Homework 1: OpenMP

Due: 5pm on Friday, April 24, 2020

- Info on the HPC cluster: `http://hpc.oit.uci.edu`
- Info on OpenMP: `https://computing.llnl.gov/tutorials/openMP`

In this assignment, you will implement a multithreaded version of the Mergesort and Quicksort algorithms using the OpenMP programming model. You will use the UCI HPC cluster. You should also use this assignment to familiarize yourself with the tools you will be using throughout the course.

Getting the scaffolding code on the HPC cluster

We will use the `git` distributed version control system. The baseline code for this assignment is available at this URL: `https://github.com/UCI-HPC/EECS224-hw1`

To get a local copy of the repository for your work, you need to use git to clone it. So, let's make a copy on the HPC cluster and modify it there. To do that, run the following command on the cluster.

```
$ git clone https://github.com/UCI-HPC/EECS224-hw1.git
```

If it works, you will see some output similar to the following:

```
Cloning into 'EECS224-hw1'...
remote: Enumerating objects: 11, done.
remote: Counting objects: 100% (11/11), done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 11 (delta 0), reused 11 (delta 0), pack-reused 0
Unpacking objects: 100% (11/11), done.
```

There will be a new directory called **EECS224-hw1**.

Compiling and running your code

We have provided a small program, broken up into modules (separate C/C++ files and headers), that performs sorting sequentially. For this homework, you will make all of your changes to just two files as directed below. We have also provided a `Makefile` for compiling your program. To use it, just run `make`. It will direct you with the right flags to produce the executable. For example, `make quicksort` will produce an executable program called `quicksort` along with an output that looks something like the following:

```
$ make quicksort
icpc  -O3 -g -o driver.o -c driver.cc
icpc  -O3 -g -o sort.o -c sort.cc
icpc  -O3 -g -o parallel-qsort.o -c parallel-qsort.cc
icpc -O3 -g -o quicksort driver.o sort.o parallel-qsort.o
```

Run `quicksort` on an array of size `1000` as follows:

```
$ ./quicksort 1000
Timer: gettimeofday
Timer resolution: ~ 1 us (?)

N == 1000

Quicksort: 0.000182 seconds ==> 5.49451 million keys per second
        (Array is sorted.)
My sort: 0.000119 seconds ==> 8.40336 million keys per second
        (Array is sorted.)
        (Arrays are equal.)
```

Running jobs on the cluster

The HPC cluster is a shared computer. When you login to `hpc.oit.uci.edu`, you were using the login node. You should limit your use of the login node to light tasks, such as file editing, compiling, and small test runs of, say, just a few seconds. When you are ready to do a timing or *performance* run, then you submit a job request to a grid engine (GE) scheduler. To submit a job request, there are two steps: Create a batch job script, which tells GE what machine resources you want and how to run your program. Submit this job script using a command called `qsub`. A batch job script is a shell script file containing two parts: (i) the commands needed to run your program; and (ii) metadata describing your job's resource needs, which appear in the script as comments at the top of the script. We have provided a sample job script, `sort.sh`, for running the Quicksort program you just compiled on a relatively large input of size 10 million elements.

Go ahead and try this by typing the following commands:

```
$ qsub sort.sh
$ qstat -u <UCInetID>
```

The first command submits the job. It should also print the ID of your job, which you need if you want to, say, cancel the job later on. The second command, `qstat`, lists the contents of the central queue, so you can monitor the status of your job request.

For other useful notes and queue commands, such as `qdel` for canceling a job, see the HPC cluster documentation on running jobs at `http://hpc.oit.uci.edu/running-jobs`.

When your job eventually runs, its output to standard output or standard error (e.g., as produced print statements) will go into output files (with .o### and .e### files, labeled by the job ID ###). Go ahead and inspect these outputs, and compare them to the commands in qsort.sh to make sure you understand how job submission works.

C/C++ style guidelines

Code that adheres to a consistent style is easier to read and debug. Google provides a style guide for C++ which you may find useful: `http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml`.

1 MergeSort

Although we've given you a lot of code, to create a parallel Mergesort you just need to focus on editing `parallel-mergesort.cc`. Right now, it is mostly empty. In this file, first implement a sequential divide-and-conquer Mergesort algorithm. Record its performance and make sure that its not slower than the reference Quicksort implementation used for validation.

Next, there are two parts you need to parallelize:

1. the two recursive calls; and
2. the merge step.

First, leave the merge step sequential and parallelize the recursive calls. Then, parallelize the merge step you implemented. To get full credit, your implementation needs to beat the easy parallelized algorithm in which the merge step remains sequential.

2 Quicksort

The repo we've provided includes a file called `parallel-qsort.cc`. Right now, it just contains a "textbook" sequential version. Open the code and browse it, to make sure you at least understand the interfaces.

There are two parts you need to ultimately parallelize:

1. the partition step; and
2. the two recursive calls.

First, leave the partition step sequential and parallelize the recursive calls. Then, parallelize the partition step, which the `partition()` function implements. Note that unlike parallelizing the recursive calls, parallelizing the partition step will not be a simple matter of inserting OpenMP directives. You will need to come up with a different approach.

In addition to submitting your code, briefly describe how your Quicksort parallel algorithm works and whether it requires auxiliary storage (and if so, how much). Analyze the work and depth of your parallel partitioning algorithm.

Submission

When youve written up answers to all of the above questions, turn in your write-up and tarball/zip of your code by uploading it to Canvas.