Lab 1

Concurrent Programming

Due: 11:59pm, September 25, 2019

This lab should familiarize you with fork/join parallelism, barriers, and locks.

Summary: Write, in C or C++, two sorting algorithms. The first algorithm should use fork/join parallelism to parallelize either Quicksort or Mergesort (you may use your code from Lab 0). The second algorithm should use locks to implement BucketSort across multiple sorted data structures (e.g. STL's map). As with Lab 0, you are emulating the performance of the UNIX `sort -n` command. This lab is an individual assignment. Unlike Lab 0, your code for Lab 1 should time itself. Bonus points will be awarded for the fastest implementations of Quick/MergeSort and BucketSort.

Code Requirements / Restrictions:
Fork/Join Quick/Mergesort: Your fork/join algorithm should be an implementation of either Mergesort or Quicksort. Your parallelization strategy should use fork/join parallelism, implemented using fork, join, and barriers. Locks should not be necessary. You are allowed to use pre-written unsorted data structures (e.g. STL's vector), and your Lab 0 code, but you may not use any pre-written sorted data structure or sorting algorithm (e.g. STL's map) for this algorithm.
Locking BucketSort: Your second algorithm should be an implementation of BucketSort using sorted data structures (e.g. STL's map - note that sorted data structures are allowed in this algorithm, but not the other). BucketSort normally uses unsorted buckets, using sorted buckets makes the task significantly easier - you need not recurse to sort each bucket after the first insertion (see Figure 1).
Your implementation of Locking BucketSort should avoid idling threads if at all possible. For instance, a problem decomposition that assigns the bucket ranges evenly across threads will result in idle threads when the source distribution is highly skewed e.g. (1,2,3,4,5,6,7,1001,1000,1002). Instead, you should work to ensure that all threads are busy at all times, even at the cost of synchronizing across buckets.

```
function bucketSort(array, k) :
  buckets : new array of k empty sorted lists
  M : the maximum key value in the array
  for i = 1 to length(array) do
    insert array[i] into sorted buckets[floor(k * array[i] / M)]
  return the concatenation of buckets[1], ...., buckets[k]
```

Figure 1: BucketSort using sorted buckets

Lab write-up: Your lab write-up should include:

- A description of your two parallelization strategies
- A brief description of your code organization
- A description of every file submitted
- Compilation instructions
- Execution instructions
- Any extant bugs

I expect your lab write-up for this project will be around a page or two.

Code style: Your code should be readable and commented so that the grader can understand what's going on.

Submission: You will submit a zip file of your lab to canvas. When unpacked, the directory should contain all files required to build and run your program, along with a brief write-up. Pay particular attention to the requirements for compilation and execution, as some testing will be done using automatic scripts.

Compilation and Execution: Your submitted zip file should contain a Makefile and the project should build using a single `make` command. The generated executable should be called `mysort`. The `mysort` command should have the following syntax (pay attention, it is slightly different from Lab 0):

```
mysort [--name] [source.txt] [-o out.txt] [-t NUM_THREADS] [--alg=<fj,bucket>]
```

Using the `--name` option should print your name. Otherwise, the program should sort the source file. The source file is a text file with a single integer on each line. The `mysort` command should then sort all integers in the source file and print them sorted one integer per line to an output file (specified by the `-o` option). The time taken to sort the file (excluding the time for file I/O and to initially launch / finally join threads) should be printed to standard out in nanoseconds (see `text.c` for examples). The `-t` option specifies the number of threads that should be used, including the master thread, to accomplish the sorting. See Figure 2 for `mysort` syntax examples. The `getopt` and `getopt_long` method calls are helpful for parsing the command line.
See the included `test.c` and `Makefile` for boilerplate thread launching and timing code using the high resolution `CLOCK_MONOTONIC` timer. You are welcome to use any/all of this code for building your applications. Note that `test.c` does not appropriately handle the `mysort` command line syntax.

Testing: Testing can be done using the same methodology as Lab 0 (e.g. `shuf`, `sort`, and `cmp`).
Test machines will be available for you to execute code on. These machines have multiple hyperthreaded cores in a single socket. In order to connect to these machines, you'll need to be logged into the University network (UCB Wireless or a vpn). Your user name will be your last name, all lower case. Your password will be the four digit time you submitted Lab 0. If you didn't submit Lab 0, you should come talk to me. Test machines will be brought up slowly this week, at this point, the only one available is at 128.138.189.219 (see Figure 3 for how to log in and copy code over). Note that other students running on the machine may interfere with your timing experiments, so try to use one that is unoccupied.

Grading: Your assignment will be graded as follows:

Unit tests (80%) We will check your code using sixteen randomly generated input files, eight for each algorithm. Correctly sorting a file is worth five points.

Lab write-up and code readability (20%) Lab write-ups and readable code that meet the requirements will get full marks. Incomplete write-ups or unreadable code will be docked points.

Bonus Points (up to 10%) The fastest Quick/Mergesort algorithm on the largest unit test will receive five extra points. The second fastest will receive four, the third fastest three, etc. The same scoring applies to BucketSort, for a total of up to ten extra points.

Recall that late submissions will be penalized 10% per day late, and will only be accepted for three days after the due date. Canvas submissions include the submission time.

```
### print your name
./mysort --name
# prints:
Your Full Name

### Consider an unsorted file
printf "3\n2\n1\n" > 321.txt
cat 321.txt
# prints
3
2
1

### Sort the text file and print to file
./mysort 321.txt -o out.txt -t5 --alg=fj
# prints time taken in seconds for 5 threads on fork/join sort:
294759

cat out.txt
# prints:
1
2
3
```

Figure 2: Examples of your `mysort` program's syntax

```
### to copy the current directory to
#   your home directory on the test machine
scp ./* lastname@128.138.189.219:~
# it'll ask for your password and
# prints something like:
lastname@128.138.189.219's password:
Makefile  100%   52    16.4KB/s   00:00
test.c    100% 1685   185.1KB/s   00:00

### to *recursively* copy the current directory to
#   your home directory on the test machine
scp -r ./* lastname@128.138.189.219:~

### to copy your remote home directory to
#   your local machine's current directory
scp lastname@128.138.189.219:~ .

### copy to the test machine only changed files
#   from the current directory to your remote home directory
rsync -avzrh . lastname@128.138.189.219:~

### to login to test machine
ssh lastname@128.138.189.219
# now you are on the test machine!

# since we already copied over code, we can:
make
# prints:
gcc test.c -pthread -O3 -g -o test
# and also execute
./test
# prints:
creating thread 2
creating thread 3
creating thread 4
creating thread 5
Thread 1 reporting for duty
Thread 2 reporting for duty
Thread 3 reporting for duty
Thread 5 reporting for duty
Thread 4 reporting for duty
joined thread 2
joined thread 3
joined thread 4
joined thread 5
Elapsed (ns): 260993
Elapsed (s): 0.000261

# To see who else is running things:
top
```

Figure 3: Using the test machines