Homework 1: Work-Depth model; OpenMP

Due: 11:59 P.M. on Sunday, February 5, 2017

- 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 algorithm 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.

This homework's reading: Chapter 27 of "Introduction to Algorithms" by CLRS (file on Canvas) and Chapter 7 from Intro to Parallel Computing (textbook).

You may work in **teams of two**. Submit one copy of the homework per team. Be sure to indicate your assignment partner by creating a `README` file as part of your submission.

Part 0: Getting Started

The login node of the HPC cluster is: hpc.oit.uci.edu. Use a ssh client and your UCInetID login/password to log into the cluster.

```
$ ssh <UCInetID>@hpc.oit.uci.edu
```

Documentation regarding the cluster is available online at `http://hpc.oit.uci.edu`.

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/EECS117/hw1.git`

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/EECS117/hw1.git
```

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

```
Cloning into 'hw1'...
remote: Counting objects: 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.
Checking connectivity... done.
```

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

Git basics

Create a README file and enter some of your details: name, partner's name, and email. When you've done so, you can add and commit your change to your repository.

```
# Add the changed file
$ git add README

# Commit -- saves file with a message ("-m" option)
$ git commit -m "Added README"
```

You can learn more about git at http://git-scm.com/book/en/v1/Git-Basics.

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 lab, you will make all of your changes to just one file 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 mergesort-omp will produce an executable program called mergesort-omp along with an output that looks something like the following:

```
$ make mergesort-omp
g++  -O3 -g -o driver.o -c driver.cc
g++  -O3 -g -o sort.o -c sort.cc
g++  -O3 -g -o parallel-mergesort.o -c parallel-mergesort.cc
g++ -O3 -g -o mergesort-omp driver.o sort.o parallel-mergesort.o
```

Run mergesort-omp on an array of size 100 as follows:

```
$ ./mergesort-omp 100
```

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, mergesort.sh, for running the Mergesort 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 mergesort.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 mergesort.sh to make sure you understand how job submission works.

Using a debugger

GDB is a debugging tool that allows you to view your program's state as it is executing. For example, it can help you debug when you get a segmentation fault.

To debug your program using `gdb`, you should first build a "debug" version of your program. Modify your Makefile to add the compiler flags `-g` (adds debug symbols) and `-O0` (disables optimizations) as follows:

```
COPTFLAGS := -g -O0
```

Now recompile your program. You can now start a debugging session in `gdb` as follows:

```
$ gdb  mergesort-omp
```

This command should give you a `gdb` prompt, where you can type the command `run <args>` or `r <args>`. If your program crashes, giving you back a prompt, you can type the command `backtrace` or `bt` to get a stack trace to further inspect what caused the error in your program.

To learn more about `gdb`, type `help gdb` on the command line.

Using a memory checker

Some memory bugs do not crash the program, so `gdb` cannot tell you where the bug is. You can use the memory checking tool `valgrind` to track these bugs:

```
$ valgrind ./mergesort-omp 100
```

The C/C++ programming language requires you to free memory after you are done using it, or else you will have a memory leak. Valgrind can track memory leaks in the program. When you run valgrind, you will see a summary of the memory leaks in your program at the end. To get more information, you can build your program in debug mode and run valgrind, again using `--leak-check=full`.

Verify that valgrind doesn't complain about any errors or memory leaks before committing your code. The staff code does not have any memory leaks.

Profiling using perf

The Linux Perf Events subsystem uses a sampling approach to gather data about important hardware and kernel events, such as cache misses, branch misses, page faults, and context switches.

The `perf` program, distributed in the linux-tools package, records data during profiling runs and displays the results in the terminal. We're only going to use the `stat` subcommand. It produces an output that looks something like the following if you run it on `make`:

```
$ perf stat make

 Performance counter stats for 'make':

         2.606767 task-clock                #    0.817 CPUs utilized
                2 context-switches          #    0.767 K/sec
                0 cpu-migrations            #    0.000 K/sec
              258 page-faults               #    0.099 M/sec
        7,897,851 cycles                    #    3.030 GHz                     [61.52%]
        1,855,579 stalled-cycles-frontend   #   23.49% frontend cycles idle    [67.89%]
        1,705,432 stalled-cycles-backend    #   21.59% backend  cycles idle    [84.81%]
        6,335,917 instructions              #    0.80  insns per cycle
                                            #    0.29  stalled cycles per insn [87.35%]
        1,394,321 branches                  #  534.885 M/sec                   [87.35%]
           52,969 branch-misses             #    3.80% of all branches         [87.35%]

      0.003188710 seconds time elapsed
```

You can choose specific events, such as `L1-dcache-load-misses`, with the `-e` option.

```
$ perf stat -e L1-dcache-load-misses make

 Performance counter stats for 'make':

           66,394 L1-dcache-load-misses

      0.003038134 seconds time elapsed
```

You can see a full list of events by running `perf list`. For more examples on using `perf`, refer to: https://perf.wiki.kernel.org/index.php/Perf_examples.

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.

Part of your grade on assignments is based on the readability of your code.

Part 1: Parallel 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, implement the parallel mergesort algorithm described in class. To get full credit, your implementation needs to beat the "easy" parallelized algorithm in which the merge step remains sequential.

After you change your code, don't forget to recompile (by running `make`) and submit a batch job to collect timing data. If you did it correctly, the '.e*' file will not show any abnormal termination errors and you will observe better performance than the sequential version. Once you are satisfied, add and commit your implementation. Committing often is a good practice. It also helps revert to an earlier implementation if it happens to be faster than your current solution.

Part 2: Matrix transpose

(a) Consider the transpose algorithm shown in Exercise 27.1-7 of the CLRS reading (page 792 or PDF page 23). Argue that this transpose algorithm is correct.

(b) Now do exercise 27.1-7, which is to analyze the work, span, and average available parallelism of this algorithm.

(c) Do exercise 27.1-8 of the CLRS reading as well (same page).

OPTIONAL EXTRA CREDIT: Parallel Quicksort

To create a parallel Quicksort, you just need to focus on editing `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.

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.

Once you've created an implementation you are satisfied with, use the same add / commit procedure as above. In addition to submitting your code, briefly describe how your 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 you've written up answers to all of the above questions, turn in your write-up and tarball of your code by uploading it to eee.uci.edu.