Contents

1 Overview

The purpose of this assignment is introduction to programming in shared and distributed memory models and on GPUs. Your goal is to parallelize a toy particle simulator (similar particle simulators are used in mechanics, biology, astronomy, etc.) that reproduces the behavior shown in the following animation:

The range of interaction forces is limited as shown in grey for a selected particle. Density is set sufficiently low so that given n particles, only O(n) interactions are expected. Suppose we have a code that runs in time T = O(n) on a single processor. Then we’d hope to run in time T/p when using p processors. We’d like you to write parallel codes that approach these expectations. Don’t forget to use the no output `-no` option for your actual timing runs you use in these calculations and plots.

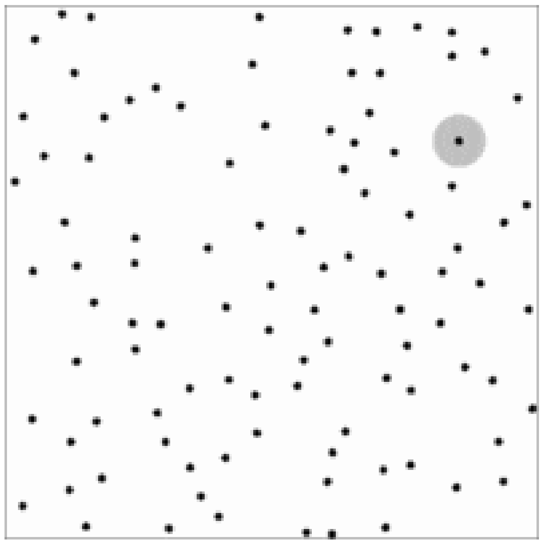

Figure 1: Animation

2 Remote XSEDE/Moodle Students: Please Read

Dear Remote Students, we are thrilled to be a part of your parallel computing learning experience and to share these resources with you! To avoid confusion, please note that the assignment instructions, deadlines, and other assignment details posted here were designed for the local UC Berkeley students. You should check with your local instruction team about submission, deadlines, job-running details, etc. and utilize Moodle for questions. With that in mind, the problem statement, source code, and references should still help you get started (just beware of institution-specific instructions). Best of luck and we hope you enjoy the assignment!

3 Correctness and Performance

A simple correctness check which computes the minimal distance between 2 particles during the entire simulation is provided. A correct simulation will have particles stay at greater than 0.4 (of cutoff) with typical values between 0.7-0.8. A simulation were particles don't interact correctly will be less than 0.4 (of cutoff) with typical values between 0.01-0.05 . More details as well as an average distance are described in the source file.

The code we are providing will do this distance check based on the calls to the interaction function but the full autograder will do the checks based on the outputted txt file. We'd recommend keeping the correctness checker in your code (for the OpenMP and MPI codes the overhead isn't significant) but depending on performance desires it can be deleted as long as the simulation can still output a correct `.txt` file.

The performance of the code is determined by doing multiple runs of the simulation with increasing particle numbers and comparing the performance with the `autograder.cpp` provided. This can be done automatically with the `auto-*` scripts. However, make sure your code is fully tested before running the autograders at full scale (repeatedly running the autograders unmodified for testing can eat up your compute hours allocation quickly).

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)

For more details on the options provided by each file you can use the `-h` flag on the compiled executables.

Important note for Performance:

While the scripts we are providing have small numbers of particles (500-1000) to allow for the O(n^2) algorithm to finish execution, the final codes should be tested with values 100 times larger (50000-100000) to better see their performance.

4 Teams

You may work in groups of 2 or 3 (no more, no fewer). One person in your group should be a non-EECS student, but otherwise you are responsible for finding a group. The number of non-EECS students is a little less than half the total enrollment, so some teams may not have a non-EECS student. In that case, one person in such team should be a non-CS student. Once you have a group, add yourselves to a bCourses group (we have pre-created empty groups) by following these instructions. Also, we recommend you come to GSI office hours with your group to discuss the distribution of work among team members.

5 Source Code

You may start with the serial and parallel implementations supplied below. All of them run in O(n^2) time, which is unacceptably inefficient. Individual files are described below. **Here is a zip of all of them.**

5.1 Common Files for Parts 1 & 2

- `common.cpp`, `common.h`: an implementation of common functionality, such as I/O, numerics and timing
- `autograder.cpp`: a code that helps calculate performance for both serial and parallel implementations
- `Makefile`: a makefile that should works on all NERSC machines using Slurm if you uncomment appropriate lines
- `job-edison-serial`, `job-edison-pthreads24`, `job-edison-openmp24`, `job-edison-mpi24`: **sample** batch files to launch jobs on Edison. Use `sbatch` to submit on Edison. First, however, adjust the parameters for testing vs. final runs as appropriate.
- `auto-edison-serial`, `auto-edison-pthreads24`, `auto-edison-openmp24`, `auto-edison-mpi24`: **sample** batch files to autograde performance on Edison. Use sbatch to submit on Edison. First, however, adjust the parameters for testing vs. final runs as appropriate.

6 Part 1: Serial & OpenMP

6.1 Source Code (contained in the above zip file under "Source Code")

- `serial.cpp`: a serial implementation
- `openmp.cpp`: a shared memory parallel implementation done using OpenMP

6.2 Part 1 Submission

In total, there are 2 codes and 1 pdf you need to submit. You need to create one serial code that runs in O(n) time, and one shared memory implementation (OpenMP), for this part of the assignment. You also need to submit a short report.

Here are some items you might show in your report:

- A plot in log-log scale that shows that your serial and parallel codes run in O(n) time and a description of the data structures that you used to achieve it.
- A description of the synchronization you used in the shared memory implementation.
- A description of the design choices that you tried and how did they affect the performance.
- Speedup plots that show how closely your OpenMP code approaches the idealized p-times speedup and a discussion on whether it is possible to do better.
- Where does the time go? Consider breaking down the runtime into computation time, synchronization time and/or communication time. How do they scale with p?
- A discussion on using OpenMP
- **Submit the zip file on bCourses by the Part 1 due date (listed on bCourses).**

7 Part 2: MPI

7.1 Source Code (contained in the above zip file under "Source Code")

- `mpi.cpp`: a distributed memory parallel implementation done using MPI

7.2 Submission

For Part 2, in a zip file (see below), submit the files from Part 1, your MPI code for Part 2, and extend your original report to include Part 2. Your MPI code should run in O(n) time with, hopefully, O(n/p) scaling. You are welcome to improve your code from Part 1 though not required; if you do, include a description of your improvements (and why they work) in your report as well.

Your submission should be a single `teamname_hw2.tar.gz` that, when unzips, gives us a folder named `teamname_hw2`. In this folder there should be one subfolder named `part1`, a report file named `report.pdf`, and a text file named `members.txt` containing your teammate's names, one line for each member. We need to be able to build and execute your implementations for you to receive credit. Spell out in your report what `Makefile` targets we are to build for the different parts of your report.

For example, `03_hw2.tar.gz` should have:

```
03_hw2 (a folder)
  |--part1 (a folder)
  |----source files for serial, openmp, mpi
  |----Makefile with make targets serial, openmp, mpi
  |--report.pdf
  |--members.txt
```

And `members.txt` should contain a line for each member and nothing else, for example:

```
Jenny Huang
Marquita Ellis
```

Please follow these formatting/organization conventions or your GSI's will have to do extra busy work, which makes them sad.

Here are some items you might add to your report:

- A plot in log-linear scale that shows your performance as a percent of peak performance for different numbers of processors. You can use a tool like Craypat to tell you how many flops are performed for different sizes of n.
- A description of the communication you used in the distributed memory implementation.
- A description of the design choices that you tried and how did they affect the performance.
- Speedup plots that show how closely your MPI code approaches the idealized p-times speedup and a discussion on whether it is possible to do better.
- Where does the time go? Consider breaking down the runtime into computation time, synchronization time and/or communication time. How do they scale with p?
- A discussion on using MPI.

Submit the zip file on bCourses by the Part 2 due date (listed on bCourses).

8 Part 3: GPU

8.1 Overview

You will also be running this assignment on GPUs. You have access to Texas Advanced Computing Center's Stampede (ranked 7th in the Top 500 Supercomputer list as of November 2014). Each node of Stampede has 2 Intel Sandy Bridge processors and an Intel Xeon

Phi accelerator. 128 nodes are also equipped with a single NVIDIA K20 GPU. Refer to the TACC Stampede User Guide for how to ssh into the system, how to compile CUDA code, how to submit jobs, etc. You can find your Stampede username by logging in to XSEDE User Portal and look under the My XSEDE Resources section.

You can run your gpu program by submitting our provided job script `sbatch job-stampede-gpu` or run interactively by using TACC's idev application. To use `idev`, type `idev -q gpudev` on the login node and wait for your job to start. When you are done, type logout to get back to your login node.

8.2 Source Code

(Click to download full tar)

We have provided a naive O(n^2) GPU implementation, similar to the openmp, and MPI codes listed above. It will be your task to make the necessary algorithmic changes and machine optimizations to achieve favorable performance across a range of problem sizes.

8.3 Help

A simple correctness check which computes the minimal distance between 2 particles during the entire simulation is provided. A correct simulation will have particles stay at greater than 0.4 (of cutoff) with typical values between 0.7-0.8. A simulation were particles don't interact correctly will be less than 0.4 (of cutoff) with typical values between 0.01-0.05 .

Adding the checks inside the GPU code provides too much of an overhead so an autocorrect executable is provided that checks the output txt file for the values mentioned above. We also provide a clean serial implementation that can run on a gpu for you to start from, serial.cu.

8.4 Submission

Similarly to Part 2, submit your files in a single `teamname_hw2.tar.gz` in the following format:

For example, `03_hw2.tar.gz` should have:

```
03_hw2 (a folder)
  |--part2 (a folder)
  |----source files for gpu
  |----Makefile with make target gpu
  |--report.pdf
  |--members.txt
```

And `members.txt` should contain a line for each member and nothing else, for example:

```
Jenny Huang
Marquita Ellis
```

Please follow these formatting/organization conventions or your GSI's will have to do extra busy work, which makes them sad.

Please include a section in your report detailing your GPU implementation, as well as its performance over varying numbers of particles. Here is the list of items you might show in your report:

- A plot in log-log scale that shows the performance of your code versus the naive GPU code
- A plot of the speedup of the GPU code versus the serial, openmp, mpi runs on the CPU of the node
- A description of any synchronization needed
- A description of any GPU-specific optimizations you tried
- A discussion on the strengths and weaknesses of CUDA and the current GPU architecture

Submit the zip file on bCourses by the Part 3 due date (listed on bCourses).

9 Resources

- Programming in shared and distributed memory models have been introduced in Lectures 4 through 7, which are available at the course website.
- Shared memory implementations may require using locks that are availabale as `omp_lock_t` in OpenMP (requires `omp.h`) and `pthread_mutex_t` in pthreads (requires `pthread.h`).
- You may consider using atomic operations such as `__sync_lock_test_and_set` with the GNU compiler. This syntax changes between compilers.
- Distributed memory implementation may benefit from overlapping communication and computation that is provided by nonblocking MPI routines such as `MPI_Isend` and `MPI_Irecv`.
- Other useful resources: pthreads tutorial, OpenMP tutorial, OpenMP specifications and MPI specifications.
- It can be very useful to use a performance measuring tool in this homework. Parallel profiling is a complicated business but there are a couple of tools that can help.

  - IPM is a profiling tool that is inserted into your link command in your `Makefile` (after you module load `ipm`) and instrumented versions of your MPI calls are put into your program for you.
  - TAU (Tuning and Analysis Utilities) is a source code instrumentation system to gather profiling information. You need module load tau to access these capabilities. This system can profile MPI, OpenMP and PThread code, and mixtures, but it has a learning curve.
  - HPCToolkit Is a sampling profiler for parallel programs. You need `module load hpctoolkit`. You can install the `hpcviewer` on your own computer for offline analysis, or use the one on NERSC by using the NX client to get X windows displayed back to your own machine.
  - If you are using TAU or HPCToolkit you should run in your `$SCRATCH` directory which has faster disk access to the compute nodes (profilers can generate big profile files).

- Hints on getting O(n) serial and Shared memory and MPI implementations (pdf)
- NVIDIA CUDA Programming Guide v6.5 (Note that Stampede's default CUDA module is CUDA 5.5, and it only has up to CUDA 6.0, so some of the features you found in this guide might be unavailable.)
- CUDA - Wikipedia
- An Introduction to CUDA/OpenCL and Manycore Graphics Processors