focused my implementation on dividing tasks row-wise, it could mean that the program will potentially suffer performance losses when the inputs become largely flat. However, realistically such a world does not exist as the Earth is not flat.

Methods

Running Experiments

After compiling the program with `make build`, the `perf stat` utility was used to understand the performance of the system.

```
# Run perf stat on all code
perf stat -e cache-misses,context-switches,cycles,instructions -- ./goi_omp
    sample_inputs/sample6.in test6.out 40
# ---- OR ----
perf stat -e cache-misses,context-switches,cycles,instructions -- ./goi_omp
    sample_inputs/lol.in test_lol.out 40
```

`perf stat` was ran on machines of different types mentioned at the start of this document. After the `perf stat` completes running, the elapsed times were recorded. In the experiment, number of threads were varied until the number of hyperthreads in the system. The example in the above code snippet was used on the Intel Dual Socket Xeon Silver 4114, which has a total of 40 hyperthreads. Two inputs were used to understand the behaviour of the program.

- Provided input `sample6.in`. It is a matrix of 50x60
- Devised input `lol.in` to be exclusively be ran on the Intel Dual Socket Xeon Silver 4114 machines. It is a matrix of 40x40

Results and Discussion

Unless otherwise specified, the program input and its corresponding execution time would be a 50x60 size matrix ran on 1,000,000 generations.

Assumptions

- No other programs in the background that utilise the CPU exist
- Background processes do not have significant impact on the execution of the simulation

Effectiveness of Simultaneous Multithreading (Hyperthreading)

As shown in figure 3, a maximum speedup of 9.88x was recorded when 20 threads were used on the Dual Socket Intel Xeon Silver 4114. 20 threads also coincides with the total number of physical cores present on the Dual Socket Xeon Silver 4114. Subsequent increase in number of threads have also gone to show that the computer is not able to effectively speedup the program, suggesting the fact that the Thread Level Parallelism advantage gained with hyperthreading on the Intel CPUs is insignificant for this particular simulation. This phenomenon is also somewhat observed on the Intel-7700 where the hyperthreading only improves the speedup of the program somewhat slightly. It suggests that hyperthreading is effective only under special circumstances. These special circumstances include effective branch prediction and absence of bubbles in the pipeline. These circumstances do not happen all the time, and are highly variable. Another hypothesis is that the performance gains from hyperthreading could be negated by the increased communication between threads, hence performance increases become slight or the same as observed in the experiment.

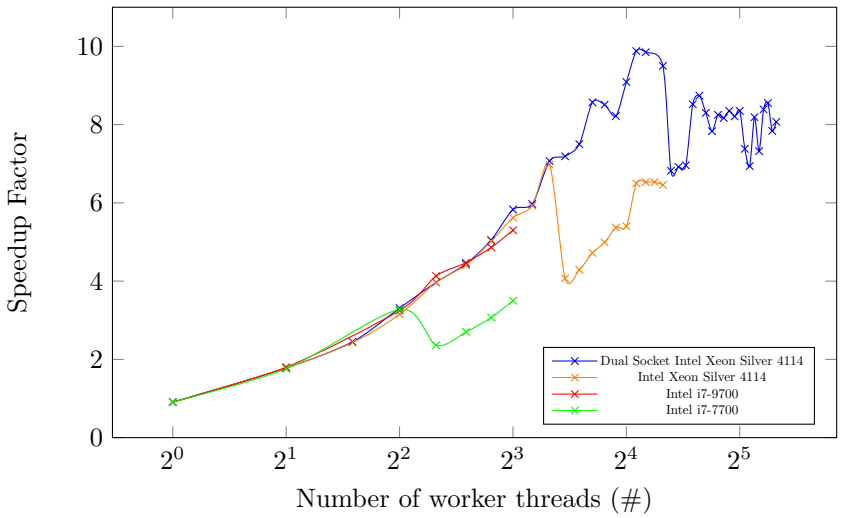

Figure 2: Speedup Factor against Number of Threads on soctf-pdc-019

Optimal Granularity of Program

The very first step of Foster's Design Methodology is to decompose the computation into finer tasks. There are two principles to go about determining the size of such tasks.

- Number of tasks $\geq$ number of physical cores
- Size of task >> overhead of parallelism

Figure 4 illustrates these two principles and the importance of abiding in these principles to obtain the greatest gains with parallelism. In general, most of the fastest execution timings occurred when number of threads (or tasks) were greater than or equal to the total number of physical cores present on the CPU. However, as the number of threads increase, performance begins to reduce and execution timings increase. The phenomenon as mentioned could be due to the increase overhead incurred with having spawned more threads. Another explanation is that with the increase number of threads, more information will be stored in the cache leading to an overflow of data. In other words, not all the information required by the thread is not readily dispensable as the CPU has to cater to the other threads. As a result, the amount of cache misses increases hence slowing the program down. This suggests that the optimal granularity of the program is overall reliant on the number of physical CPU cores present. Another conclusion is that the optimal number of tasks is not based on how big the task is, but if is there a core for an individual task. Generally, this rule is consistent with other processors tested on the 50x60 matrix case.

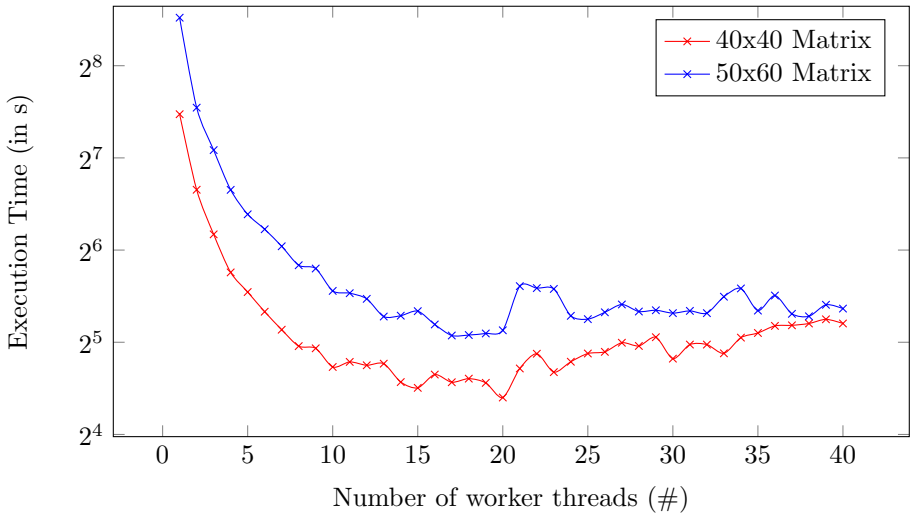

Figure 3: Execution Time against Number of Threads

Minimal Amount of Context Switches

With the threads only initialised once and terminated at the end of the program, context switches is kept at a relative minimum. Fewer than 6000 context switches were observed on the Dual Socket Intel Xeon Silver 4114 when ran on 40 threads.

pthreads implementation

Up till now, the discussion of the parallel program was done on using the OpenMP library, a multiprocessing library that easily parallelises programs. However, a separate implementation involved another library - pthreads. Overall, it has been observed that the OpenMP is a lot more efficient in handling threads. As shown in table 1, each thread created with OpenMP utilises marginally lesser CPU. A likely explanation is an effective scheduler used in OpenMP, while the pthreads implementation is a lot more primitive and scheduler was not an efficient in utilising the CPU.

Worker Threads	1	2	3	4	5	6	7	8
CPU Utilisation (OpenMP)	6%	8%	9%	10%	10%	11%	11%	11%
CPU Utilisation (pthreads)	12%	13%	13%	13%	13%	13%	13%	11%

Table 1: CPU Utilisation Per Thread on OpenMP and pthreads

Areas of Improvement

Performance Speedup

In general, the speedup gained from parallelism has yet to meet its theoretical maximum. The speedup is still lesser than the number of physical cores present on the machine, which leaves a lot left to be desired. The sequential component of the parallel program can be furthered parallelised, such as the portion included to initialise the world. However, further research can be done to further improve the parallelism of the simulation.

Memory Usage

The memory usage compared to the sequential implementation is overall higher with the parallel program. Further optimisation can be used to reduce the memory usage of the OpenMP implementation.

Appendix

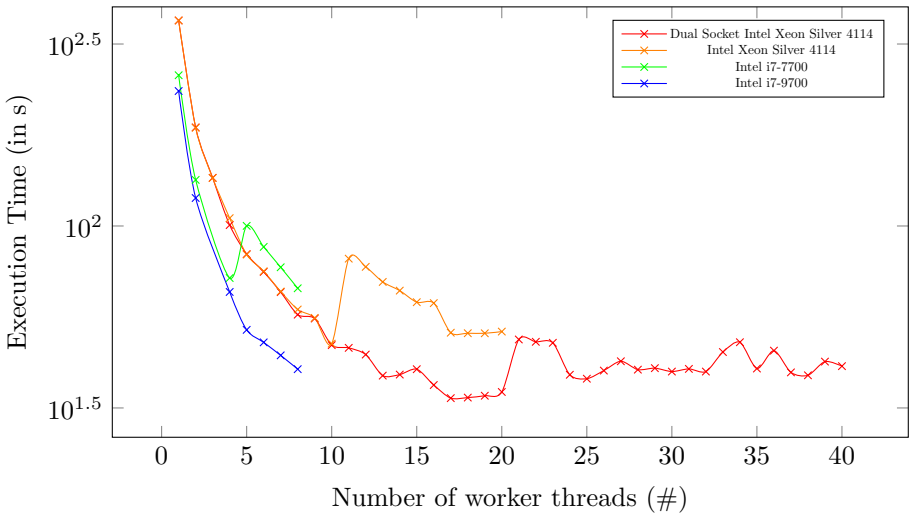

Figure 4: Execution Time against Number of Threads using a 50x60 Matrix