For the 3rd (and last) homework assignment, you will be required to convert the serial (or your OpenMP) n-body problem code into a distributed-memory parallel code using the Message Passing Interface (MPI). Once you have the code working, you will 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 Stampede. There is a slightly modified version of the original n-body problem that we used for HW2 in the hw3/ directory. I've added hints on where you may want to add some code and added some useful functions for MPI parallelism. The starting source file is named mpi_nbody3.cpp though this does not actually use MPI (yet).

2) Verify that the code builds with the MPI-aware compiler and runs using 1 process on a Stampede2 node. The Makefile builds with Intel C++ by default by invoking `mpicxx` which is a wrapper for icpc and hides the tedious paths needed for the MPI library. If you type `mpicxx –version` you'll get the same output as if you typed `icpc –version` …

   To run, you'll need to launch the application using `ibrun -n 1 ./mpi_nbody3 <arguments>`

   This starts "-n #" copies of your binary on the local compute node you've been given. Start with 1 process. You should see the same output as you got from HW2. In fact, you should verify that you get the same answer. (You can compare the last few lines of screen output.)

   Note that Stampede2 (and most of TACC systems) use a very non-standard job launcher called `ibrun`. Most systems use something like `mpirun` but TACC likes to roll their own for whatever reason.

   If you run with -n 2, you should see two copies of the screen output (and possibly unordered). Why? Well, you're running your code at the same time on two processes attached to the same terminal. They don't communicate though so you're just running two copies of the same code doing twice the work.

   As before, you can experiment with different compilers and verify that they all work. Since we need to use the mpicxx wrapper, it's best to use the `module` command to change the environment instead of just invoking the different C++ compiler directly as we have before. The default compiler module is `intel` … type `module list` for a rundown of all of your loaded modules. (Modules are very common on HPC platforms as they make software package management much easier.) To swap out the intel compiler suite for GCC, just type `module swap intel gcc`.

3) Now for the coding … you'll add MPI parallelism to partition the work across multiple processes. Unlike OpenMP, you'll have to define all of the parallelism and move data between processes as needed. For example, if you want each process to handle a portion of a for-loop, you'll have to define the [start,end) indices for each process. To make this easier (and since it's so common), I've created a helpful function name

```
partition_range(const int global_start, const int global_end, const
      int num_partitions, const int rank, int& local_start, int&
      local_end)
```

This takes a global (or serial) iteration range, the total number of partitions that will be used, and the rank of the local partition you're querying, and returns the partitioned [start,end) range. Each process should call this function to get its own range. The partitioned ranges will not overlap and the work is split as uniformly as possible. (The difference between processes – or partitions – will be 1, at most.) That is, if you are using 100 particles and 3 processes – which does not divide evenly, rank-0's range is [0,34), rank-1 is [34,67), and rank-2 [67,100).

There are two broad options for distributed-memory parallel for the n-body problem. In the 1st approach, each process has arrays large enough to store all of the particles but only computes the forces and updates the positions for a subset. At the end of each step, you gather all of the updated positions from all processes so that all processes have the same data arrays before starting the next step. This means that the total memory used increases with the # of MPI processes – not ideal but it's easier to program.

The 2nd option is for each process to only have storage for its own particles. (Using the above partitioning example, this implies that rank-0 stores 34 particles and rank-1 and rank-2 store 33 particles and their local index ranges are [0,34) and [0,33).) This scales in terms of memory usage but is harder to program. This requires that when you compute the forces on the local particles on a given process, you need to get (receive) the particles from every other process and include those particle's contribution to the force. Ideally, you would only store the particles from one sibling process at a time to keep the memory usage scalable. This is not easy and each process ends up sending (NP-1)*(n/NP) pieces of particle data … which is a lot.

So, let's try option #1: the gather approach. This will use only MPI collective operations. You can select which one to use. One tricky part is the initialization. That uses random #'s which are hard to do in parallel. So, initialize the particles on one process and then broadcast the initialized arrays to everyone else. When you call the acceleration, update, and search routines, the outer loop should only be over a subrange of the total iterations. Use the partitioning function I provided. This requires that you determine the rank and total number of MPI processes at run-time. And, as with OpenMP, the search routine will require 1 or more reductions. At the end of each time-step (i.e., after search(…)), gather all of the particles to all the processes. You can use a gather operation followed by a broadcast or you can use one of the "all" variants of the MPI library.

MPI explicitly sends messages and the time spent in the library functions can be directly measured – and adds overhead. Add another timer measurement variable to record the time spent communicating data (i.e., in a MPI library function). Write this time out similar to how the total accel, update, and search times are written.

4) It's time to run another series of strong and weak scalability benchmarks. Since you already have results from OpenMP, let's repeat those same benchmarks.

   Run serial and parallel cases with 1k, 2k, 4k, 8k, and 16k particles using 1, 2, 4, 8, 16, 32, 64, 128, and 256 processes on 4 nodes. Don't bother running the 16k case with 1 ranks as it'll take too long. (It took about 10 minutes with 1 rank.) That is, start with 2 ranks; but, remember this when you compute the speed-up later!!

(Stampede2's nodes have 68 cores but we'll just use 64 ranks per node for now.) To request 4 nodes with 64 ranks per node interactively, type `idev -n 256 -N 4` which means give me 256 ranks total (-n #) and 4 nodes (-N #). The # of ranks per node (64) is just n/N here.

Record the average time per step. Make sure you use enough steps to give consistent results – or reuse what you learned from HW2. These results will be used to compute the strong scalability and efficiency.

Now run the weak scalability test using the same parameters as for the OpenMP study but using up to 256 processes on 4 nodes.

5) Write another brief report that summarizes the performance results you measured. Create plots showing the strong scaling speed-up and efficiency and the weak scaling efficiency just as you did for HW2. Also, plot the communication run-time (which should be less than the total run-time per-step.)

   How does the MPI scalability compare to the OpenMP study? Does the code scale better or worse? What is the ratio of the communication time to the total time and how does that change with the # of processes in the strong and weak scaling studies?

   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.

Extra Credit (5%): Determine the mean, min, and max run-times for the different functions for each process and determine how much variation there is. What is the implication to the overall performance?