**Wator simulation benchmark on multi-threaded performance improvements**

Alexandre Paquette

Department of Computing – South East Technological University

ZDEVC4202 - Concurrent Development

Joseph Kehoe

Date Submitted: December 1st, 2023

# Executive Summary

The aim of this project is to assess the impact of parallelization on the performance of the Wator simulation under reproducible conditions. The benchmark utilized a Windows Subsystem for Linux on a Windows machine, running Ubuntu 22.04.3 LTS, with an Intel Core i5-7300HQ processor and predetermined global variables for grid dimensions and species percentages, as well as seeded grid generation.

The simulations were run with thread counts 1, 2, 4, and 8. They were repeated 20 times per thread count, each lasting 100 chrono. The parallelization was achieved using OpenMP pragma, with the number of threads controlled by a global variable. Horizontal tiling was used to distribute the rows of the grid between each thread.

The results demonstrated a clear performance improvement when transitioning from one to two threads, with a marginal increase from two to four threads. There is no noticeable improvement when going from four to eight threads, likely due to the system's limitation of four cores.

# Parameters

This benchmark was run in Ubuntu 22.04.3 LTS in Windows Subsystems for Linux on a Windows machine, build 19045.3693. The CPU is an Intel Core i5-7300HQ @ 2.50GHz, with 4 cores. Below are the values used for the global variables when compiling the project:

const double fishPercent = 0.05; /\*! Percentage of fish that should be rendered \*/

const double sharkPercent = 0.01; /\*! Percentage of sharks that should be rendered \*/

const int xdim = 500; /\*! Number of columns in the grid \*/

const int ydim = 500; /\*! Number of rows in the grid\*/

const int numSharks = xdim \* ydim \* sharkPercent; /\*! Number of sharks to be spawned \*/

const int numFish = xdim\*ydim \* fishPercent; /\*! Number of fish to be spawned \*/

const int FishBreed = 3; /\*! Chronon before fish breed \*/

const int SharkBreed = 5; /\*! Chronon before shark breed \*/

const int SharkStarve = 7; /\*! Chronon before shark starve \*/

int threads = 1; /\*! Number of threads to run \*/

const int WindowXSize = 1080; /\*! X window size\*/

const int WindowYSize = 1920; /\*! Y window size\*/

const int cellXSize = WindowXSize / xdim; /\*! Cell X dimensions \*/

const int cellYSize = WindowYSize / ydim; /\*! Cell Y dimensions\*/

The threads variable was overwritten and incremented to 1, 2, 4, and 8 in the runBenchmark() method. The random generation seed in the initializeEcoSystem() method was set to 1868995012. Under these conditions, the ecosystem was found to be in equilibrium for several minutes at a time, which is ideal for benchmarking performance with parallelization.

# Methodology

## Benchmarking

To get accurate results, the benchmark was run with a seeded grid initialization and looped over an array containing the thread counts to use (1, 2, 4, and 8 in this case). The outer loop iterates through the array of thread count and applies the number of threads to the simulation. The simulation was run 20 times per thread count for 100 chrono. At the end of each test, the time is recorded using omp\_get\_wtime() before and after running the simulation. The total time is calculated by subtracting the start time from the end time and outputted with the thread count for the test.

## Parallelization

The updateGrid() method is parallelized using Pragma OMP. The thread count is available via a global variable and is updated in the runBenchmark() method. The grid is tiled between each thread horizontally using integer division. To account for uneven divisions, the last thread is assigned the remainder of the rows not divided between the previous threads. Boundary checking is not utilized in this simulation, as it is assumed each thread will run at similar speeds, which should not cause any collisions.

# Results

**Table 1**

*Raw data of the benchmarks for the Wator simulation with pre-determined conditions previously outlined*

|  |  |  |  |
| --- | --- | --- | --- |
| **1 Thread** | **2 Threads** | **4 Threads** | **8 Threads** |
| 10.4716 | 9.80837 | 8.50083 | 8.38411 |
| 15.0117 | 9.69724 | 8.39543 | 8.37591 |
| 15.0266 | 9.6732 | 8.38746 | 8.3329 |
| 15.0043 | 9.62354 | 8.5442 | 8.69025 |
| 14.9453 | 9.84872 | 8.41562 | 8.33798 |
| 15.1287 | 9.70566 | 8.51845 | 8.27673 |
| 15.0792 | 9.74303 | 8.34818 | 8.39092 |
| 15.0413 | 9.70237 | 8.45686 | 8.25124 |
| 15.0008 | 9.73692 | 9.02836 | 8.41408 |
| 15.0204 | 9.95068 | 8.45865 | 8.35004 |
| 14.9268 | 9.69024 | 8.55243 | 8.28271 |
| 15.1016 | 9.88615 | 8.25263 | 8.23686 |
| 15.1747 | 9.79952 | 8.54181 | 8.37024 |
| 14.9876 | 9.69848 | 8.32681 | 8.38174 |
| 14.9786 | 9.64886 | 8.43135 | 8.40461 |
| 15.0628 | 9.69563 | 8.44262 | 8.39542 |
| 15.0778 | 9.73042 | 8.34109 | 8.28945 |
| 15.1306 | 9.77018 | 8.42591 | 8.34846 |
| 15.1631 | 9.8251 | 8.31436 | 8.29597 |
| 15.0364 | 9.68169 | 8.62409 | 8.38268 |

Table 1. shows the raw results of each trial run. Since the system only has four cores, the results for 4 threads and 8 threads are effectively identical. Otherwise, there is a clear increase in performance from one thread to two threads and a marginal increase in performance from two threads to four. However, there is an outlier in the single-threaded dataset where the very first run completed faster than subsequent runs. This issue was duplicated at different thread counts where regardless of thread count; the first run is always faster than subsequent runs.

**Table 2**

*Derived data from Table 1 denoting average, max difference, and min difference*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Threads** | | | |
|  | **1** | **2** | **4** | **8** |
| **Max Diff** | 0.356205 | 0.20488 | 0.563003 | 0.330635 |
| **Average** | 14.8185 | 9.7458 | 8.465357 | 8.359615 |
| **Min Diff** | 4.346895 | 0.12226 | 0.212727 | 0.122755 |

Table 2. This table shows the average run at each thread count, as well as the difference between the longest runtime and shortest runtime for each thread.

**Figure 1**

*Error bar plot chart displaying the data from Table 2*

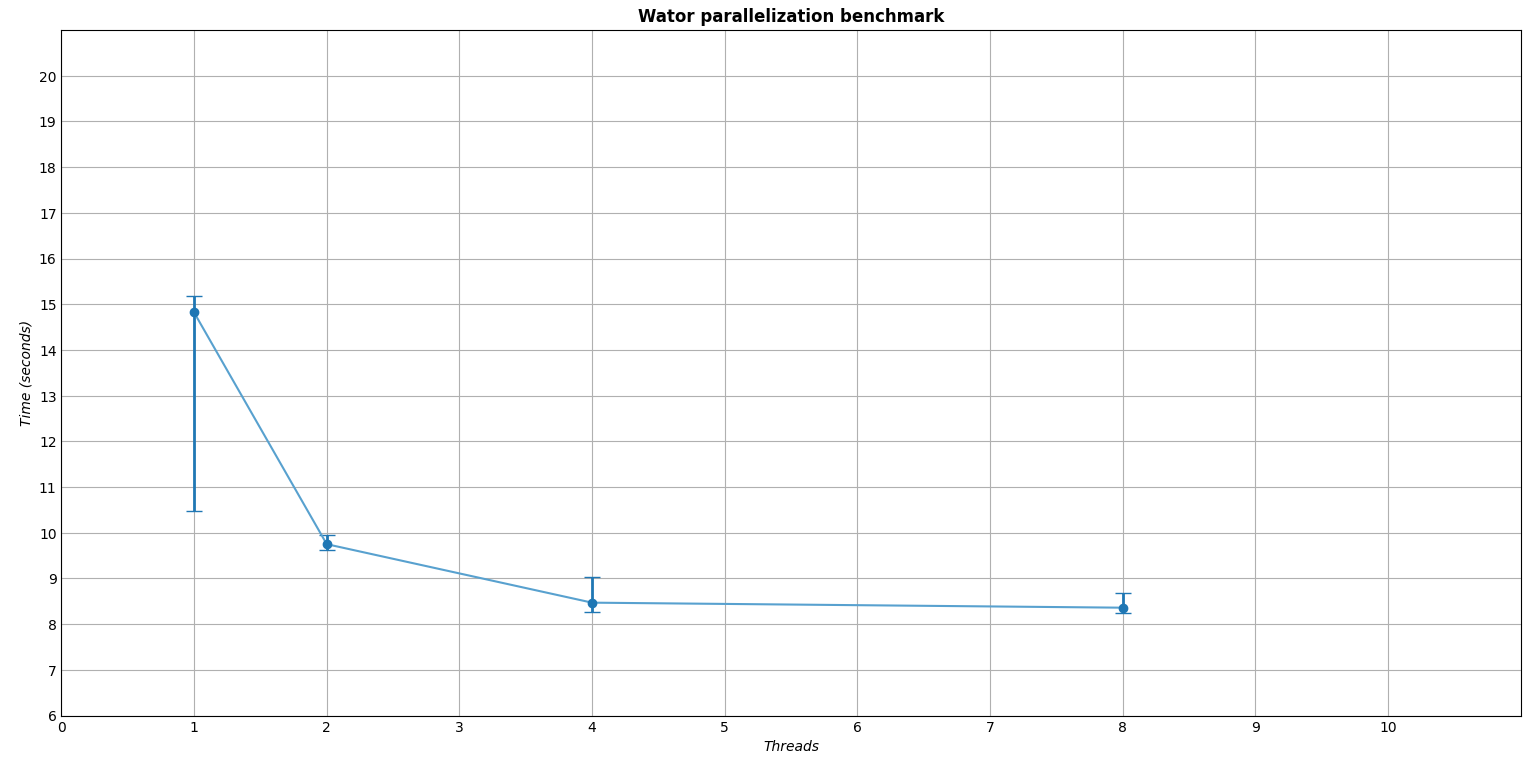


Figure 1. This graph is a visual representation of the data calculated in Table 2. This graph was generated with Python using matplotlib. We can see a clear trend towards rapid initial improvement in performance when jumping from a single thread to two, a marginal increase from two threads to four, and no noticeable increase with 8 threads.

# Conclusion

In summary, the benchmark results highlight a noteworthy performance boost with the transition from one to two threads, followed by diminishing returns as more threads are added. However, the study revealed that exceeding the number of available cores on the system led to no discernible performance gains, aligning with the principles of Amdahl's law. This emphasizes the importance of considering hardware limitations in optimizing parallelization strategies for the Wator simulation.