Project: Part 1

Xingku Yin
Alexandru Marcu

Background:

We are tasked with parallelizing a toy particle simulation. For the first part of the report we are tasked with implementing a serial algorithm that ideally would run with linear, O(n), time complexity (ideally), where n is the number of pixels.

We are given a naive algorithm that runs in quadratic, O(n^2), time complexity. This is due to the fact that the naive algorithm checks, for each particle in the simulation, the interaction between itself and all other particles in the simulation. This is where we will make the majority of our improvements with respect to time complexity.

To run either the Serial or OMP implementation, first run the script ./setup-project.sh . We have modified the script to only run the make targets for Serial and OMP and autograder, and to disregard MPI, as that is for part-2 of the project. Then, to run the code, it is identical to the method outlined in the hand out. Use the command sbatch auto-teach-* , where auto-teach-* is either of the provided auto-teach scripts for their respective implementation (either serial or OMP).

Serial O(n) implementation:

We proceed with our serial implementation using the first method outline in the ProjectHelper.pdf linked in the project hand out. With this implementation, we partition the simulation space (which is a square), into bins of size cutoff * cutoff, where cutoff is the minimum distance allowed between particles. That is, when two particles are within the cutoff distance, they are repelled from one another. Then, at every time stamp, we place particles into their respective bins, and instead of checking a respective particle's interaction with all other particles in the simulation space, we only check the respective particle's interaction with those particles in neighboring bins. This cuts down the number of interactions we need to check by a

significant margin, and allows us to bring our time complexity closer to O(n), in the average case.

To maintain a reference to these bins, we use a C++ vector container, which represents our 2D simulation space using a 1D array of bins (A2 was our motivation for this), which gives us a convenient way of calculating which neighboring bins are valid. Typically we check 8 neighboring bins for a particular particle, but some particles may lie in bins along the edges of our simulation space, and thus won't have 8 neighboring bins.

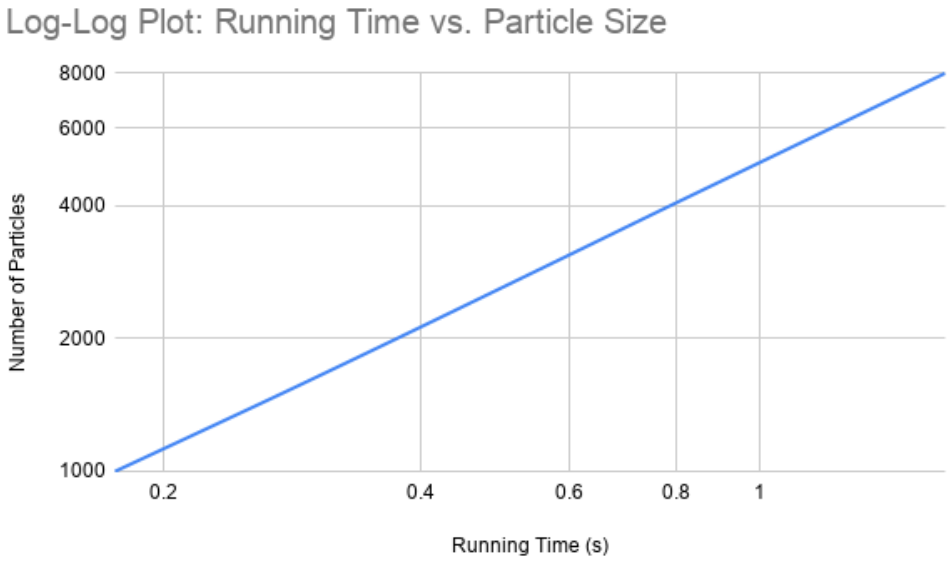
This choice of data structure and general algorithm is where we incur most of our performance costs which slows us down.

This stems from the fact that at each time stamp of the simulation, we are required to clear the particles from all the bins (as their position at time stamp *t* is different from their position at time stamp *t-1*), and then we must populate the bins with all the particles and their new location. Clearing the bins from the vector container is an O(n) operation, likewise with placing the particles in their respective bins.

Serial O(n) Data & Analysis:

Small Data Set Data & Analysis:

Run-time (s)	Particle Size (n particles)	Slope Estimate
0.175593	1000	1.057091
0.275157	1500	1.107797
0.3756	2000	1.081690
0.578837	3000	1.066667
0.785322	4000	1.060452
1.21357	6000	1.073404
1.6528	8000	1.073770

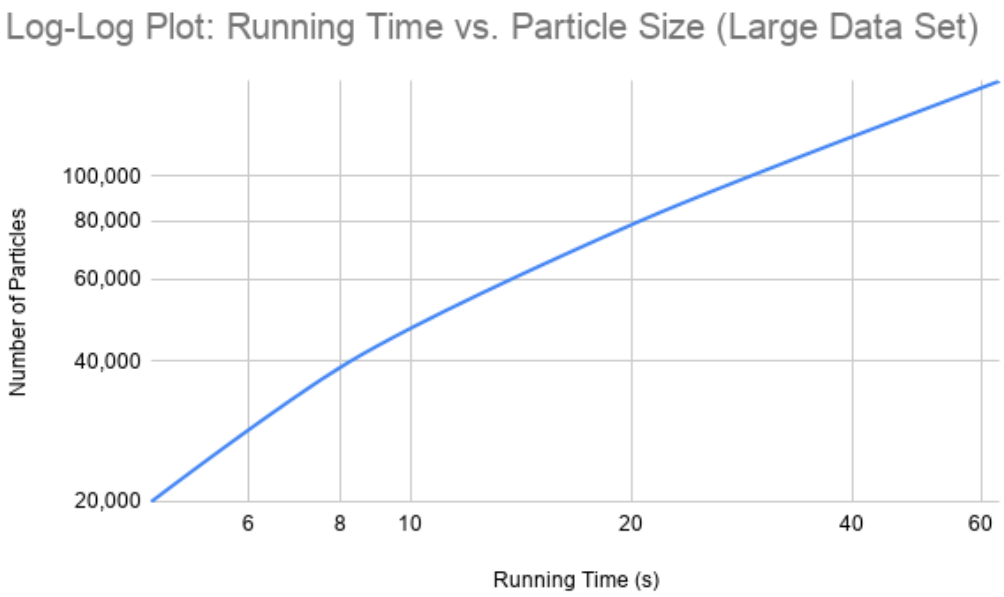


Looking at the table data for the small data set (auto-teach-serial-small), we can see that our algorithm performs very well, and is showing the desired result of being as close to as linear in time complexity. The only outlier seems to be for 1500 particles, which is the only datapoint to have a slope above 1.1 (where slope of 1.0 is desired). This outlier might be caused due to load on the server perhaps, and might not be as apparent if we took an average over multiple runs.

Looking at our Log-Log plot, it is apparent and clear that our algorithm is performing with the desired O(n) time complexity. The autograder gives us a slope estimate of 1.074134 for the slope of best fit. Perhaps this could be improved on more, but we think this is very desirable and would be very difficult to improve on, as it is only 7% away from the idealized target of 1.0.

Large Data Set Data & Analysis:

Run-time (s)	Particle Size (n particles)	Slope Estimate
4.4302	20,000	1.079134
8.2638	40,000	0.899439
20.5047	80,000	1.311072
63.5191	160,000	1.631236



Running our optimized serial implementation on the large data set (auto-teach-serial-large) reveals slightly less than desired results, when compared to the resulting data from the small data set. Although the slope estimate for best fit was given by auto grader to be 1.205227, we can see from our Log-Log plot that the curve is not as desirable as the curve from the small data set Log-Log plot. Looking at the table of data, this is clearly due to the fact that for the largest particle size test of 160,000 particles, we have only made a marginal improvement to the run time complexity.

We tried to improve the performance for the largest particle size test, but we were unable to come up with an adequate solution. After some failed attempts, we decided that we should proceed with our OMP implementation, and with that (as later discussed in more detail), we were able to achieve the benchmark given in the project hand out (~10s). This leads us to believe that although the slope for 160,000 particles is merely ~1.6, compared to the desired 1.0, this outlier is acceptable given the sheer size of the test, as our algorithm performs as required for more reasonable particle sizes given that we are running these tests on a singular core.

OMP Implementation & Strategy Discussion:

Our omp implementation is based on our optimized O(n) serial implementation. We improved our omp implementation with new ideas for optimizing it step by step. There are some similarities among all implementations, being that we parallelize the clearing of the bins at each

time stamp (since all bins are independent of each other), as well as computing the forces (parallelizing the code from our O(n) serial implementation), and moving the particles. These three parallelizations are done using a simple #pragma omp for directive. This alone was not enough to achieve the benchmark time given in the handout, so we attempted to get more creative, and identify areas where we can improve.

This section outlines our step by step thought process on how we attempted to make our algorithm more efficient, and concludes with our final, complete, implementation. The main idea between all these implementations is related to figuring out an efficient manner of splitting the work of placing particles in their respective bins amongst all threads.

Firstly, we started with the initial idea of all threads sharing a universal lock based when trying to write particles into bins. The reason behind is that memory leaks will occur if that part of code is not protected as that is a critical section. As expected, the execution is really slow, similar to the original omp.cpp in the starter code. But we solved the issue of memory leak and made sure this is the only critical section in the program. The run time for the 160000 particles test is around 70s. The code for this version is saved at openmp_plan0.cpp

We then tried to improve it by only letting 1 thread, the first thread that reach the section of putting particles into bins, to put all particles into bins. A barrier is placed below this section, so all threads, will wait for the threads that put all particles into bins to finish its task, and the will continue with a new iteration. The reason behind this implementation is that we suspect constant acquiring for lock takes a lot of time. The result proved our idea to be correct as it run much faster. The runtime for the 160,000 particles test is around 20s. The code for this version is saved at openmp_plan1.cpp

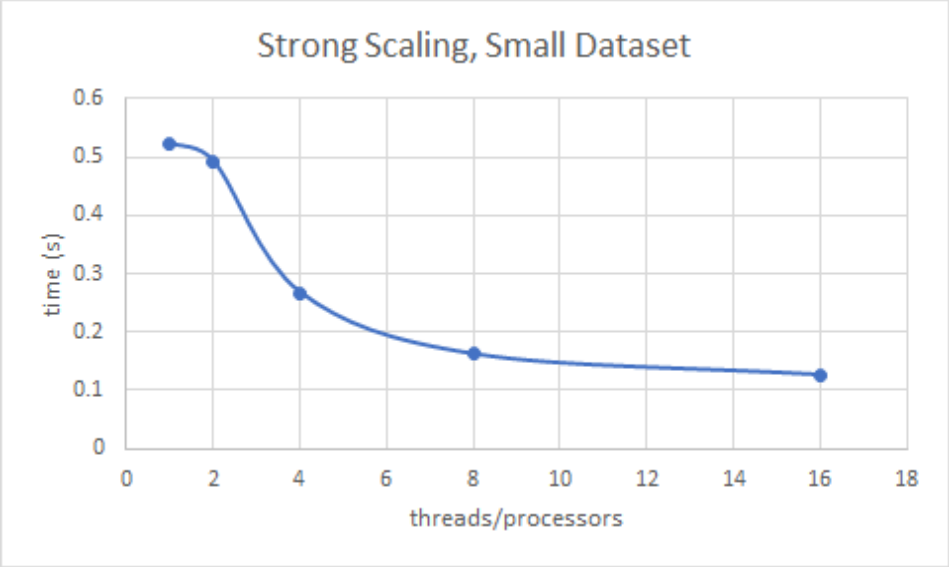
As you can probably tell, the next improvement we are going to make is to make the task of putting particles into bins synchronized. The first idea we came up with is that each threads is assigned certain amount of particles, all threads are assigned the same amount except the last thread that is going to take all the remaining. Each threads have their own set of local bins. All threads will place their assigned particles into their own bins. After that, they will put the particles in their local bins to the shared bins. The section of putting particles from own bins to shared bins is surrounded with a lock. They will meet at a barrier and then continue with the iteration. This program did really poorly as it did not even finish the 80000 particles task left along the 160000 particles one. The reason we think it is so slow is that the lock makes every threads wait and the process of putting particles from thread's own bins to shared bins is inefficient since to update the shared bin's with the local bins of the thread, we are required to use the vector function insert(), which runs in O(n). This makes this implementation strategy too computationally heavy to be efficient. The code for this version is saved at openmp_plan2.cpp

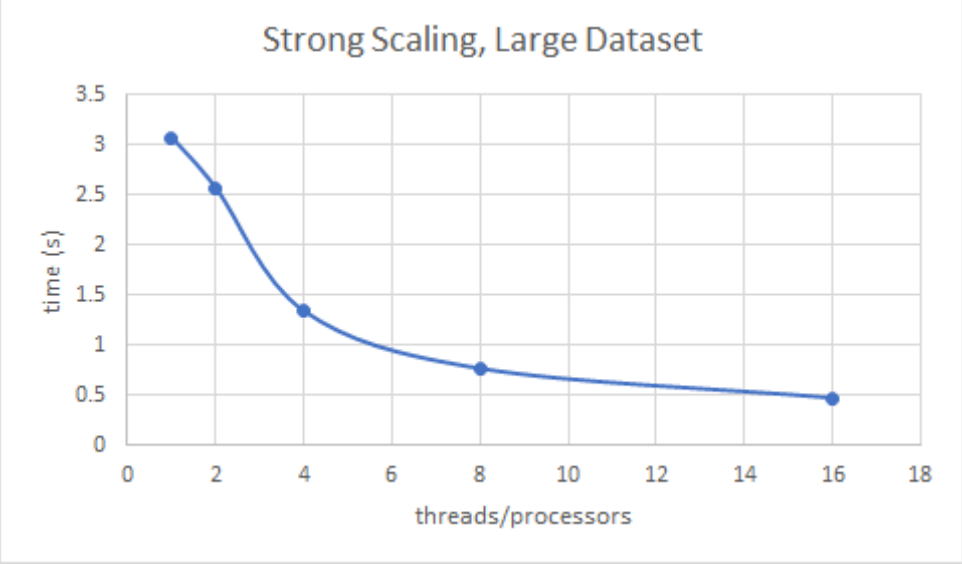
In our final, and working, implementation, we keep a shared reference for all the bins amongst all the threads. Then, we divide the amount of particles evenly among all threads (with the last thread potentially being responsible for slightly more), and each thread is responsible for placing their chunk of particles in the respective bins. To handle the synchronization issues that

arise when placing particles in their respective bins at each time stamp, we create an array of omp_lock_t locks, where each particle bin has a corresponding lock. When a thread wishes to place a particle in the particle's respective bin, it acquires the lock, pushes the particle into the bin, and releases the lock. This improves on the previous locking strategies we attempted, because since the particles are uniformly distributed, and because the threads are assigned contiguous shards of particles, it is very rare that the threads will be fighting for a specific bin's lock. In the unlikely case that this does happen however, we are using the vector function push_back(), which runs in O(1), compared to the previous implementation strategy in which we used the vector function insert(), which runs in O(n). Thus, if two threads attempt to acquire the same lock, it does not block for a significant amount of time since push_back is O(1). This implementation strategy allowed us to achieve the benchmark given by the project hand out of ~9s for 160,000 particles. The code for this version is saved at openmp.cpp as well as openmp_plan3.cpp

OMP Data & Analysis:

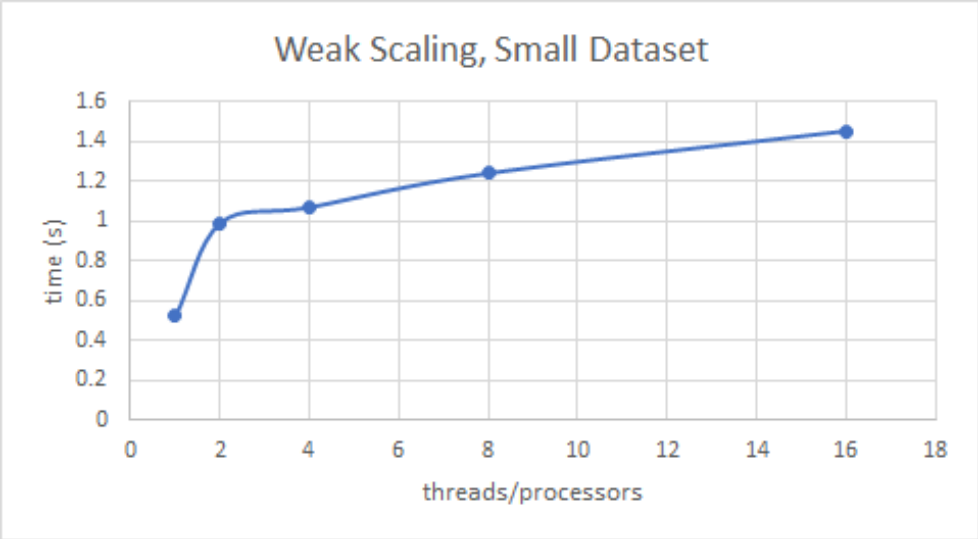
Strong Scaling Performance:

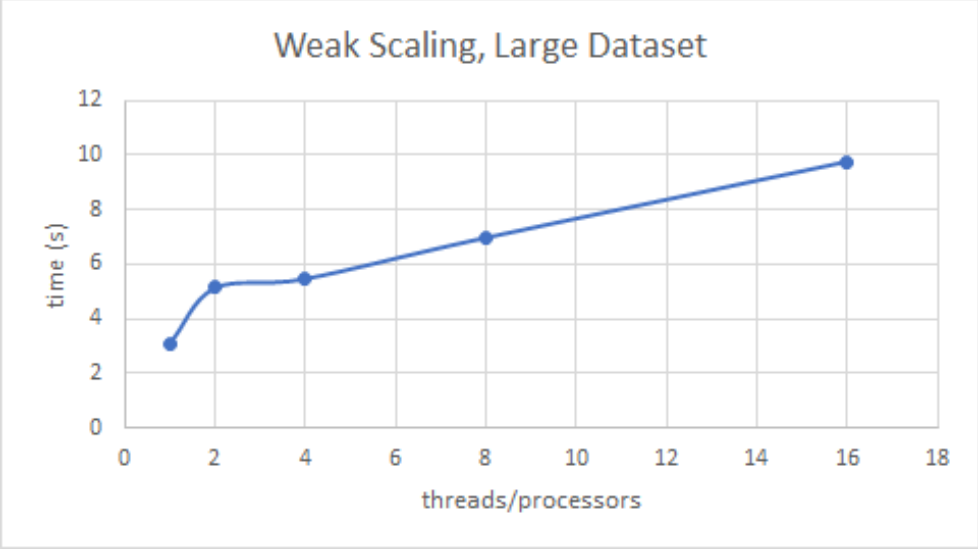




Strong Scaling performance is measured by keeping the problem size (in this case, the number of particles in the simulation) constant, while increasing the number of processors. With this in mind, the graphs for both the small and large dataset seem to suggest that the algorithm is working as intended. We can note that as we increase the processor count, the time to run the simulation decreases. This is consistent, as it is trivial that the more cores we use to solve the problem, the faster the problem should complete. If we were not to see this trend, it would be erroneous.

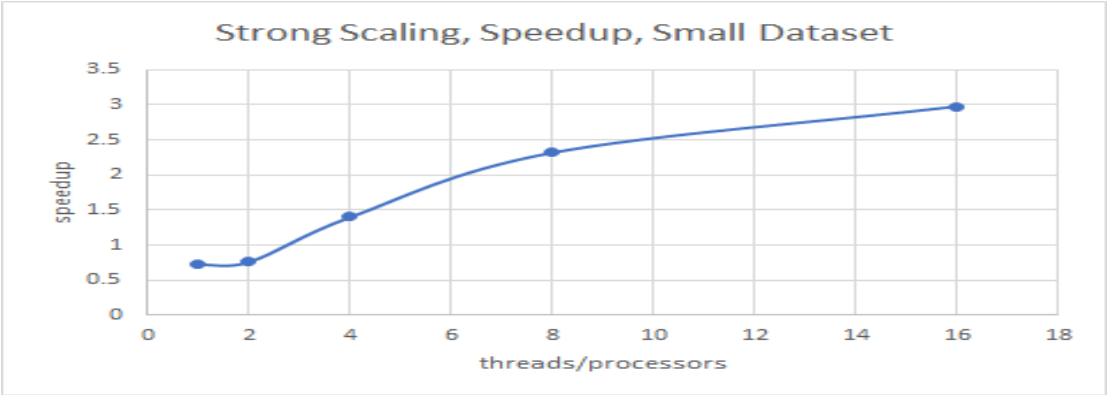
Weak Scaling Performance:

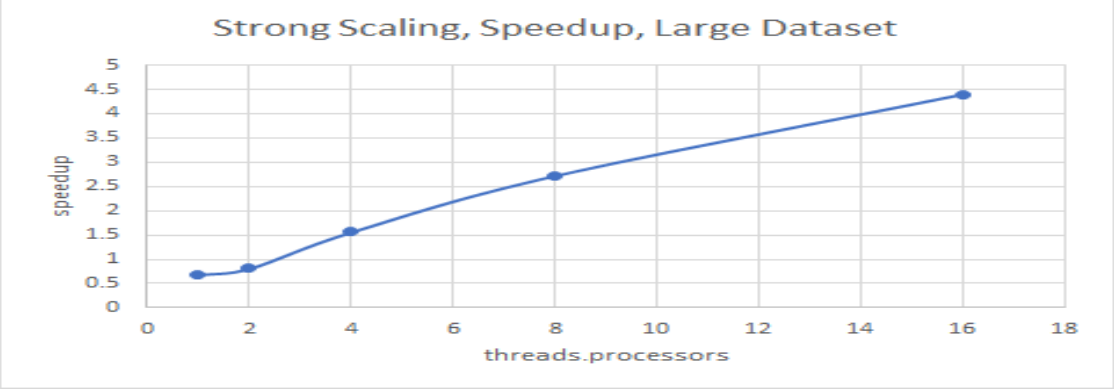




Weak Scaling performance is measured by increasing the problem size proportionally to the number of processors, such that the work per processor is evenly increased. Except for the first 2 data points, the slope is constant.

(2) The strong scaling plots that show the speedup of your optimized OpenMP implementation.

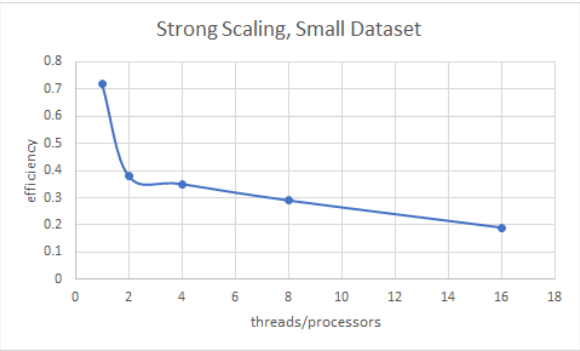


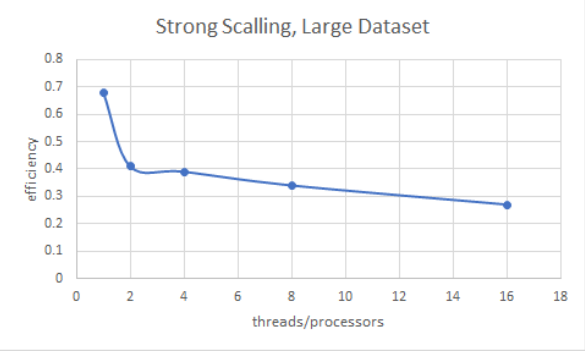


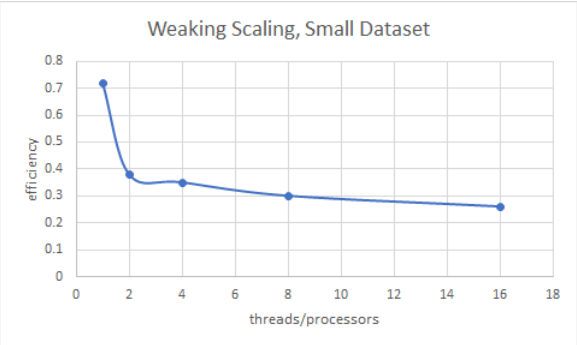
For the strong scaling speed up, as we saw from online sources [1], the speedup of strong scaling slope usually slowly goes flat due to the limit of the serial part of the code that cannot be solved parallelism. We can see the flatten of the curve in the results of both datasets. The flatten is more visible in the small dataset in comparison to big data. We think the reason behind is because more particles make it reach the limit slower.

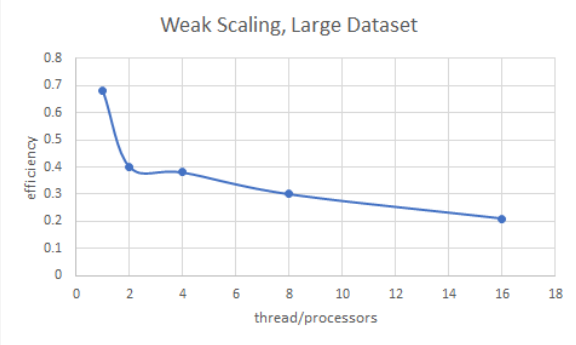
It is much slower from the idealized p-times speedup. We may be able to make improvement by redesigning locks to make less wait.

(3) The strong and weak scaling plots that show the efficiency of your optimized OpenMP implementation.









The efficiency behavior is quite uniform. We can see a sharp decrease in efficiency while the curve is flatter later on.

(4) And a table that shows the average weak and strong scaling efficiency of your optimized OpenMP implementation.

Strong Scaling, Small Dataset	Weak Scaling, Small Dataset	Strong Scaling, Large Dataset	Weak Scaling, Large Dataset
0.38	0.4	0.42	0.39

The average efficiency is pretty uniform as well which matches the above graphs.

[1] Xin Li. 2019. Scalability: strong and weak scaling. (December 2019). Retrieved March 26, 2021 from https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/