Scott Merrill

Computer Architecture Final Project

Comparison Between Linear and Parallel Algorithms

Introduction

For the final project I attempted the linear implementation and parallel implementation of an algorithm in order to measure the difference in its performance among all implementations.

Algorithm

The algorithm I used was one designed myself and can be imagined as a spaceship fleet fighting simulator. The pseudocode is as follows:

```
● Initialize all ships in each fleet to a random location in a 20,000 unit cubed
  box. Each ship has 3000 health and 1000 power.
● While neither fleet has been completely destroyed:
    ○ For each fleet:
        ■ For each ship that is not destroyed:
            ● Find the nearest ship in the enemy fleet that is not
              destroyed
            ● Subtract power from enemy ships health
    ○ For each fleet:
        ■ For each ship in fleet:
            ● If health is less than or equal to 0, set destroyed flag
● Print out the number of remaining ships in each fleet
```

I made the decision to to have a “destroyed” flag and to set the flag after each fleet had a round to damage the other fleet because I felt that would make it more realistic in terms of simulating ships firing at the same time, and because not having it and testing “destroyed” solely by ship health would give the first fleet a distinct advantage. The fleet that went first in the loop would destroy the second fleet’s ships, and thus the other fleet would have fewer ships to deal damage with.

Implementations

I implemented the algorithm linearly in C, in parallel using pthreads and in parallel using OpenMP. For implementing the ships I used a struct with floats for the x, y and z coordinates, a pointer to another ship struct for the target, and some integers for health and power. Fleets were implemented using a struct as well, with an array of ships structs, an integer to count how many ships in the fleet had been destroyed, and a pointer to the enemy fleet.

The algorithm lends itself nicely to parallelization--you can either parallelize it at the fleet level, or if you want more threads, at the ship level by have a set of ships for each thread. In my case, I only parallelized it at the fleet level, meaning I had a thread for each fleet, in both pthreads and OpenMP, when targeting, dealing damage, and when setting the damage flag. Because I implemented it this way, I did not have to use mutex locking--because the thread iterates through each ship in a fleet, only one allied ship could only be dealing damage to an enemy ship at any given time, meaning no multiple access on health. I could have also parallelized targeting at the ship level without the need for mutex locking, as each ship is only reading from the enemy ship's position variables and its address.

I also planned an implementation for OpenCL, but was unable to accomplish it due to a number of difficulties. The implementation was planned to parallelize targeting at the ship level, by writing the targeting function as the kernel.

Testing

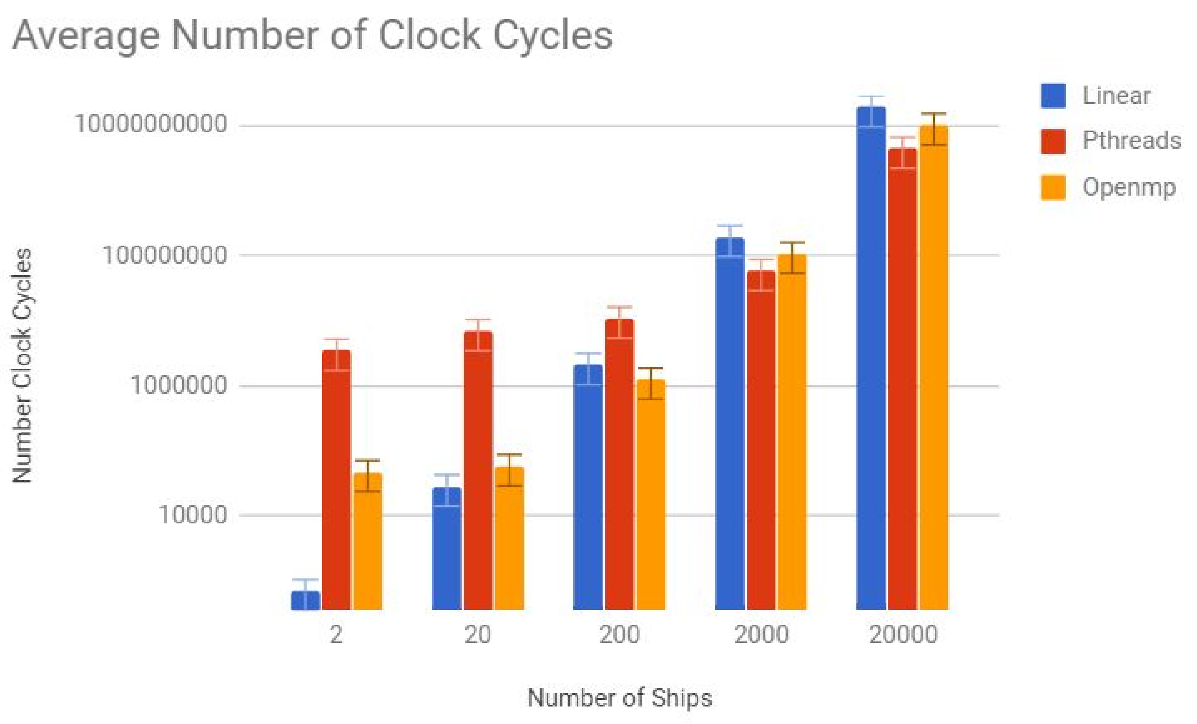
Testing was accomplished by wrapping the above algorithm with some code that would measure the number of clock cycles needed for one fleet to get destroyed and output that number of clock cycles, and the clock cycles needed per spaceship, to some csv files. There were tests for 2, 20, 200, 2000, and 20,000 total spaceships, and there were 20 trials per test.

Results

The graphs of the results along with the tables are shown below. The first graph and table are the average number of clock cycles it took to run the algorithm, are scaled logarithmically, and show the general trends as the number of ships increases. For numbers of ships below 200 it took the least number of clock cycles for the linear implementation to run, for 200 ships it took the OpenMP the least amount of time to run, and for above 200 ships it took the pthreads implementation the least amount of time to run.

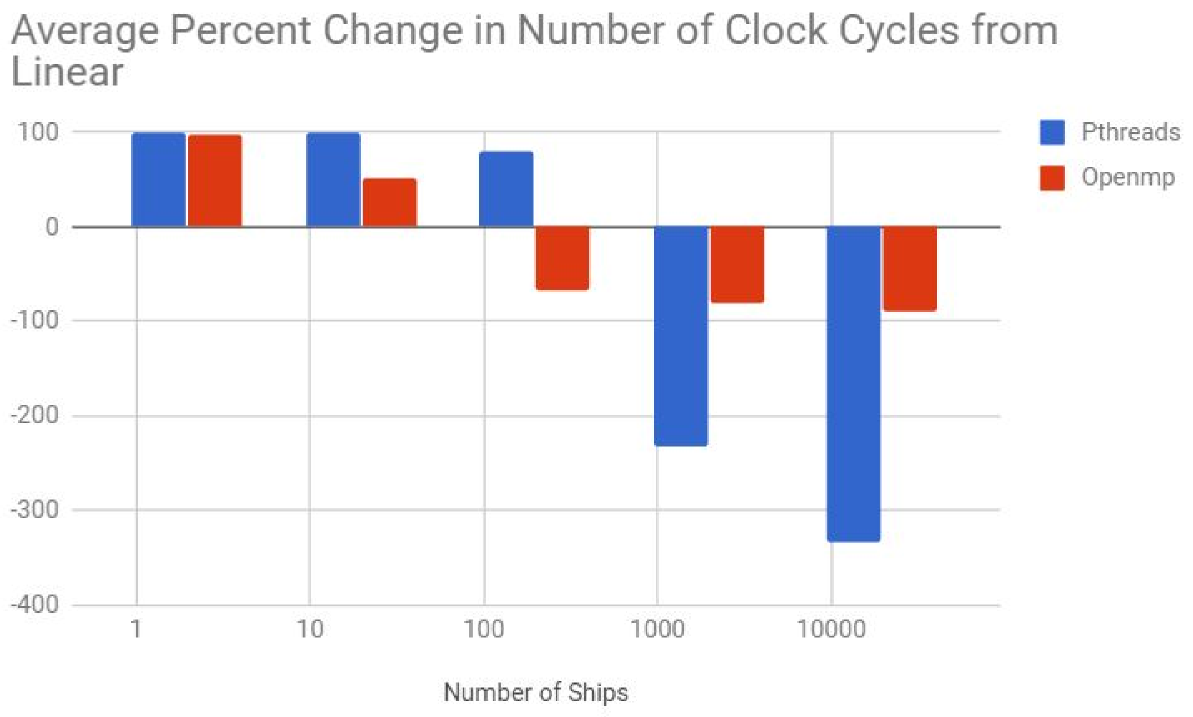
The amount of time the linear implementation takes scales linearly with the number of ships. Both OpenMP and pthreads start off taking more clocks cycles, and seem to approach a linear scaling with the number of ships, but take fewer clock cycles.

At a lower number of ships, the standard deviation in the number of clock cycles was much closer to the number of clock cycles it took to run.



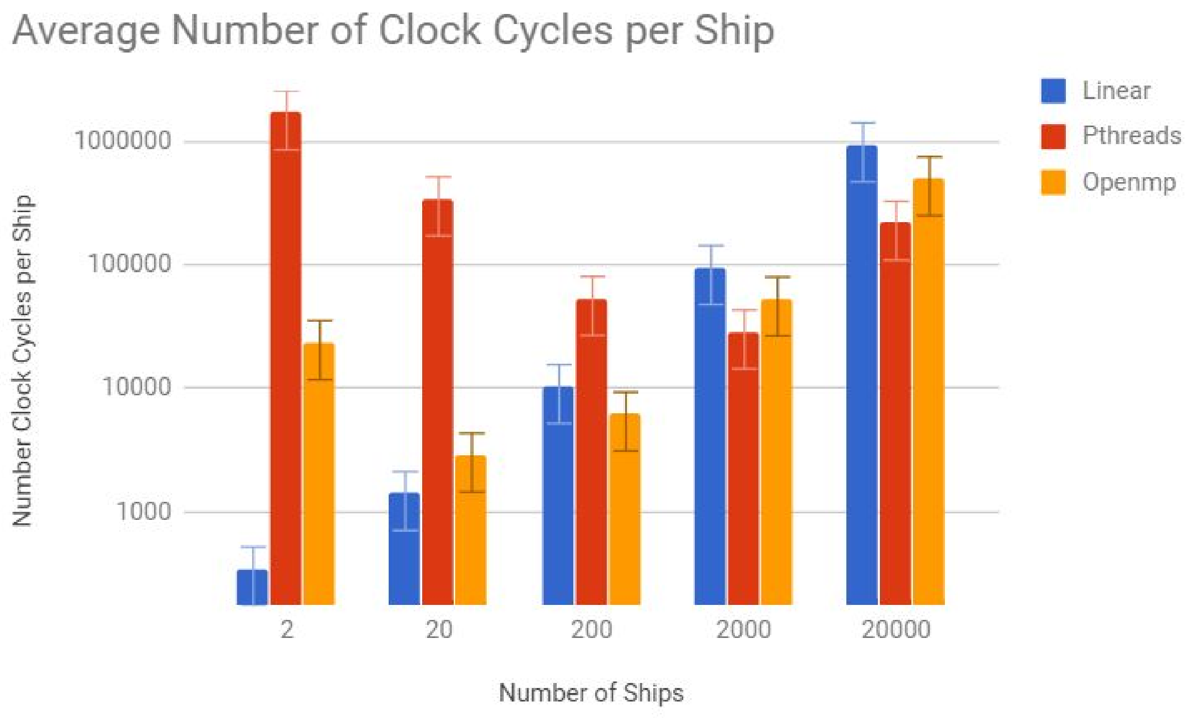
Average Number of Clock Cycles				Std Dev		
Number of Ships	Linear	Pthreads	Openmp	Linear	Pthreads	Openmp
2	693.2	3442087.5	47216.95	332.6310743	794710.6787	111484.683
20	28318.3	6913649.6	57882	12522.51962	963480.3863	13760.50187
200	2084085.7	10758495.25	1247443.3	200848.6976	1965754.098	93965.94817
2000	191598196.6	57757873.6	106602710	3787844.848	4922601.283	1702814.825
20000	18943518715	4383927599	10063853818	208938666	107012429.3	123971721.2

The second graph and table are of the percent change in the amount of clock cycles from the linear implementation to the other implementations. This data is useful because it show just how much faster on average the other implementations were than the linear implementation. At a low number of ships pthreads and OpenMP take almost 100% more time than the linear implementation, or twice as long. At around 200 ships, OpenMP shows a 67% decrease in the number of clock cycles, and after that we can see that pthreads shows a massive 300% decrease in the number of clock cycles taken, meaning it took about ⅓ the time of the linear algorithm.



Percent Change from Linear		
Number of Ships	Pthreads	Openmp
2	99.97986106	98.53188315
20	99.59040013	51.07580941
200	80.62846475	-67.06857137
2000	-231.7265416	-79.73107494
20000	-332.1129464	-88.23324601

The final graph shows the average number of clock cycles taken per ship. This data is interesting because it shows the optimum number of ships to use each implementation for. The linear implementation has no optimum point, as its slope is always positive, OpenMP is optimum at around 20 ships (even though the linear implementation is still faster), and pthreads is optimum at around 2000 ships.



Conclusions

The linear algorithm scaled linearly with the number of ships because for each additional ship it takes proportionately more clock cycles to process.

I hypothesize that pthreads and OpenMP had the large low ship number clock cycle costs because it takes clock cycles to move the data around and setup the thread on the new core.

I am not sure why pthreads and OpenMP outperform each other in different situations. One forum post compares OpenMP to being ideal for same task in parallel (which seems to be what I'm doing), and pthreads for being ideal in performing different tasks at the same time. More research is required.

I think that the overall conclusion that can be drawn from this is that each implementation is better at different working set sizes, and that you can optimize your spaceship fight code (and perhaps other algorithms) by using different implementations for different working set sizes. You should definitely be able to optimize best by creating each implementation and seeing which one runs the fastest.