Instructions

For the 2nd assignment, you will (1) convert an existing serial C++ code into a parallel, multi-threaded code using OpenMP and (2) run a series of benchmarks on Stampede2 to measure the parallel speed-up and efficiency.

1. Pull the latest updates to the HPC class repo from BitBucket to Stampede2. The serial code, based on the n-body code we discussed in class, is in the hw2/ directory.

2. Verify that the serial code runs and that you can change the number of particles in the benchmark simulation. Start small with perhaps 100 particles (-n 100) and try doubling that a few times. The cost increase should be quadratic. The last line of screen output gives the average time per step (there are 100 steps by default).

Stampede2 has several GCC releases and the Intel C++ compiler. The Intel C++ compiler is usually the fastest and it has the latest OpenMP (4.5) features so let's go with icpc. You can specify the compiler when building with the provided Makefile by setting the CXX environmental variable:

```
prompt% CXX=icpc make
```

This will compile the nbody3 binary with the icpc compiler. The default is to use g++ (v 5.4.0) which will work, too. As with HW1, you'll get different performance with the two compilers. Experiment to see what happens to the cost per time-step.

Finally, experiment with running more steps. The default is 100 steps. Does the average time per step change if you double and quadruple it? Find the number of steps needed to produce consistent results.

As a point of reference, I benchmarked the code using 1000 particles with 500 steps using the GNU compiler took 41.7 ms per step on average and only 2.52 ms. See for yourself. Recall what we discuss about SIMD vectorization.

Once you're comfortable building and running the serial code, it's time to make it run across multiple cores in parallel.

1. To make the code run in parallel, you'll need to add OpenMP parallel pragmas to several of the for-loops. These include the outer loops in `accel_register()`, `update()`, and `search()`. Also, determine the number of threads you're using at run-time and have that printed once during each run. This will require using an OpenMP API function, not a pragma.

Examine the three main for-loops and determine how best of parallelize them. Be careful about data dependencies and race conditions. You will need to change the Makefile to enable OpenMP. Add `-fopenmp` to either compiler's option (`CXXFLAGS`) to enable this feature.

Note that you can add the OpenMP parallel statements in stages; you don't have to code all of them at once. That is, you can make one loop at a time parallel and verify that the results are the same. The best way to verify is to look at the screen output. Every 10 steps, the min, max, and average particle velocity is printed. Those shouldn't change. Once you have all of the code modifications completed, you're ready to benchmark the parallel code.

You can change the number of threads by adjusting the OMP_NUM_THREADS environment variable. Try with 2, 4, 8, 16, and 32 threads. Again, the results should not change and, hopefully, the time per step will decrease.

1. It's time to run a series of serial and parallel benchmarks. We're going to run two major benchmarks, one of measure the *strong* scalability and another to measure the *weak* scalability.

Run serial and parallel cases with 1,000, 2,000, 4,000, and 8,000 particles using 1, 2, 4, 8, 16, 32, and 64 threads. (Stampede2's KNL nodes have 68 cores and each core can run 4 hardware threads so you can try with more threads if you'd like.) Record the average time per step. Make sure you use enough steps to give consistent results. These results will be used to compute the strong scalability and efficiency.

Now run a weak scalability test. The goal here is to keep the workload per-thread constant. This means you need to increase the number of particles when you increase the number of threads. Start with 200 and 1,000 particles serially and determine how many particles are needed when you double the number of threads and continue this up to 64 threads. Recall that the n-body problem cost scales as n^2 where n is the # of particles. Use that to determine how to increase the # of particles per thread.

1. Report a brief report that summarizes the performance results you measured. Create plots showing the strong scaling speed-up and efficiency (see Lecture #1) and the weak scaling efficiency. Use these plots to answer these two central questions: (1) does the n-body algorithm scale in parallel and (2) is the scalability constant or dependent upon the number of threads and particles?

Also include a summary of code modifications that were necessary. Discuss difficulties you encountered and what you did you overcome them. You report should be self-contained in a single document. Incorporate plots into your brief write-up. Do not submit only a spreadsheet with timings.

As with all assignments, you are encouraged to work with a partner. Only one report is needed per group but everyone must submit a note to Sakai indicating with whom you worked, how long this assignment took, and did you run into any problems.