Lab 10 - *CilkLab*

Due: *Fri, May 2nd, 2014 @ 11:59 pm*

Parallel and Sequential Data Structures and Algorithms 15-210 (Spring '14)

1 Introduction

In this lab, we move away from SML to give you a taste of what parallel programming in another language is like! We will use C++ and the C++ extension *Cilk Plus* exclusively in this lab.

You'll be solving a coding problem that we call *Friendly Stars*, and a purely written question (no coding!) on parallel sorting. Note that this lab is *very different* from all the other labs in this class, and we hope that you'll enjoy watching your parallel code *actually be* parallel! Nothing like seeing some real-life *speedup* to close out your 15–210 lab experience :-).

2 Files

After downloading the assignment tarball from Autolab, extract the files by running:

```
tar -xvf cilklab-handout.tgz
```

from a terminal window. Some of the files worth looking at are listed below. You should only modify the files denoted by *, as these will be the only ones handed in by the submission script.

1. `Makefile`
2. * `stars/cilkStars.cpp`
3. `stars/serialStars.cpp`
4. `stars/util.h`
5. `sorting/quickSort.cpp`
6. `sorting/sampleSort.cpp`

Additionally, you should create a file called:

```
written.pdf
```

which contains the answers to the written parts of the assignment.

3 Submission

To submit your assignment to Autolab, open a terminal, `cd` to the `cilklab` folder, and run:

```
make
```

Alternatively, run `make package`, open the Autolab webpage and submit the `handin.tgz` file via the "*Handin your work*" link.

4 Cilk Plus

In this lab, we will be using Intel's *Cilk Plus*, which is an extension to C++ that supports parallel computing. It provides simple language extensions that basically match the `par` construct and the parallel `map` constructs we have been using in the course. The `cilk_spawn` and `cilk_sync` constructs, described below, allow you to do fork-join parallelism in the same way as `par`. They create multiple tasks that run in parallel and synchronize when finished. `cilk_for` allows you to do parallel `for` loops, which are similar to a `map`.

Since Cilk Plus is embedded in an imperative language you can use it imperatively. However, the whole idea of Cilk is to avoid race conditions, and therefore effectively write "functional"-like code with respect to parallelism. Cilk has a race detector (*Cilk screen*) which dynamically identifies any races. Since this assignment is simple, you will not need it; but if you decide to do any significant programming in Cilk, we strongly suggest you use it (just Google it!). We are assuming that you have a working C knowledge from 15-122, as you will be writing some *basic* C/C++ code. If you have any questions about syntax and/or imperative constructs, please don't hesitate to ask your TAs![1]

To get started, there are a few Cilk Plus keywords you will need to know. Note that these are just keywords that *extend* C++ syntax; this is *not* a completely new language.

4.1 `cilk_spawn`

Specifies that a child function can execute asynchronously with its caller. That means it does not require the caller to wait for it to return.

The code following a `cilk_spawn` is referred to as a *continuation*. The continuation may be *stolen* by another worker to execute in parallel with the function that was spawned. Note that `cilk_spawn` is an expression of the *opportunity* for parallelism. The Cilk Plus runtime will decide whether or not to run the spawned function in parallel with its caller.

4.2 `cilk_sync`

This is a *barrier* that enforces all spawned calls in this function to complete before execution continues. This means that workers that complete early must wait there for the rest to catch up. There is always an implied `cilk_sync` at the end of every function that contains a `cilk_spawn` . It is important to note that `cilk_sync` only affects the child functions that are spawned from this function; functions spawned higher on the call tree will continue to run in parallel with the function executing `cilk_sync` .

Using `cilk_spawn` and `cilk_sync` , we can implement the SML function `Primitives.par` . Let's say we have two functions $f(x)$ and $g(x)$ that we want to run on inputs x and y respectively, in parallel. Recall that in SML this is:

```
val (a, b) = Primitives.par(fn () => f(x), fn () => g(y))
```

[1]The performance numbers given in class were based on programs written in Cilk. Professor Blelloch and his students have developed a large set of algorithms in the language, including algorithms for *sorting, BFS, MST, nearest neighbors, maximal independent set, max flow, hashing, suffix trees, minimum edit distance, and Delaunay triangulation.*

We can express the same thing in Cilk Plus using `cilk_spawn` and `cilk_sync` :

```
a = cilk_spawn f(x);
b = g(y);
cilk_sync;
```

4.3 `cilk_for`

Parallelizes a simple `for` loop. That is, it converts it into one where iterations of the loop body can be executed in parallel. It is important to note that the Cilk Plus runtime assumes that the developer is aware of the eligibility of parallelization in the program. That is, that this loop body can be parallelized without race conditions.

5 Getting Started

Since Cilk Plus is just an extension to C++, the GCC compiler has built-in support for it. However, the version installed on the `unix.andrew.cmu.edu` and the `ghc.andrew.cmu.edu` machines have not yet been updated to the latest version (v4.9), or the experimental branch of v4.8 that has support for Cilk Plus. To work around this, we have installed this experimental branch of GCC 4.8 in the 15-210 directory on AFS for your use. As a result, **you MUST use EXCLUSIVELY the Andrew or Gates machines to work on the lab—you cannot do this locally on your own computer.**

5.1 Environment setup

To temporarily use our version of GCC for this lab, you need to run one of the following two options **everytime** you login to AFS, **before** you start working. If you're using `bash`:

```
$ source /afs/cs/academic/class/15210-s14/cilk/gccvars_bash.sh
```

Or if you're using `csh`:

```
$ source /afs/cs/academic/class/15210-s14/cilk/gccvars_csh.sh
```

To check that this 'worked', you can run `which gcc`. You're good to go if it says something to the effect of `/afs/cs/academic/class/15210-s14/cilk/gcc-cilk-install/bin/gcc`. Otherwise, go back and make sure you typed the correct commands.

If you want this to be done automatically upon login, you can `source` the `gccvars_bash.sh` file in your `bashrc` (or `gccvars_csh.sh` in your `cshrc`). You can modify your `bashrc` (same for `cshrc`) by doing:

```
$ vim ~/.bashrc
    # Insert the exact source command from above
$ source ~/.bashrc
```

If you put this in your `*rc` file, **REMEMBER to remove it when you complete the lab!**. Though this directory won't disappear anytime soon, it is safest to just use the local `gcc` for your future classes.

If you encounter any problems setting up your environment, please contact your nearest neighborhood TA immediately, as you won't be able to make further progress in the lab!

5.2 Machines to use

As long as you have your environment properly set up, you should be able to run your code for this lab on any Andrew or Gates machine. Because we care about actually observing empirical speedup in this lab, the specs of the machine you run on will affect the performance you see. You may choose to just SSH into `unix.andrew.cmu.edu` and let the system pick the machine for you, or you can choose which you want to work on. It would be interesting for you to try machines with different *logical* core counts, to see what speed variation you get.

- The Andrew machines at `unix4-6.andrew.cmu.edu` have 2 physical 2.80GHz Intel Xeon E5-2680 CPUs with 10 cores each. With Hyper-threading turned on, this amounts to **40** logical cores.
- The Andrew machines at `unix1-3.andrew.cmu.edu` have 2 physical 2.40GHz Intel Xeon E5645 CPUs with 6 cores each. With Hyper-threading turned on, this amounts to **24** logical cores.
- The GHC3000 machines at `ghc27-50.ghc.andrew.cmu.edu` have 1 physical 3.20GHz Intel Xeon W3670 CPU with 6 cores. With Hyper-threading turned on, this amounts to **12** logical cores.
- The GHC5205 machines at `ghc51-76.ghc.andrew.cmu.edu` have 1 physical 2.67GHz Intel Xeon W3520 CPU with 4 cores. These machines have no Hyper-threading support, so there are just **4** logical cores.

5.3 Submission

As stated in Section 3, submission to Autolab happens in the same way as with the past labs. However, we will not be using an autograder for this lab, so you should test your code by yourself. We will use a script to run your code on our own machines after the due date to assess correctness and performance, and assign points accordingly.

6 Parallel Sorting

As experienced Computer Science students, you all know and love the various sorting algorithms. But how does sorting parallelize? You've seen in lecture where the parallel opportunities are in Quicksort and Mergesort, and even explored randomized Quicksort. In this problem, we take another look at Quicksort and then at a sorting variant that you may not have heard of before: Samplesort.

6.1 Logistics

The `sorting` subdirectory contains *already-written* implementations of Quicksort, Samplesort, and the shared sort timer. Read: **You do not have to write any code for this section!** Each sorting algorithm has a serial version and a parallel version implemented using Cilk Plus keywords. Running `make` in this directory (assuming you have the Cilk Plus environment set up) generates the executables `qsortTime` and `ssortTime`. Both of these binaries are executed by running:

```
$ make
$ ./qsortTime -t 8 -n 10000000
$ ./ssortTime -t 12 -n 100000000
```

`-t` lets you specify the number of cores that Cilk should use, and `-n` is the number of elements to be sorted. You are encouraged to play around with these numbers to see the difference they make!

For each execution of `qsortTime` or `ssortTime`, we alternate between the serial and the parallel versions for a fixed number of repetitions (to smoothen out minor inconsistencies and variations in timings). After each run of the parallel version, the relative speedup is computed and printed.

6.2 Quicksort

Task 6.1 (5%). Read and understand the `quickSort` function and its helpers in `quickSort.cpp`. What is its work and span? **Note:** You may assume that the partitioning step generates roughly equal-sized halves.

Task 6.2 (10%). Try running `qsortTime` with 100M elements on a range of thread counts, keeping in mind how many *logical* cores the machine you are working on has. Roughly what is the maximum speedup you can achieve? Are you able to observe linear speedup (i.e. a 20x speedup when run on 20 cores)? **Optional:** It would be interesting for you to collect a wide range of data points and plot these on a chart (number of cores vs. speedup) to observe the trend.

Task 6.3 (10%). Notice that in both `quickSort` and `quickSortSerial`, we switch to a different sorting function once the number of elements is under some threshold. Why do you think we do this? Note that this happens in `sampleSort` too.

6.3 Samplesort

The concept of Samplesort somewhat resembles that of Quicksort. However, instead of picking a single pivot, it picks a larger set of p pivots P (the value of p depends on the implementation). Next, the bucketing step buckets the input array A into p buckets, where element $A[i]$ is placed in bucket j if $P[j-1] \leq A[i] < P[j]$. Once bucketed, each process needs to be told which set of elements it is responsible for; there are many ways to implement this step, one of which is using offset indices (as is done in `quickSort.cpp`). Finally, once each process has ownership of its chunk of A, use a fast sequential sorting algorithm to locally sort each bucket. As a result, the distributed array is now sorted!

For those who are still confused about Samplesort, **we encourage you to ask on Piazza** and have fellow students clear up the confusion!

Task 6.4 (15%). Read and understand the `sampleSort` function and its helpers in `sampleSort.cpp`. The algorithm and implementation are split into 4 steps:

1. Choosing pivots to define buckets
2. Bucketing elements of the input array
3. Redistributing elements
4. Final local sort

For each step above, determine its work and span, then use this analysis to determine the work and span of the **entire** `sampleSort` (not `sampleSortSerial`) function. Also, how much *parallelism* is there? **Note:** You may assume that all buckets are about the same size.

Task 6.5 (10%). Consider running `ssortTime` with 100M elements on a *supercomputer* with 100,000 cores. Would this `sampleSort` implementation scale well? Why or why not?

7 Friendly Stars

You're out lying on Flagstaff Hill, stargazing on a beautifully warm and cozy Pittsburgh Spring evening (`http://bit.ly/1lz6XkW`) and you notice: *some stars are much closer together than others*. It is almost as if some pairs of stars were more *friendly* with each other than other pairs. So you get curious: you want to find out which pair of stars in the sky are the friendliest, and you want to do this in as little time as possible. Of course, being a theoretical Computer Scientist, you assume that there are a fixed, finite number of stars in the sky (or you just limit this to the range of your vision), that no two stars overlap each other, and that the position of a star is represented by a point in $\mathbb{R}^2$.

Define the *friendliness* of stars by the *Euclidean distance* between their respective positions. The *Friendly Stars* problem is simple: **given a set of star positions, which pair of stars is the friendliest?**

7.1 Logistics

We provide you a serial implementation `serial_friendly_stars` in `serialStars.cpp` that brute forces through every pair of points in the input. You should read and understand this implementation, then focus your efforts on `cilkStars.cpp`, where you will code your parallel version. The file `util.h` contains declarations of some functions that may be helpful to you. Don't worry about `CycleTimer.h`; we just use it to time your implementation.

7.2 Implementation

In this problem, you will code up a solution to the Friendly Stars problem using Cilk Plus, making sure that your implementation is *race-free*. The tester will test your code for correctness using the serial implementation, and then time your parallel version in order to compute what speedup you achieved.

Task 7.1 (10%). We could very trivially parallelize `serial_friendly_stars` by changing the two `for` loops to `cilk_for` loops. However, this implementation is incorrect. Why? If you want to, try implementing this in `cilk_friendly_stars`, then compile and run it. Note that you are not submitting any code for this task—just your written observations.

Task 7.2 (40%). In `cilkStars.cpp`, implement the function

```
ppair cilk_friendly_stars(point *inp, int N);
```

where `inp` is a size `N` array of input star positions. Your function should return the pair of positions of the friendliest stars. Note that all these types as well as several helpful functions are provided to you in `util.h`. **You should not modify any types or helper functions in the starter code**. Note that only `cilkStars.cpp` will be handed in, so any modifications you make outside this file will be ignored.

Your implementation should have $O(n^2)$ work and $O(n)$ span. We are aware that there are more efficient ways to solve this problem, but you are not required to implement them for this task.

Task 7.3 (Extra Credit). Beat our reference times by implementing an $O(n \log n)$ work and $O(n)$ span solution to this problem! We will award up to **20%** of extra credit here.

7.3 Testing

We will not be using Autolab's autograding feature for this lab. Instead, to test your `cilkStars` code, just run `make` in the `stars/` subdirectory (remember to run only on Andrew or Gates machines), then execute the compiled binary `cilkStars`. Please don't try to run `smlnj` or `CM.make` ;-).

```
$ cd cilklab/stars
$ make
$ ./cilkStars -n 100000
```

Here, `-n` specifies the number of input points, and there is also a `-t` that lets you limit the number of threads used. The tester will generate a large number of random test cases, then run the serial implementation followed by your parallel implementation, then compare the two for correctness and performance.

7.4 Grading

Your implementation of `cilk_friendly_stars` will be graded based on the speedup that it achieves. We will run your code ourselves after the due date and award full points to implementations that are within some fixed percentage of our reference time, and anything slower will receive points with linear falloff. This percentage and reference time will be posted on Piazza shortly. Note that because different machines have different specs and varying inconsistencies, we will finalize these details nearer to the due date.

Have fun watching your parallel code achieve speedup!

***So long**, and thanks for all the fish!*