Lab 1

Parallel Quick/Bucket Sort Program

Derek Prince

Code Organization

The code is in a single file save for the argument parser in the header. main() starts by setting up the argument parser and then parsing out the
the options. Options are parsed as if-checking statments with `--help` and `--name` having first and second priority as they return with 0 after running. If neither of those options are selected then it pulls out the othre relevant input paramerers.

After reading in the input file, the program either runs the parallel quicksort code or the bucketsort depending on which argument (`fj | bucket`) was passed in at launch.
Since making threads is expensive, there is first a check to see if it would not make more sense to simply run a lighter, sequential version of the algorithm before diving into setting up a semi-thread pool.

Quicksort

The quicksort algorithm starts by calling out the parallel quicksort routine in a main thread. Once in the quicksort function, the main thread continues splitting the arrays into a partial order,
only creating threads for the higher (left)end of the array and

continuing the work on the lower (right) itself as it would be idle waiting on join() otherwise.

It keeps track of how many threads are left by the thread ID passed to it in the void *args parameter. If the thread id reaches the `NUM_THREADS` global then it will finish sequentially.
I had considdered trying to track thread joins in other branches in hope to take the resources back in the event of running out of threads but it would have likely required atomic instructions.

Bucketsort

This algorithm follows a more directly parallel scheme by splitting off all the threads at once in a `for{...}` loop and joining with the master again after finishing their tasks.
The main “trick” here is that the threads are divided across the buckets based on a target element-per-thread `#define` in an attempt both keep enough threads to make it as parallel as efficient
while avoiding having so many threads that each of them barely does any work. At the moment the target is 10 elements per thread but I’m sure that could be improved with further testing (likely a higher number).

The bucketsort algorithm is also spread into a main thread caller (that also finishes the remaining sorts that dont divide evenly into threads) and an insert task spun out of the main thread to the worker threads.

Compilation instructions

Unzip file and cd into top level with the Makefile. The file structure should look as such:

```
 --- Lab1/
| ------ Readme.md
| ------ Writeup.pdf
| ------ Makefile
| ------ Source/
|       |------ mysort.cpp
|       |------ Include/
|       |       |------ cxxopts.hpp
```

To build simply type `make` and it will compile with g++ and C++11 standards as well as -Wall -Werror.

`make clean` will remove object files while `make uninstall` will remove the executable.

Execution Instructions

The program operates as the lab directed with a few extra options that should be mostly transparent.
The main difference being the help menu accessed by running `mysort --help`.

To specify number of threads to run on (default 5), there must be a space between `-t` and the number.

To have `mysort` print author name (me) run `mysort --name`.

Ex: to run mysort with fork/join qsort on 28 threads and output to ex.txt `mysort input.txt -o ex.txt --alg=fj -t 28`.

Output will by default be written to a `output.txt` file unless specified.

Terminal Output

The program will only print the time taken in this version. I believe the terminal outfile text print will still work if the code is uncommented (because of `stl::iterators`) but I have not tested it.

Expected Input

A text file with one number per line up to the size of the range of an int.

Ex:

```
45
1
9
11
3456
6
9
0
```

Letters as fas as has been tested are ignored and removed from the

input. See extant bugs section for more.

File Descriptions

- Makefile - makefile to automate build with gcc.
- Readme.md - Markdown version of this PDF
- source/mysort.cpp - Entry point of program containing all relevant code.
- source/include/cxxopts.hpp - CLI argument parser library

Credits

cxxopts for command line parsing because nobody but C++ needs to solve this issue again.
Clang Format for cleaning my erratic coding styling.

Extant Bugs

None *known* bugs but unsupported file formats are largely untested. Letters and leading 0's are have proven to be ignored for all basic testing cases.

A file of only 0s will crash on overflow as it does an infinite recursion downwards. I don't realistically see this being a problem so I didnt limit it.