Lab 3

Derek Prince

Parallelization Strategies

As we are using OpenMP in this lab, I chose to focus on breaking the quicksort algorithm into independent tasks for OpenMP to schedule and execute as it deems fit. For the quicksort algorithm this was very simple as it immediately divides the task into sub arrays that will never overlap. This made for an idea task candidate and was what I went with in the end. I chose not to do each left and right subarray as a parallel region as my implementation gave no promises to creating a balanced load (though it makes an approiximated attempt at choosing a fair pivot index). Because of this, each branch must be allowed to progress without waiting for the other. The \`omp parallel\` is defined in \`main()\` where quicksort is called and the only has one section, called with \`nowait\`, to spawn the quicksort tasks. As opposed to Lab 1 where I let the master thread handle the left-most branch of the quicksort, I chose to let OpenMP do the task scheduling as it sees fit here and told it to schedule the left-most task as well. As for contention and data sharing, I left the sorting vector as the only shared data and copied (via \`firstprivate\`) the array bounds into each task. The partition index (pi) could have been left as shared but I did not want to have to worry about contention on the data between branches. As the partition index was only ever read I imagine the difference is negligle but this way I do not have to worry about it if I change something.

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 file names and attempts to open and read in data from the input file. If successful, it then tells OpenMP there is a parallel section with `my_quicksort()` as the only region. In `my_quicksort()` the two branches are each assigned and OpenMP task and follows the standard quicksort recursion thereafter.

The code is almost entirely sequentially read with only the quicksort algorithm being broken out at the top of the file with it's associated partition function.

Compilation instructions

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

```
 --- Lab3/
| ------ Readme.md
| ------ Writeup.pdf
| ------ Makefile
| ------ Source/
|       |------ mysort.cpp
|       |------ Include/
|       |       |------ cxxopts.hpp
```

To build simply run `make` while in the Lab3 directory.

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

`--help` output:

```
$ ./mysort --help
Text parse and sort with OpenMP.
Usage:
  ./mysort [OPTION...] positional parameters

  -n, --name         My name (for grading purposes)
  -o, --output FILE  Output file name
  -h, --help         Display help options
```

Example usage to sort an input text file `input.txt` and creating a sorted output file `output.txt`: `$ ./mysort input.txt -o output.txt`

Terminal Output

During normal sorting operation the program prints each number to the command line once on their own lines.

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
345
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

- **DerekPrince_writeup.pdf**
  - You are here.
- **Readme.md**
  - Markdown version of this PDF
- **Makefile**
  - makefile to automate build with gcc.
- **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.
- stackedit for a cleaner render.

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 didn't limit it.