Conway's Game of Life

Buğra Sipahioğlu

Asst. Prof. Didem Unat

March 1, 2020

Note

The evaluator(s) of this assignment can look up the word *BUGRA* in the life.c file. All the changes in the code are commented with the indicator of the name specified.

Data Parallelism

First of all, there are not any explicit implementations in order to make master thread do the initialization. This is because the first thread that runs the program is the master thread and there will not be any other threads until this first thread encounters a parallel region. Therefore, OpenMP libraries were not used explicitly to use master thread for initialization. Secondly, data parallelism was the part that created a respectful amount of speed up. This is due to the sequential nature of the time. In other words, the loop that simulates the time cannot be parallelized, but the nested loops for cell updates can be.

On the other hand, the nested loops that update cells are both parallelizable since there are not any loop-carrier dependencies. In fact, the only operation regarding the current world is read operation. Next world is involved in a write operation, but the threads share different parts of the loops with the openMP library. The nested loops are parallized in this part of the assignment. The variables *currWorld*, *nextWorld*, and upper-bounds (*nx*,*ny*) are shared because they are not unique to a thread and the worlds are assumed to be big data. Too many copies of the large amount of data would create overflow in the stack of threads. The loop indices (i,j) are private because each thread has a different part of the loop, which means different indices. Population is updated each iteration and it allows race conditions. In order to prevent this, the

reduction function of OpenMP has been used. Collapse functionality is used to parallelize both of the nested loops as the latest versions of OpenMP allow such usage. Finally, static scheduling is used for this parallelization. The reason is that every thread in the nested loops does the same job and reaches the same array. Data is homogen. Chunk size is calculated by division of total iteration by number of threads in order to give every thread approximately equal amount of work, therefore achieve one dimensional decomposition.

Task Parallelism

Task parallelism was implemented using the task and single constructs of OpenMP. Before entering the parallel region, two environmental variables are set: *dynamic* and *nested*. Dynamic variable is set to false so that OpenMP does not change the number of threads in the run time. Nested predicate allows OpenMp to utilize nested loops. This is for old versions in the cluster as newer ones do not require nested predicate anymore.

Parallel region has been initialized with the shared data, which are the big data. Although there is no need for specifying shared variables, it is assumed to be good practice. Also, the number of threads is set to the desired number of threads. Inside the region, single construct allows only one thread to distribute tasks. First task also creates another parallel region for the data parallelism mentioned before. In the first task, after the cell updates, pointers needed to be switched. Since there are many threads running the same code, this part is vulnerable to race conditions, thus a critical section has been put. Also, if this part is not handled by critical section or any kind of atomic operations, the plotting thread will be misguided and will draw false plots. In the second task, one thread that is pinned to that task updates the plot. This is the plotting thread.

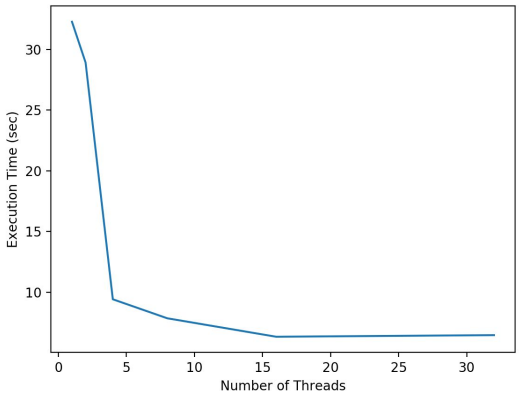
Bottlenecks & Parameter Tuning

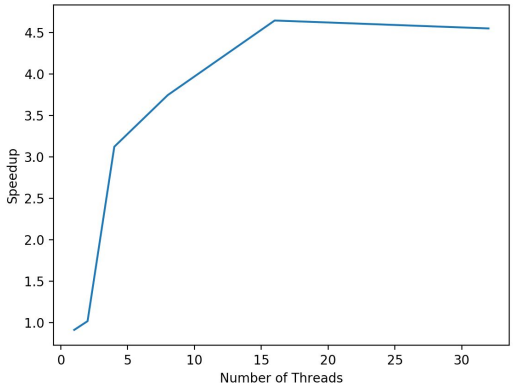
The primary bottleneck of the implementation is parallelization overhead. In every iteration, a parallel region has been created with the addition of new single regions and tasks. Also, critical sections create overhead too since threads have to wait for each other while others are writing. Another problem was the first touch policy since initialization was done by a single thread, but this issue was not investigated since it was mentioned in the description.

For the sake of parameter tuning, loop scheduling types had been tried. Chunk size was fixed since the assignment description required almost equal workloads per thread, but this can be achieved with both static and dynamic (non-guided) scheduling. According to the results, static scheduling is faster. This is due to homogenic data.

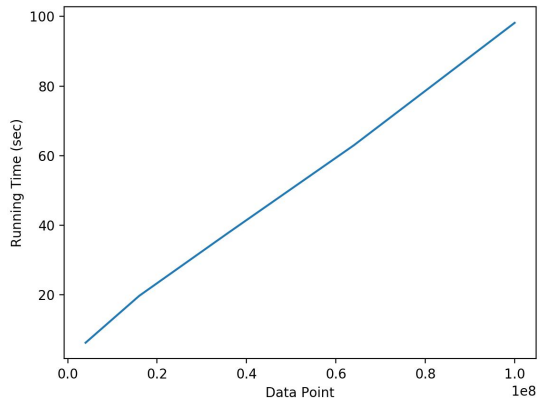
Experiments & Results

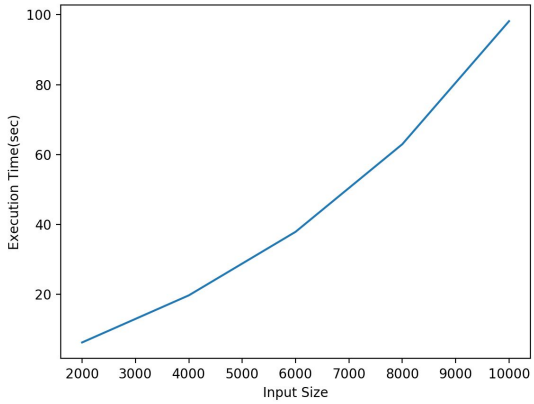
First of all, the performance data was collected from KUACC Cluster. The machine used has 16 cores and 1 thread per core. Figure on the left shows the execution time with respect to the number of threads and the other figure shows the speedup. As shown in the figures, the execution time drops significantly as the number of threads increases and a respectable amount of speedup has gained. These figures are representing strong scaling.





The slow downs after sixteen threads is due to KUACC hardware since the machine used for this assignment has sixteen cores with one thread per node. The below figures show the scalability (running time per data point) and running time per input size. These figures are representing partial weak scaling since the number of threads weren't increased at the same rate of input size. Number of threads was kept at sixteen.





According to the figures, the program is not fully scalable since running time almost linearly increases with the input size. On the other hand, the number of threads should be increased at the same rate in order to understand scalability. Execution time needs to increase with the input size because the upper-bounds of nested for loops increase with the input size.

As a final observation, execution time differs highly with respect to Input Output(IO). For example, approx. 25 seconds had passed with the IO while the execution without IO cost approx. 3 seconds.This is because graphic contents,plots in this case, cost time. Since all of the CPUs are used in this assignment, every program that is not related to the execution increases the execution time.