Parallel Algorithms (15-499), Spring 09
Assignment #3 **Due: February 12th and February 19th**

Getting Started

This assignment involves programming algorithms in Cilk++. To run your code we will give you an account on `multi6.aladdin.cs.cmu.edu`, which is a linux machine with 16 hardware threads (8 cores each which have 2 "hyperthreads"). In the assignment you can work in groups of 2 or 3. The assignment is broken into two parts. The first part consists of problems 1–3 and is due on February 12th. The second part (problem 4) is due on the 19th.
You should first copy over the assignment code directory to one of your directories.

```
cp -r /afs/cs/academic/class/15499-s09/assignments/hw3 .
```

This directory contains four sub-directories (quicksort, matrix_transpose, mergesort, and barnes_hut) corresponding to the four problems. Note that multi6 has afs so you can copy it to an afs directory and use it on multi6 if you want.
To run Cilk++ you need to add the Cilk binary path to your $PATH environment variable. With the bash shell (the default on multi6) you can do this with:

```
export PATH=/afs/cs/academic/class/15499-s09/cilk64/bin:$PATH
```

It is best if you add the line to your .bashrc file.

Problems

Please do the following problems and submit a group writeup at the start of class. Also copy any code files to the directory:

`/afs/cs.cmu.edu/academic/class/15499-s09/handin/`*username*`/hw3/`

where *username* is your Andrew ID. Only one project member needs to hand in the assignment. The first three problems are taken from the MIT course 6.197 from Fall 2008 (Amarasinghe, Leiserson, Rinard). Leiserson is a founder of Cilk Arts which developed Cilk++, and also an author of the algorithms textbook you probably used in 15-451.

Problem 1: Quicksort

Introduction to a Cilk Program In this problem, you will experiment with an existing Cilk++ project, and learn how to use the CilkScreen Race Detector. You should make running your Cilk applications through CilkScreen a standard practice.

1. Build the qsort binary by running make. This will produce a parallel quicksort binary. The binary's command line syntax is:

   ```
   ./qsort [ndata] -cilk_set_worker_count [n]
   ```

   - `ndata`: Number of data points. Default = 10,000,000
   - `n`: The number of workers. Default = # of machine threads on your system (16 on multi6).

Run qsort with the 10,000,000 data points (the default) using 1 through 16 operating system threads. What are the execution times? What happens when you run qsort with more system threads than machine threads (16 on multi6)?

2. Add the following line to the start of the file to introduce a race condition in the parallel code:

   ```
   #define INTENTIONAL_RACE
   ```

   Look at the code enabled by this change and explain how the race could cause qsort to fail to sort the array of integers.

3. Write a shell script using your favorite scripting language to obtain qsort's failure rate when sorting 10000 integers with 8 threads. Your script should run qsort at least 1000 times.

   Hint: The application will return 0 when the sort succeeds and 1 when the sort fails. If you write a bash script, you can obtain the return value using the $? variable.

4. Now run qsort through the Cilk race detector using the following command:

   ```
   cilkscreen -- ./qsort 10000
   ```

   Is CilkScreen able to detect the race?

Problem 2: Matrix-Transpose: Writing your own Cilk Program

In this problem, you will parallelize a simple matrix transpose application written in C++ by converting it to a Cilk program and using Cilk's `cilk_for` loop.

1. Build the matrix-transpose binary by running make. The binary's command line syntax is:

   ```
   ./matrix-transpose [n]
   ```

   - `n`: Number of rows and columns. Default = 2000

   Run matrix-transpose with n=1000, 3000, and 10000. What are the execution times?

2. Convert matrix-transpose into a Cilk++ application. You will have to first rename the matrix-transpose.cpp to matrix-transpose.cilk, and modify the Makefile to compile the new file with the Cilk++ compiler. Next modify matrix-transpose.cilk to include the Cilk header file and to replace `main` and `cilk_main`. By using `cilk_main` the command line argument `-cilk_set_worker_count [n]` is available to specify the number of workers (hardware threads used). Finally, add the "`-fcilk-stub`" compile time flag to the $CFLAG variable in the Makefile in the case that make is called with "`DEBUG=1`". This ensures that the debug build creates the sequential version of the code so that debugging is easier.

   Hint: You can look at the qsort code and Makefile for help.

3. Parallelize your code by converting the outer loop to a `cilk_for` loop in the `matrix_transpose` function.

4. Repeat the tests you performed with the C++ version while experimenting with the number of workers. What performance do you observe? What is the speedup of your parallel code when running with 2, 4, 8, and 16 threads?

5. Run the CilkScreen Race Detector as before. There should not be any races. What happens if you change the inner loop in `matrix_transpose()` to:

```
for (int j=0; j<=i; ++j) {
```

Problem 3: Mergesort: Parallelizing a Divide and Conquer Algorithm

1. Build the mergesort binary by running make. This will produce a sequential mergesort binary. The binary's command line syntax is:

   ```
   ./mergesort [ndata] -cilk_set_worker_count [n]
   ```

   - `ndata`: Number of data points. Default = 10,000,000
   - `n`: The number of workers. Default = # of hardware threads on your system

   While the application is written in Cilk, it is not yet parallelized. Run mergesort with n=1, 2, 4, 8 and 16. What are the execution times?

2. Parallelize the algorithm by inserting `cilk_spawn` and `cilk_sync` keywords into the `sample_mergesort()` function as appropriate. Repeat your previous experiments using 2, 4, 8, and 16 workers. Report the speedup over the sequential code that you observe.

Problem 4: Barnes-Hut algorithm

Barnes-Hut algorithm simulates the gravitational interactions between a set of masses. This algorithm may be generalized for other potential functions, but we will restrict ourselves to gravitational forces in 3 spatial dimensions.
First, a bit of background. The gravitational force exerted by a (point sized) body of mass m_1 located at position $\mathbf{r_1}$ on another (point sized) body of mass m_2 at location $\mathbf{r_2}$ is

$$F_{p_1}(p_2) = \frac{Gm_1m_2}{|\mathbf{r_1} - \mathbf{r_2}|^3}(\mathbf{r_1} - \mathbf{r_2}). \tag{1}$$

Note that $\mathbf{r_1}, \mathbf{r_2}$ are position vectors, and that $|\mathbf{x}|$ is the length of vector $\mathbf{x}$. G is the universal gravitational constant which can only be determined empirically. If there are n point size masses, the i-th body p_i of mass m_i being located at r_i, then the force on the j-th particle is

$$F(p_j) = \sum_{i=1, i\neq j}^{n} F_{p_i}(p_j) = \sum_{i=1, i\neq j}^{n} \frac{Gm_im_j}{|\mathbf{r_i} - \mathbf{r_j}|^3}(\mathbf{r_i} - \mathbf{r_j}), \tag{2}$$

which is just the sum of the forces exerted on body p_j by all other bodies.
If we are given a set of n bodies, we can compute the gravitational force experienced by each body in $O(n^2)$ time by computing the interaction between pairs of bodies and then adding them up appropriately. If we are willing to tolerate some inaccuracy, Barnes-Hut algorithm gives a faster way to do the computation in $O(n \log n)$ time.
Algorithm description: Given a set of n bodies, we find a axis-aligned cube C of minimum dimension containing all the bodies. We divide this cube into 8 smaller cubes by cutting in half along each dimension. We also create the tree `OctTree`, whose root is the big cube C. Each of the 8 smaller cubes is the child of the root. We continue to recursively divide up the cubes in to smaller cubes (by dividing a cube into 8 pieces by cutting in half along the 3 dimensions). Whenever we divide a cube X in to 8 subcubes, we grow the `OctTree` by creating nodes corresponding to each of the 8 subcubes as children to the node corresponding to the cube X. We stop dividing a cube if the the number of bodies located in it is lesser than a given constant `gMaxLeafSize`. The nodes corresponding to such cubes are the leaves of the tree. At each node N in the

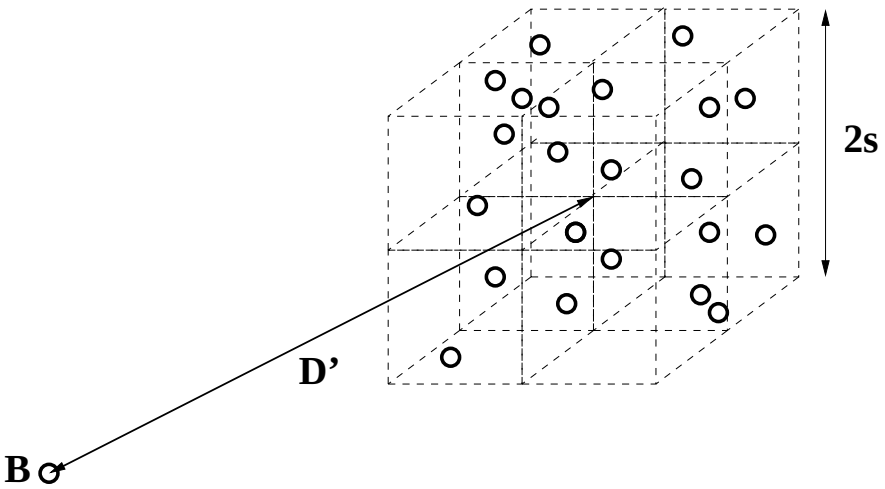

Figure 1: The cube is too close and too big to the body B because $2s/D' > \alpha$. We recursively compute the forces due to each of the eight subcubes and add them up.

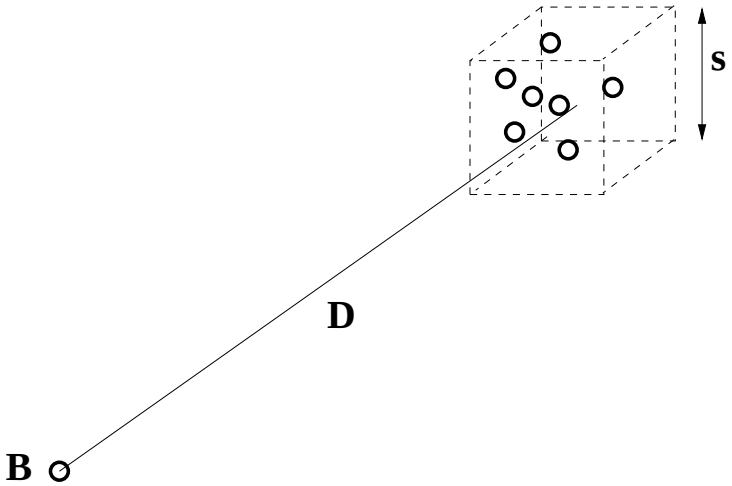

Figure 2: Here $s/D < \alpha$. Therefore, we approximate the force on body B due to all those in the cube by replacing the bodies in the cube by a single body (whose mass is the sum of masses of bodies in the cube) at the center of mass of the cube.

`OctTree`, we also store the combined mass of all the bodies in the corresponding cube (m_N), the center of the cube ($\mathbf{c_N}$), and the center of mass of the cube ($\mathbf{r_N} = \left(\sum_{i \in N} m_i \mathbf{r_i}\right) / \left(\sum_{i \in N} m_i\right)$).
To compute the force exerted on a body B by all other bodies, we start at the root of the `OctTree` and recursively go down the nodes in the tree computing the force exerted by the bodies in the nodes. If a certain node N in the tree is far enough and small enough from B according to the criteria $s/D < \alpha$ (s is the size of the corresponding cube, D is the distance of the center of cube from the body B, and α is a tolerance constant), then we approximate the force on B due to node N by

$$F_N(B) = \frac{G m_N m_B}{|\mathbf{r_B} - \mathbf{r_N}|^3}(\mathbf{r_N} - \mathbf{r_B}). \tag{3}$$

We call this force an *indirect interaction*. If the node N does not meet the criteria $s/D < \alpha$ and is not a leaf of the `OctTree`, we recursively compute the forces on B due to each of the 8 children nodes of N and add up the 8 forces. If the node N does not meet the criteria $s/D < \alpha$ and is a leaf of the `OctTree`, then we compute the forces on B due to each of the bodies in the leaf N and return the sum of the forces. We term these *direct interactions*.

You can find the C++ code for this algorithm in the directory you copied over in the subdirectory `barnes-hut`. All necessary constants are in the C++ file. The error tolerance parameter α is defined in the file as `gAlpha` (G is `gGrav`).

Your task is to restructure (or rewrite) this sequential code using Cilk commands so that it runs as fast as possible on `multi6`. The input to the program is the number of bodies the program should run on (not greater than 5 million). The program contains a function that generates random bodies given the number of bodies to be generated. Use this function without modifying it. Now that you have the bodies (data type `particle` in the code), you will first construct the `OctTree` and then compute the forces. You should time both these steps and print them out to `stdout`. It should also output the average number of direct and indirect interactions per body, the depth of the tree and the sum of forces on all the bodies. The output format of you program should be the same as the reference program you have been given. In addition to the code, also submit a separately written work and depth analysis of your code, assuming the points are distributed uniformly so that the `OctTree` is almost balanced (*ie*, the depth of the tree is $O(\log n)$).

Your program will be graded partially based on the execution times of both the steps of your code on the `multi6` machine. The teams with the best times on each step will additionally be given 10 percent extra credit (separately for each step).