CSE 6230 - Spring 2024
Instructor: Prof. Helen Xu

Homework 2 - Shared-memory programming

Due: Feb 6 @ 4:55pm on Gradescope
Deliverables: Code and writeup

This homework is a hands-on introduction to parallel programming in shared memory via **multithreading in C++**.

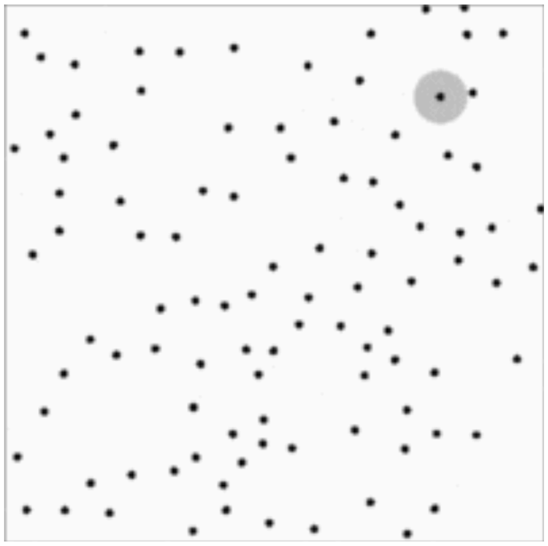
Your homework should be completed **individually** (not in groups).

Code and some parts of the handout are from UC Berkeley CS267 and MIT 6.172.

All instructions in this handout assume that we are running on **PACE ICE**. Instructions for logging in are in HW1.

Overview

In this assignment, we will be parallelizing a toy particle simulation (similar simulations are used in mechanics, biology, and astronomy). In our simulation, particles interact by repelling one another. A snapshot of our simulation is shown here:



The particles repel one another, but only when closer than a cutoff distance highlighted around one particle in grey.

Getting and running the code

Download from git

The starter code is available on Github and should work out of the box. To get started, we recommend you log in to PACE ICE and download the code:

```
$ git clone https://github.com/cse6230-spring24/hw2.git
```

There are five files in the base repository. Their purposes are as follows:

CMakeLists.txt **- you should not modify this file**

The build system that manages compiling your code.

main.cpp **- you should not modify this file**

A driver program that runs your code.

common.h

A header file with shared declarations

job-openmp

A sample job script to run the OpenMP executable

job-serial

A sample job script to run the serial executable

serial.cpp

A simple $O(n^2)$ particle simulation algorithm. It is your job to write an O(n) serial algorithm within the `simulate_one_step` function.

openmp.cpp

A skeleton file where you will implement your OpenMP simulation algorithm. It is your job to write an algorithm within the `simulate_one_step` function.

Build the code

First, we need to make sure that the CMake module is loaded.

```
$ module load cmake
```

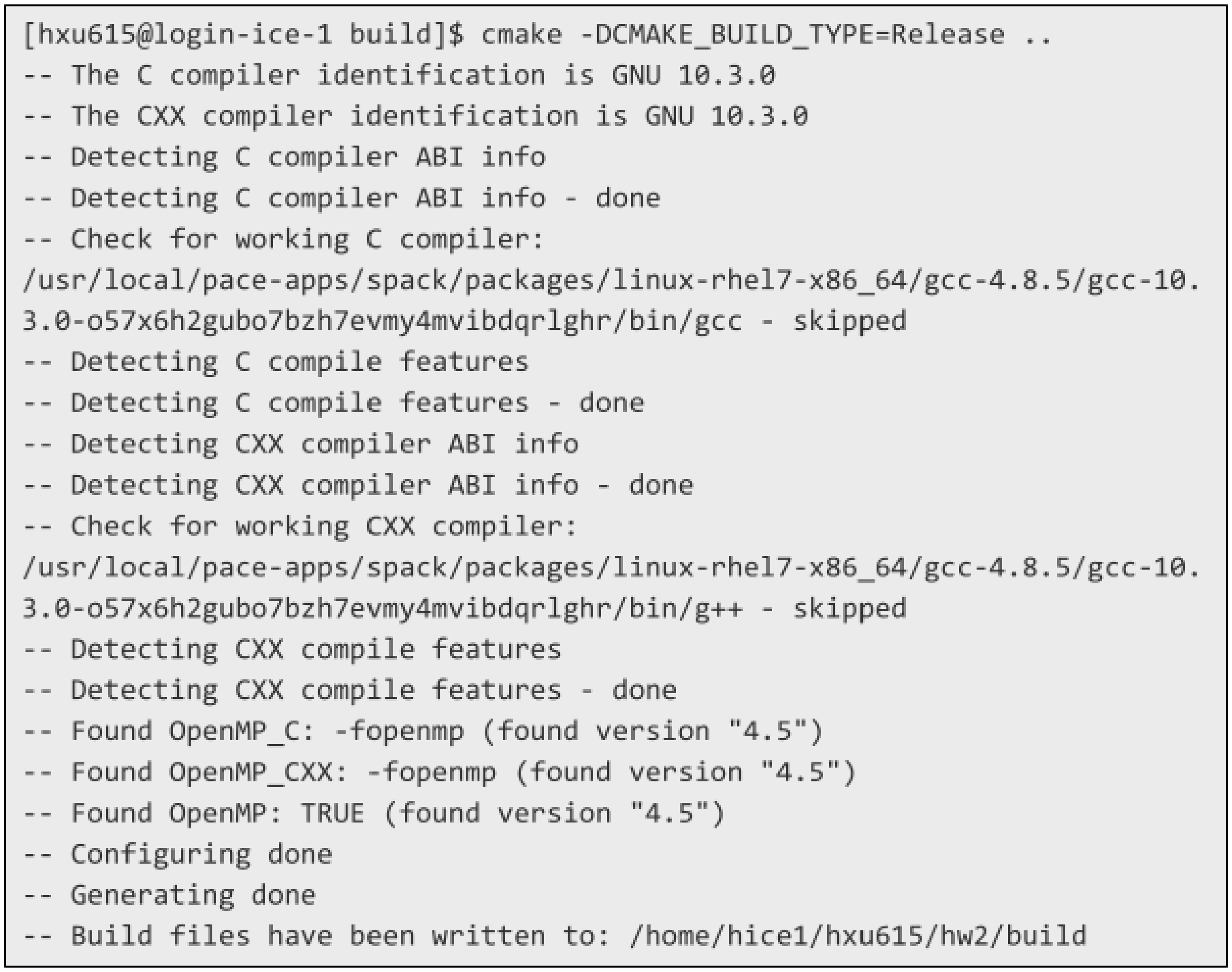
You can put these commands in your `~/.bash_profile` file to avoid typing them every time you log in.

Next, let's build the code. CMake prefers out of tree builds, so we start by creating a build directory.

In the hw2 directory:

```
$ mkdir build
$ cd build
```

Next, we have to **configure our build.** We can either build our code in **Debug** mode or **Release** mode. In debug mode, optimizations are disabled and debug symbols are embedded in the binary for easier debugging with GDB. In release mode, optimizations are enabled, and debug symbols are omitted. For example:



```
[hxu615@login-ice-1 build]$ cmake -DCMAKE_BUILD_TYPE=Release ..
-- The C compiler identification is GNU 10.3.0
-- The CXX compiler identification is GNU 10.3.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler:
/usr/local/pace-apps/spack/packages/linux-rhel7-x86_64/gcc-4.8.5/gcc-10.
3.0-o57x6h2gubo7bzh7evmy4mvibdqrlghr/bin/gcc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler:
/usr/local/pace-apps/spack/packages/linux-rhel7-x86_64/gcc-4.8.5/gcc-10.
3.0-o57x6h2gubo7bzh7evmy4mvibdqrlghr/bin/g++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found OpenMP_C: -fopenmp (found version "4.5")
-- Found OpenMP_CXX: -fopenmp (found version "4.5")
-- Found OpenMP: TRUE (found version "4.5")
-- Configuring done
-- Generating done
-- Build files have been written to: /home/hice1/hxu615/hw2/build
```

Once our build is configured, we may actually **execute** the build with **make**:

```
[hxu615@login-ice-1 build]$ make
[ 16%] Building CXX object CMakeFiles/serial.dir/main.cpp.o
[ 33%] Building CXX object CMakeFiles/serial.dir/serial.cpp.o
[ 50%] Linking CXX executable serial
[ 50%] Built target serial
[ 66%] Building CXX object CMakeFiles/openmp.dir/main.cpp.o
[ 83%] Building CXX object CMakeFiles/openmp.dir/openmp.cpp.o
[100%] Linking CXX executable openmp
[100%] Built target openmp
```

We now have two **binaries** (openmp and serial) and two job scripts (job-openmp and job-serial).

Running the code

Both executables have the same command line interface. Without losing generality, we discuss how to operate the serial program here. Just as in the first homework, we first need to **allocate an interactive node** and then run the program (**warning:** do not run on the login nodes. The benchmark will yield an incorrect result, and you will slow system performance for all users).

The full documentation about how to request allocations on PACE ICE can be found here.

For example, to ask for 1 node (-N1) with 1 task (--ntasks-per-node) (since we are not doing distributed) and 8 cores per task (-c) for 1 hour (-t), use the following:

```
[hxu615@login-ice-2 ~]$ salloc -N1 -c8 --ntasks-per-node=1 -t1:00:00
salloc: Pending job allocation 263415
salloc: job 263415 queued and waiting for resources
salloc: job 263415 has been allocated resources
salloc: Granted job allocation 263415
salloc: Nodes atl1-1-02-003-19-1 are ready for job
---------------------------------------
Begin Slurm Prolog: Jan-14-2024 15:33:56
Job ID:    263415
User ID:   hxu615
Account:   coc-cse
Job name:  interactive
Partition: coc-cpu,ice-cpu
---------------------------------------
```

To run the serial binary, go to **`hw2/build`** and run it like so:

```
[hxu615@atl1-1-02-003-19-1 build]$ ./serial
Simulation Time = 1.72535 seconds for 1000 particles.
```

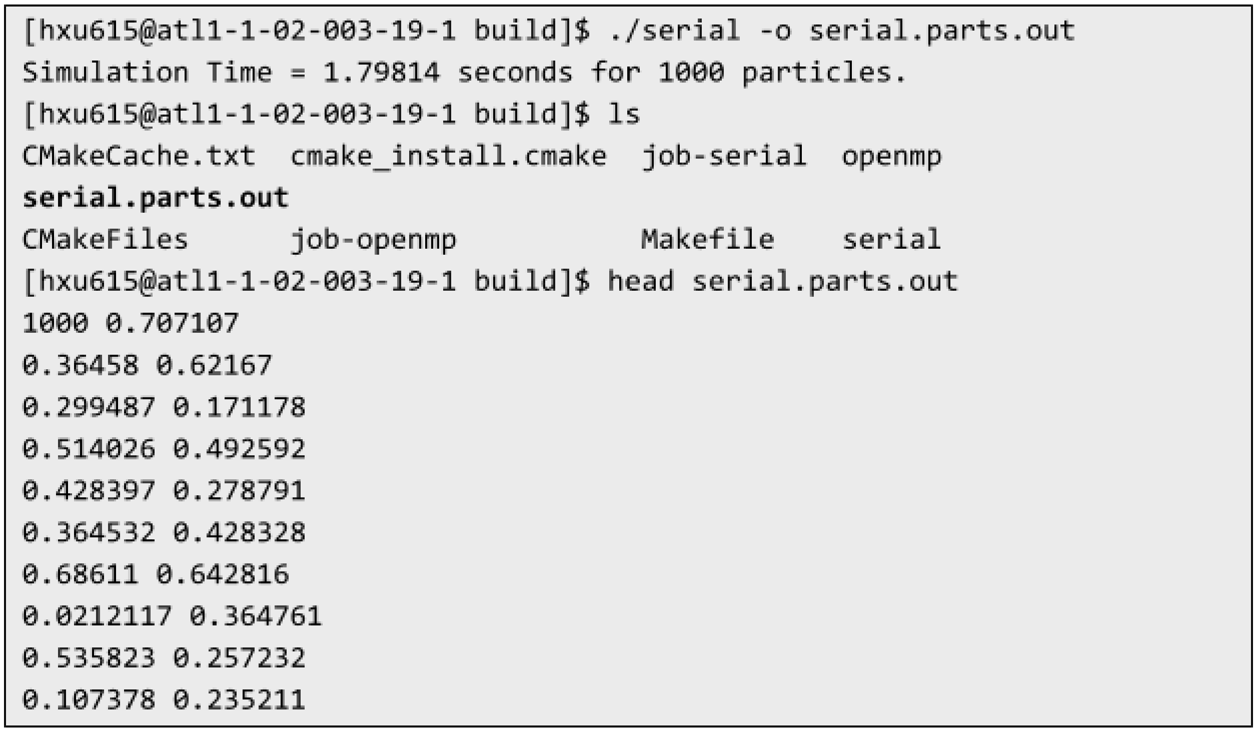
By default, the program runs with 1000 particles. You can change the number of particles with the “-n” command line parameter:

```
[hxu615@atl1-1-02-003-19-1 build]$ ./serial -n 10000
Simulation Time = 168.818 seconds for 10000 particles.
```

If we rerun the program, the initial positions and velocities of the particles will be randomized because the particle seed is unspecified. By default, the particle seed will be unspecified; this can be changed with the "-s" command line parameter:

```
[hxu615@atl1-1-02-003-19-1 build]$ ./serial -s 150
Simulation Time = 1.72026 seconds for 1000 particles.
```

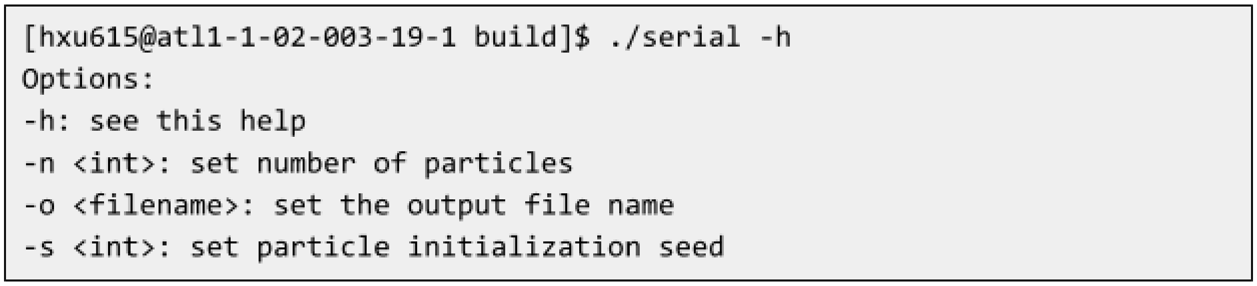
This will set the particle seed to 150 which initializes the particles in a reproducible way. We will test the correctness of your code by randomly selecting several particle seeds and ensuring the particle positions are correct when printed with the "-o" command line parameter. You can print the particle positions to a file specified with the "-o" parameter:



```
[hxu615@atl1-1-02-003-19-1 build]$ ./serial -o serial.parts.out
Simulation Time = 1.79814 seconds for 1000 particles.
[hxu615@atl1-1-02-003-19-1 build]$ ls
CMakeCache.txt  cmake_install.cmake  job-serial  openmp
serial.parts.out
CMakeFiles      job-openmp           Makefile    serial
[hxu615@atl1-1-02-003-19-1 build]$ head serial.parts.out
1000 0.707107
0.36458 0.62167
0.299487 0.171178
0.514026 0.492592
0.428397 0.278791
0.364532 0.428328
0.68611 0.642816
0.0212117 0.364761
0.535823 0.257232
0.107378 0.235211
```

This will create a `serial.parts.out` file with the particle positions after each step listed. You can use the `hw2-rendering` tool to convert this into a .gif file of your particles. See the below section on Rendering Output for more information.

You can use the "-h" command line parameter to print the help menu summarizing the parameter options:

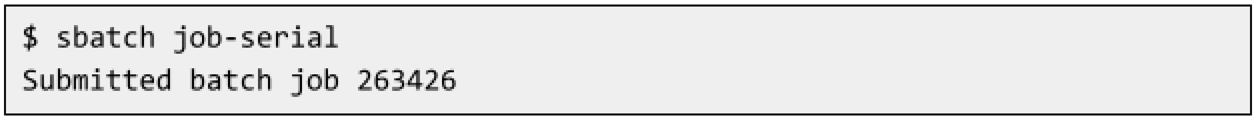


```
[hxu615@atl1-1-02-003-19-1 build]$ ./serial -h
Options:
-h: see this help
-n <int>: set number of particles
-o <filename>: set the output file name
-s <int>: set particle initialization seed
```

Submitting jobs via slurm

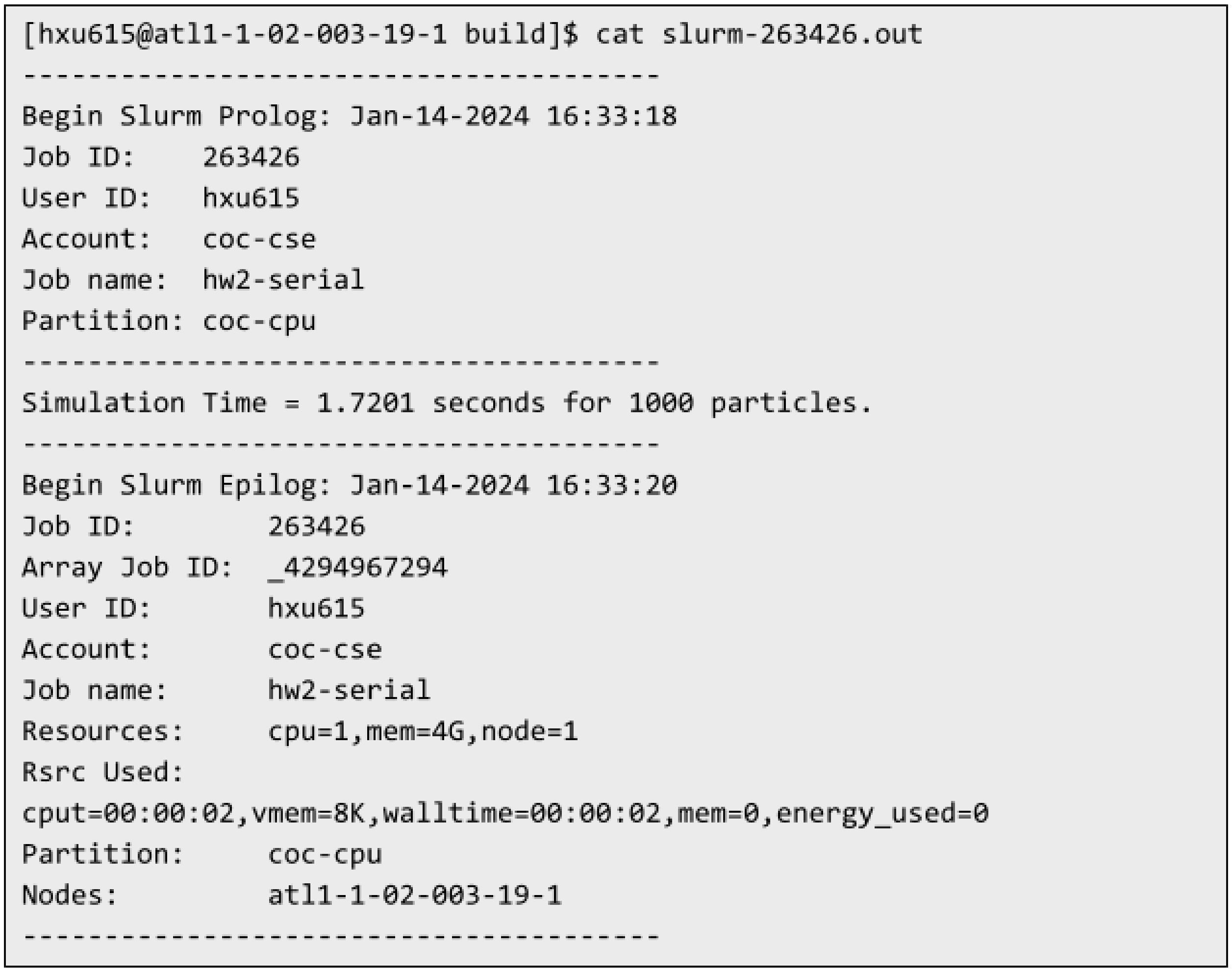
For benchmarking, you do not necessarily have to sit and wait for your job to finish on an interactive node. You can submit jobs to a queue with the job-serial and job-openmp scripts (that should appear in build/ after you build the binaries).

For example, from the `build/` directory (after you have run make), run the script with `sbatch`:



```
$ sbatch job-serial
Submitted batch job 263426
```

You should then see a corresponding output file in the same directory of the form `slurm-<job-id>.out`. For this example, the job id is 263426, so we get the file `slurm-263426.out`:



```
[hxu615@atl1-1-02-003-19-1 build]$ cat slurm-263426.out
---------------------------------------
Begin Slurm Prolog: Jan-14-2024 16:33:18
Job ID:    263426
User ID:   hxu615
Account:   coc-cse
Job name:  hw2-serial
Partition: coc-cpu
---------------------------------------
Simulation Time = 1.7201 seconds for 1000 particles.
---------------------------------------
Begin Slurm Epilog: Jan-14-2024 16:33:20
Job ID:        263426
Array Job ID:  _4294967294
User ID:       hxu615
Account:       coc-cse
Job name:      hw2-serial
Resources:     cpu=1,mem=4G,node=1
Rsrc Used:
cput=00:00:02,vmem=8K,walltime=00:00:02,mem=0,energy_used=0
Partition:     coc-cpu
Nodes:         atl1-1-02-003-19-1
---------------------------------------
```

Each output file has a prolog and epilog of metadata about the job and resources that it took up. The middle part (“Simulation Time…”) is the output of the program.

You can do the same with the `job-openmp` script.

To edit either of the slurm job scripts, edit `job-serial` or `job-openmp` at the top level `hw2/` directory, and rerun `make` in the hw2/`build/` directory.

You can also add slurm scripts as you see fit (e.g., to run the scaling experiments described below).

Rendering output

The output files that are produced from running the program with the "-o" command line parameter can be fed into the hw2-rendering tool made available to convert them into .gif files. These animations will be a useful tool in debugging. To get started, go to your home directory and clone the `hw2-rendering` repo in a convenient location:

```
$ git clone https://github.com/cse6230-spring24/hw2-rendering.git
```

This tool uses python. This can be loaded on PACE ICE with the following command:

```
$ module load anaconda3/2022.05.0.1
```

We can then convert the output files to gifs with the following command (I ran it from within `hw2-rendering` which I had cloned into my home directory, but you can run it from anywhere if you adjust the paths):

```
$ python render.py ~/hw2/build/serial.parts.out particles.gif 0.01
```

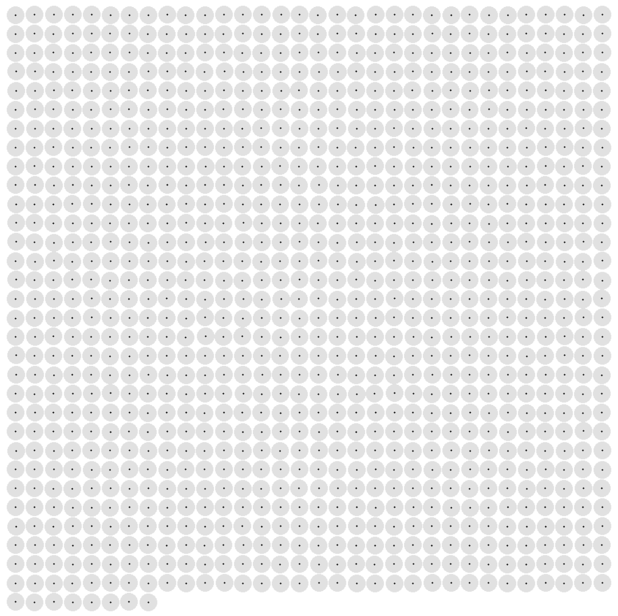
Here serial.parts.out is an output file from the "-o" command line parameter. You should find a `particles.gif` file in your directory. The number 0.01 is the cutoff distance (will be drawn around each particle).

To get particles.gif on your local machine, here is an example of how to get it with scp:

```
$ scp hxu615@login-ice.pace.gatech.edu:~/hw2-rendering/particles.gif .
```

Replace the username and paths with your own.

Here is the example particles.gif I got:



You can also view the gif with a remote desktop (e.g., via PACE Ondemand: https://ondemand-ice.pace.gatech.edu/)

Output correctness

The output files that are produced from running the program with the "-o" command line parameter can be fed into the hw2-correctness tool made available to perform a correctness check. This is the same correctness check we will be performing when grading the homework, however, we will randomly select the particle seeds. To get started clone the hw2-correctness repo:

```
$ git clone https://github.com/cse6230-spring24/hw2-correctness.git
```

This tool uses python. This can be loaded on PACE ICE with the following command:

```
$ module load anaconda3/2022.05.0.1
```

We can then test the output files for correctness with the following command: **make sure to allocate an interactive node first!**

```
$ python correctness-check.py <file-you-know-is-correct>
<file-you-want-to-test>
```

Here is a concrete example:

```
(base) [hxu615@login-ice-2 hw2-correctness]$ python correctness-check.py
verf.out ~/hw2/build/serial.correct.out
Check between verf.out and serial.correct.out succeeded!
(base) [hxu615@login-ice-2 hw2-correctness]$ python correctness-check.py
verf.out ~/hw2/build/serial.incorrect.out
Traceback (most recent call last):
  File "/home/hice1/hxu615/hw2-correctness/correctness-check.py", line
84, in <module>
    check_conditions( avg_dists )
  File "/home/hice1/hxu615/hw2-correctness/correctness-check.py", line
68, in check_conditions
    assert( np.mean( avg_dists[:50] ) < 3e-7 )
AssertionError
```

Here, `verf.out` is provided in the `hw2-correctness` repo and it is the correct output with particle seed set to 1 (i.e., with "-s 1").

The file `serial.correct.out` was the output from the baseline serial binary run with "`-s 1 -o serial.correct.out`". The file `serial.incorrect.out` was the output from the baseline serial binary run with "`-s 2 -o serial.incorrect.out`" (both of these were run as examples to demonstrate the functionality).

This can be substituted for any output you wish to test the correctness for. The correct results can be generated from the provided O(n^2) serial implementation. Remember to specify a particle seed with "-s" to ensure the same problem is solved between the two output files.

Important notes for performance

There will be two types of scaling that are tested for your parallel codes:

- In strong scaling, we keep the problem size constant but increase the number of processors.
- In weak scaling, we increase the problem size proportionally to the number of processors so the work/processor stays the same (Note that for the purposes of this assignment we will assume a linear scaling between work and processors)

While the scripts we are providing have a small number of particles (1000) to allow for the O(n^2) algorithm to finish execution, the final codes should be tested with values much larger (50000-1000000) to better see their performance.

Workflow tips

When you first start implementing the serial version, you only need to allocate one core in your interactive jobs.

When testing your parallel version, first test with a smaller number of cores (e.g., 2, 4, 8, etc.) using interactive nodes before allocating 64 cores.

After debugging on small core counts interactively, you can submit job scripts for 64 cores.

PACE-ICE has 12 64-core machines. Please try to not wait until the last moment to do your scalability plots to avoid waiting in a long queue for them.

Grading and deliverables

This homework has two deliverables: the code and the writeup.

Code

We will grade your code by measuring the scaling of both the OpenMP and serial implementations, and benchmarking your code's raw performance. To benchmark your code, we will compile it with the exact process detailed above, with the GNU compiler.

If you want to come up with faster methods to compute the particle repulsion force function (i.e. rearranging the arithmetic, changing the formula, or using some fancy instructions), make sure it still passes the correctness tests. Small differences in the floating point position values begin to add up until the simulation output diverges from our ground truth (even though your method of computation might be more accurate than ours). Since (a) the point of the assignment is to explore OpenMP parallelism, and (b) we can't anticipate every possible way to compute this force function, here is the rule: if it doesn't pass the correctness check we provide you reliably, then it's not allowed.

Writeup

Please also submit a writeup that contains:

- **A runtime plot** in log-log scale that shows that your serial codes run in O(n) time and a description of the data structures that you used to achieve it.
- **A runtime plot** in log-log scale that shows that your parallel codes run in O(n) time and a description of the synchronization you used in the shared-memory implementation.
- A description of the **design choices** that you tried and how they affect the performance.
- **A strong scaling plot** showing how your OpenMP code scales with fixed input size and more threads **up to 64 threads**. Discuss how close it is to the idealized p-times speedup and whether it is possible to do better.
- **A weak scaling plot** showing how your OpenMP code scales when both the problem size and number of threads increase **up to 64 threads**. For example, for each doubling of the input size, also double the number of threads.

The exact problem sizes and core counts you should run on are given in the following sections. **Please report on the core counts and problem sizes shown below in your**

writeup plots for uniformity between students. You are welcome to try other sizes and core counts independently.

Here is an article with more details about strong and weak scaling.

Generating a serial runtime plot

Please use the following sizes to generate a plot that shows how your serial algorithm scales with problem size:

```
$ ./serial -n 10000 // the best serial algorithm you have for the
problem
$ ./serial -n 20000
$ ./serial -n 40000
$ ./serial -n 80000
$ ./serial -n 160000
$ ./serial -n 320000
```

You should end up with a plot where the x axis is problem size (number of particles) and the y axis is runtime.

Generating a parallel runtime plot

Please use the following sizes to generate a plot that shows how your parallel algorithm scales with problem size.

First, **allocate a machine with 64 cores** (see above instructions in “Running the code”, but with -c64 instead of -c8). Then, either manually run or write a slurm job script (see “Submitting jobs via slurm” above) with the following commands:

```
$ export OMP_NUM_THREADS=64 // use all the threads
$ ./openmp -n 100000
$ ./openmp -n 200000
$ ./openmp -n 400000
$ ./openmp -n 800000
$ ./openmp -n 1600000
$ ./openmp -n 3200000
```

You should end up with a plot where the x axis is problem size (number of particles) and the y axis is runtime.

Generating a strong scaling plot

The simplest way to generate a strong scaling plot is to use different settings of OMP_NUM_THREADS and rerun your code.

First, **allocate a machine with 64 cores** (see above instructions in “Running the code”, but with -c64 instead of -c8). Then, either manually run or write a slurm job script (see “Submitting jobs via slurm” above) with the following commands:

```
$ ./serial // the best serial algorithm you have for the problem
$ export OMP_NUM_THREADS=2
$ ./openmp -n 500000
$ export OMP_NUM_THREADS=4
$ ./openmp -n 500000
... FILLIN FOR POWER OF 2 NUMBER OF CORES BETWEEN 4 AND 64
$ export OMP_NUM_THREADS=64
$ ./openmp -n 500000
```

Each one of the numbers that will be output with commands like the above is one point in your speedup plot (for that corresponding number of threads). That is, **the x axis is the number of threads, and the y axis is (multiplicative) speedup over the serial runtime.** For a given point with x = p threads, the corresponding y point is T_1/T_p (serial runtime over runtime on p threads).

Generating a weak scaling plot

The simplest way to generate a weak scaling plot is to use different settings of OMP_NUM_THREADS and rerun your code with different sizes.

First, **allocate a machine with 64 cores** (see above instructions in “Running the code”, but with -c64 instead of -c8). Then, either manually run or write a slurm job script (see “Submitting jobs via slurm” above) with the following commands:

```
$ ./serial -n 10000 // the best serial algorithm you have for the
problem
$ export OMP_NUM_THREADS=2
$ ./openmp -n 20000
$ export OMP_NUM_THREADS=4
$ ./openmp -n 40000
… CONTINUE INCREASING CORES AND PROBLEM SIZE BY POWERS OF 2
$ export OMP_NUM_THREADS=64
$ ./openmp -n 640000
```

Each one of the numbers that will be output with commands like the above is one point in your weak scaling plot (for that size and number of threads). That is, **the x axis is the number of threads, and the y axis is (multiplicative) scaled speedup over the serial version**. For weak scaling, scaled speedup is defined as efficiency * p, where p is the number of threads, and efficiency is defined as T_1/T_p.

Point distribution

Correctness of your implementation is a precondition to get any marks.

Serial part - 2 points - includes:

- Correct and linear-time serial implementation
- Performance plot in the writeup to show the linear-time performance, x axis = number of points in simulation, y axis = time)
- Description of how you achieved the linear-time algorithm

Parallel part - 3 points - includes:

- Correct parallel implementation based on the linear-time serial implementation
- Description in the writeup of how you achieved the parallelization
- Strong scaling plot in the writeup, as described above
- Weak scaling plot in the writeup, as described above

Code walkthrough

The file `common.h` defines every particle in 2D with a position, velocity, and acceleration in x and y dimensions:

```
// Particle Data Structure
typedef struct particle_t {
    double x;  // Position X
    double y;  // Position Y
    double vx; // Velocity X
    double vy; // Velocity Y
    double ax; // Acceleration X
    double ay; // Acceleration Y
} particle_t;
```

At each time step, once the force interactions have been calculated, for every particle, the simulation uses the acceleration to compute the velocity, and the velocity to compute the next position. Here is part of the move function from serial.cpp that computes the velocity and position for the next timestep:

```
    p.vx += p.ax * dt;
    p.vy += p.ay * dt;
    p.x += p.vx * dt;
    p.y += p.vy * dt;
```

The goal of this homework is to more efficiently compute the force interactions that determine p.ax and p.ay in each timestep.

If you look in common.h, it also contains several constants to parametrize the particle simulation:

```
#define nsteps   1000
#define savefreq 10
#define density  0.0005
#define mass     0.01
#define cutoff   0.01
#define min_r    (cutoff / 100)
#define dt       0.0005
```

The density is used in main.cpp like so to determine the spatial size of the bounding box of the simulation given the number of particles:

```
    double size = sqrt(density * num_parts);

    particle_t* parts = new particle_t[num_parts];

    init_particles(parts, num_parts, size, part_seed);
```

The cutoff is a constant used to determine the minimum distance that two particles should be from each other to interact, and min_r defines the minimum distance that we are allowing particles to be in the computation of the force interaction. The function fmax is a C++ function that takes the max of two floats. The force interaction (coef) is determined by the distance between the two particles (r and r2), and the mass (another constant defined in common.h).

Here is the relevant code from serial.cpp:

```
// Apply the force from neighbor to particle
void apply_force(particle_t& particle, particle_t& neighbor) {
    // Calculate Distance
    double dx = neighbor.x - particle.x;
    double dy = neighbor.y - particle.y;
    double r2 = dx * dx + dy * dy;

    // Check if the two particles should interact
    if (r2 > cutoff * cutoff)
        return;

    r2 = fmax(r2, min_r * min_r);
    double r = sqrt(r2);

    // Very simple short-range repulsive force
    double coef = (1 - cutoff / r) / r2 / mass;
    particle.ax += coef * dx;
    particle.ay += coef * dy;
}
```

The initial serial algorithm in simulate_one_step in serial.cpp performs a loop over all pairs of particles and calls apply_force, which checks within the function if the particles are close enough to interact.

```
    for (int i = 0; i < num_parts; ++i) {
        parts[i].ax = parts[i].ay = 0;
        for (int j = 0; j < num_parts; ++j) {
            apply_force(parts[i], parts[j]);
        }
    }
```

Part 1: Optimizing the serial implementation

You may be tempted to jump directly into parallelizing your code as a means to gain performance. Generally, however, the best parallel programs are those that are first highly optimized serially and then made parallel afterwards. Consequently, we recommend when approaching this homework to make the serial version of the simulation as fast as possible before going on with parallelization.

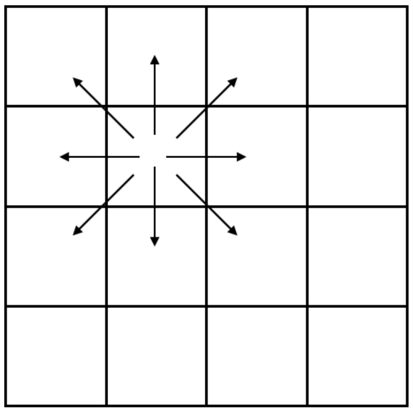
A naive algorithm to compute the forces on the particles by iterating through every pair of particles, then we would expect the asymptotic complexity of our simulation to be $O(n^2)$. This is the algorithm implemented in the starter code.

However, in our simulation, we have chosen a density of particles sufficiently low so that with *n* particles, we expect only *O*(*n*) interactions. An efficient implementation can reach this time complexity. The first part of your assignment will be to implement this linear time solution in a serial code, given a naive $O(n^2)$ implementation.

Note: The algorithm does not have to run in O(n) in the worst case, but should be linear in the number of particles in the common/average practical case.

You will begin by implementing a data structure to reduce the total number of pairwise interactions that must be checked each step of the simulation. You're free to implement any data structure you like to make the simulation faster.

Suggestion: Reduce the work by eliminating unnecessary checks between particles that are too far to possibly interact. One possibility is dividing up the entire simulation box into a 2D grid (as shown in the diagram) such that particles in one block can only interact with other particles within the same block or within immediately adjacent blocks.



Example of one possible division of the simulation into blocks.

Some things to think through and **discuss in the writeup**:

- What is the minimum size (spatial) a block needs to be?
- What is the benefit of a smaller block size vs a larger block size?
- How do you choose a block size?
- How should you store the points within a block? That is, what data structure is most efficient for maintaining the list of points contained in a spatial block in the grid? NOTE: the easiest way to get a working implementation is to use an out-of-the-box dynamic data structure, such as C++'s std::set.

Rewrite the particle simulation to be much more efficient. You should edit the `simulate_one_step` (and possibly the `init_simulation`) functions in `serial.cpp`. In terms of the simulation itself, you should only need to modify the algorithm on lines 54-59 in serial.cpp (not the code that moves the particles). If you want to add additional higher-level data (e.g., structs), you may also edit common.h.

You should not, at this point, use OpenMP (or any other method) to parallelize your particle simulation. The simulation will be parallelized in Part 2 of this homework.

Part 2: Parallelizing the simulation

Suppose we have a code that runs in time T on a single processor. Then we'd hope to run close to time T/p when using p processors. After implementing an efficient serial solution, you will attempt to reach this speedup using OpenMP.

You should edit openmp.cpp to implement a parallel version of the simulation based on the efficient serial implementation you wrote in Part 1.

Parallelize your code by inserting OpenMP primitives to the regions of code you have identified as worth parallelizing. You may need to add synchronization primitives (e.g., locks) to your data structure as well.

For any changes you make, verify that your code is still correct via the instructions in "Output correctness" above.

Helpful resources/tips

- Programming in shared memory with OpenMP are introduced in lecture (slides on Canvas)
- OpenMP tutorial and OpenMP specifications
- Shared memory implementations may require using locks that are available as omp_lock_t in OpenMP (requires omp.h)
- You may consider using atomic operations such as __sync_lock_test_and_set with the GNU compiler.
- For the serial version, you can (and should) profile it using Valgrind as we did in the first homework.
- For the parallel version, Valgrind will allow you to run a multithreaded version through it, but will serialize the threads. On PACE ICE, you can use Intel Vtune, which is compatible with multithreaded codes.

That's the end of this homework! Submit your writeup and code as described in the "Homework submission instructions" below:

+

Homework submission instructions

There are two parts to the homework: a writeup and the code. The writeup should be in pdf form, and the code should be in zip form. **There will be two submission slots** in Gradescope called "Homework 2 - writeup" and "Homework 2 - code". You should submit the writeup and code separately in their respective slots.

To get the code from PACE ICE to your local machine to submit, first zip it up on PACE ICE:

```
$ zip -r hw2.zip hw2/
```

Then from the terminal (or terminal equivalent) your local machine (e.g., your personal laptop), scp the code over.

```
$ scp <your-username-here>@login-ice.pace.gatech.edu:<path-to-hw2.zip>
<local-file-location>
```

For example, if I had `hw2.zip` in my home directory on PACE ICE and wanted to copy it into my current directory on my local machine, I would run:

```
$ scp hxu615@login-ice.pace.gatech.edu:~/hw2.zip .
```