CS 758 Course Wiki: Fall 2016 Main » Homework 2

CS 758: Programming Multicore Processors (Fall 2016)

Homework 2

You should do this assignment alone. No late assignments.

Filelist for the assignment:

- Template files

Purpose

The purpose of this assignment is to explore the features of OpenMP's support for task parallelism, including task management, synchronization, and concurrent data structures.

Programming Environment: OpenMP

OpenMP is a shared-memory programming model that attempts to automatically parallelize code that was written in a (mostly) serial fashion. OpenMP makes extensive use of compiler directives and optimizations, in addition to its own runtime library.

If you have not already done so, it is suggested that you review the OpenMP references provided in the Reading List.

OpenMP uses a Fork/Join model similar to that of P-Threads, but Fork/Join events are more frequent in OpenMP than in most P-Thread based programs. Most OpenMP programs consist of interleaved parallel and sequential sections, with "Fork" events occurring at the start of each parallel section, and "Join" events at the end of each parallel section. In non-parallel sections, only the "master thread" executes.

In order to use the OpenMP environment on ale, students should use the icc complier. Any source files that employ OpenMP directives or library calls must include the omp.h header file. Additionally, the flag -openmp must be passed to icc for both compilation and linking.

Beginning with Version 3.0, OpenMP has provided support for task-level parallelism. OpenMP tasks are similar to Intel's Threaded Building Blocks (TBB), but use cleaner syntax. Information about OpenMP tasks can be found at the following places: Good simple examples Detailed documentation

Programming Task: N-Body Simulation

An n-body simulation calculates the gravitational effects of the masses of n bodies on each others' positions and velocities. The final values are generated by incrementally updating the bodies over many small time-steps. We will look at two approaches to this problem. First, we will calculate the pairwise force exerted on each particle by all other particles, an $O(n^2)$ operation. Second, we will use an quadtree data structure to implement an 0(n log n) approximation algorithm. A great overview of the O (n log n) algorithm can be found here. For simplicity, we will model the bodies in a two-dimensional space.

The physics.

We review the equations governing the motion of the particles according to Newton's laws of motion and gravitation. Don't worry if your physics is a bit rusty; all of the necessary formulas are

included below. We already know each particle's position (r_x, r_y) and velocity (v_x, v_y). To model the dynamics of the system, we must determine the net force exerted on each particle.

- **Pairwise force.** Newton's law of universal gravitation asserts that the strength of the gravitational force between two particles is given by the product of their masses divided by the square of the distance between them, scaled by the gravitational constant G, which is 6.67 × 10-11 N m^2 / kg^2. The pull of one particle towards another acts on the line between them. Since we will be using Cartesian coordinates to represent the position of a particle, it is convenient to break up the force into its x and y components (F_x, F_y) as illustrated below.

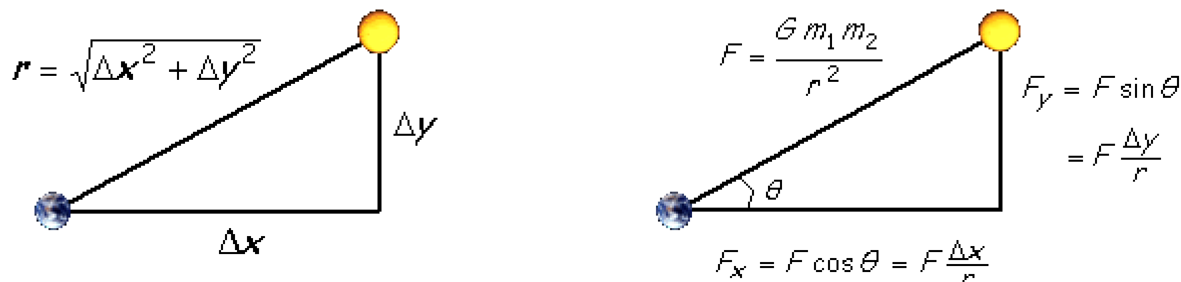

Figure 1: N-Body Calculation

- **Net force.** The principle of superposition says that the net force acting on a particle in the x or y direction is the sum of the pairwise forces acting on the particle in that direction.
- **Acceleration.** Newton's second law of motion postulates that the accelerations in the x and y directions are given by: $a_x = F_x / m$, $a_y = F_y / m$.

The numerics.

We use the leapfrog finite difference approximation scheme to numerically integrate the above equations: this is the basis for most astrophysical simulations of gravitational systems. In the leapfrog scheme, we discretize time, and update the time variable t in increments of the time quantum Δt. We maintain the position and velocity of each particle, but they are half a time step out of phase (which explains the name leapfrog). The steps below illustrate how to evolve the positions and velocities of the particles.

For each particle:

1. Calculate the net force acting on it at time t using Newton's law of gravitation and the principle of superposition.
2. Calculate its acceleration (a_x, a_y) at time t using its force at time t and Newton's second law of motion.
3. Calculate its velocity at time t + Δt / 2 by using its acceleration at time t and its velocity (v_x, v_y) at time t - Δt / 2. Assume that the acceleration remains constant in this interval, so that the updated velocity is: $v_x = v_x + \Delta t\, a_x$, $v_y = v_y + \Delta t\, a_y$.
4. Calculate its position at time t + Δt by using its velocity at time t + Δt / 2 and its position at time t. Assume that the velocity remains constant in the interval from t to t + Δt, so that the resulting position is given by $r_x = r_x + \Delta t\, v_x$, $r_y = r_y + \Delta t\, v_y$. Note that because of the leapfrog scheme, the constant velocity we are using is the one estimated at the middle of the interval rather than either of the endpoints.

As you would expect, the simulation is more accurate when Δt is very small, but this comes at the price of more computation.

Problem 1: Parallelize O(n^2) N-Body

For this problem, you are to parallelize the O(n^2) pairwise version of the N-body simulation using **both** OpenMP parallel for (program 1a) and OpenMP Tasks (program 1b).

You are required to make an argument that your n-body implementations are correct. A good way to do this is to initialize the bodies to a special-case starting condition, and then show that after a number of time steps the state of the grid exhibits symmetry or some other expected property. You need not prove your implementation's correctness in the literal sense. However, please annotate any simulation outputs clearly.

Problem 2: Parallelize O(n log n) N-Body

For this problem, you are to parallelize the O(n log n) pairwise version of the N-body simulation using **both** OpenMP parallel for (program 2a) and OpenMP Tasks (program 2b). You may find the loop version of this gets poor speedup. Don't spend too much time trying to optimize this, as this is mostly an exercise in understanding why tasks are really useful. Instead, you should focus your energy on exploring the many ways to utilize tasks by parallelizing the O(n log n) version of the N-body simulation. You will find that this version of the simulation heavily utilizes recursion; recursive calls often make great tasks.

In this problem you should experiment with the granularity of tasks. Too many tasks leads to high overheads and too few tasks parallelize poorly. You should incrementally modify your parallization strategy until you find one that is a good balance between overhead and parallelization that leads to good performance.

Problem 3: Analysis of N-Body Algorithms

In this section, you will analyze the performance of your three N-Body implementations.

Part A:

Plot the normalized (versus the serial n^2 version) speedups of programs 1a and 1b on the same graph for N=[1,2,4,8,16,24,32,...,256] threads for **4096** bodies and **100** time steps. The value of dt is irrelevant to studying scalability: with the number of time steps held constant, it only affects the length of time simulated, not the duration of the simulation itself. Thus you may choose any value you like. Also, remember that the execution time for 1 thread comes from the sequential version of the program.

Part B:

Plot the normalized (versus the serial n log n version) speedups of Programs 2a and 2b on N=[1,2,4,8,16,24,32,...,256] threads for **65536** bodies and **100** time steps.

Part C:

Plot the execution time of Programs 1a, 1b, and 2b on N=[1,2,4,8,16,24,32,...,256] threads for **4096** bodies and **100** time steps on the same graph.

Problem 4: Questions (Submission Credit)

1. In problem 1, which (OMP parallel for or OMP Tasks) had better performance? Why do you think that is?
2. Describe your parallelization strategy for both problems 1 and 2. Which was easier? Which scaled better?
3. Comment on the relative speedups between parallelization strategies. Was the speedup worth the additional effort of the more difficult program?

Source Code

We will provide you with working implementations of both n^2 and n log n n-body simulations found here.

This assigment can be completed by implementing new subclasses of the NbodySimulator class. See the existing code/Makefile for more direction.

Tips and Tricks

- **Start early.**
- Example of fibonacci computation using openmp tasks
- Explained slightly in a short slide deck
- Another slide deck that explains Tasks
- Slide deck on tree traversal using openmp

What to Hand In

A tarball of your entire source code including a Makefile and a README file, should be emailed to the professor before the beginning of lecture. The README should include 1) directions for compiling your code, 2) directions for running your code, 3) any other comments. Use subject line **[CS758] Homework 2**, so that email filters work properly.

Please turn in a homework write-up on **paper** at the beginning of lecture. You must include:

- A printout of your parallel implementation of Programs 1a and 1b. Only include relevant code.
- A printout of the parallel implementation of Program 2. Only include relevant code.
- Arguments for the correctness of Programs 1 and 2.
- The plots as described in Problem 3a and 3b, including labels describing your data.
- Answers to the questions in Problem 4.

Important: Include your name on EVERY page.

Page last modified on September 22, 2016, at 02:25 PM, visited times