Recitation 3: Parallel Programming

This recitation demonstrates how to use Cilk Plus to parallelize software for multicore processors. It also demonstrates how to use the Cilkscreen race detector and the Cilkview scalability analyzer.

1 Getting started

We recommend that you work on course machines. Since we will use gnuplot, you should enable X11 forwarding by using `-x` option:

```
$ ssh -x <username>@cloud.csail.mit.edu
```

To get a local copy of the repository for your work, you need to use Git to clone it:

```
$ git clone /afs/csail.mit.edu/proj/courses/6.172/student-repos/fa14/recitations/
> recitation3/<username>.git recitation3
```

You can also get the recitation code from GitHub:

```
$ git clone https://github.com/mit6172/recitation3 recitation3
```

2 Cilk Plus

2.1 The cilk_spawn and cilk_sync keywords

Compile the `fib` program in the `fib/` subdirectory. Then, time the execution for finding `fib(45)` using the `time` command:

```
$ time ./fib 45
```

You will get 3 different times as outputs, labeled `real`, `user`, and `sys`. The `real` time is wall time, which is what you see from a clock. The `user` time is the time a CPU spent in user mode. The `sys` time is the time a CPU spent in the kernel. The sum of `user` and `sys` is the actual CPU time of the command. Since the current `fib` is a serial program, you will find that wall time is slightly higher than CPU time.

Next, we want to parallelize the program to take advantage of the other 11 processors on the cloud machines. You can do this by adding `cilk_spawn` in front of function calls that you want to execute in parallel. You also need to add `cilk_sync` to wait for all spawning tasks to finish. Lastly, you should include Cilk header using

```
#include <cilk/cilk.h>
```

Before you run your Cilk Plus program, type

```
$ export CILK_NWORKERS=12
```
on the command line, to let the machine know that it will use 12 cores to run the program.

- **Exercise:** Parallelize `fib` using `cilk_spawn` and `cilk_sync`, and time the output binary.

  You might find that the new version is not faster than the first one (in terms of *real* time). Furthermore, the CPU time is much higher than wall time, because the program uses multiple processors to run. Under what circumstances would the parallel version to be slower? Try to fix your program using “coarsening” to get a parallel speedup.

### 2.2 The `cilk_for` keyword

The `transpose/` subdirectory contains the source code form `transpose`, an in-place matrix-transpose program. As you saw in lecture, you can replace the `for` loops with `cilk_for` loops to parallelize `transpose`.

- **Exercise:** Parallelize `transpose` using `cilk_for`, and time the output binary with input size 10000.

### 3 The Cilkscreen race detector

Compile and run the `qsort-race` program in the `qsort-race/` subdirectory. This code is parallelized by simply adding `cilk_spawn` and `cilk_sync` to a serial quicksort program. As such, there is a race in the parallelization. We can use Cilkscreen to detect the race as follows:

```
$ cilkscreen ./qsort-race 1000 1
```

- **Exercise:** Use Cilkscreen to find this race. Then, fix the race, and use Cilkscreen to confirm that no more races exist.

### 4 The Cilkview scalability analyzer

The `qsort/` subdirectory contains `qsort.64`, another parallel quicksort program. You can use Cilkview to analyze the scalability of this program.

- **Exercise:** Run the following command to get a graph of the speedup of `qsort.64` on different numbers of workers:

```
$ cilkview --trials=all --plot=gnuplot ./qsort.64
```

Cilkview’s command-line output includes work and span measurements for the Cilk program in terms of instruction counts, as well as parallelism and “burdened parallelism” measurements based on the measured work and span. With the `--plot=gnuplot` option, Cilkview produces two files that `gnuplot` can evaluate to produce the speedup graph. You can modify these files and run them through `gnuplot` again to produce images to include in your homework and project write-ups.