Lab 2

Derek Prince

Brief Description

In this lab I extended Lab 1's bucket sort algorithm to use one of a collection of possible locks – specifically TAS, TTAS, Ticket, MCS, Pthread.h, and my own with atomic_flag that I wanted to test out.
In addition to this, a sepparate counter micro-benchmark was made to test out the locks as well as two barriers, Pthread.h and a sense reversal.

Notes

Folling unix convention, there are always spaces expected between short token (i.e. `-t` vs `--threads`) and the input. Similarly, long-form tokens always expect an `=` between them and the input (ex. `--alg=bucket`). I only mention it because it doesn not *exactly* match some of the examples in the writeup but is more consistent.

All programs can be run with `--help` to show what is expected.

I was intriqued by `std::atomic_flag` so I created a simple TAS lock around it as an extra locking option. That is the sole reason for it's existence.

Algorithms

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.

Algorithm Performance

counter

As barriers were not used in mysort, extensive perf analysis was not done on `mysort` in favor of `counter` benchmark to more directly correlate the data. (However, TTAS and Atomic_Flag locks performed the best on a list of 1000 random numbers if only barely)

Data was gathered with GNU perf and averaged over 10 runs.

- Locks

  ```
  perf stat -e L1-dcache-store,L1-dcache-load,L1-dcache-prefetches,branch-instructions,branch-misses,branch-loads,branch-load-misses,page-faults ./Lab2 -o output -t 50 --iter=1000 --lock=<lock type>
  ```

Lock	TAS	TTAS	Ticket	MCS	Pthread	AFlag
Run Time (ns)	0.0011690	0.0011206	0.0011086	0.0010869	0.0010391	0.0011791
L1 Cache (Hit %)	90.205	89.955	73.41375	65.6925	95.2225	87.72375
Branch (Miss %)	1.525	1.4725	1.44125	1.495	1.41375	1.495
Page Fault	249.375	250	249	249.25	248.625	248.625

- Barriers

  ```
  perf stat -e L1-dcache-store,L1-dcache-load,L1-dcache-prefetches,branch-
  ```

```
instructions,branch-misses,branch-loads,branch-load-misses ./Lab2 test.txt -o output -t 10 iter=1000 --bar=<barrier type>
```

Barrier	Sense	Pthread
Run Time (ns)	0.0008271	0.0076188
L1 Cache (Hit %)	77.89	40.6625
Branch (Miss %)	1.075	2.41375
Page Fault	168.25	169.25

Analysis

For the locks, the fastest lock (as in the sorting test) was `std::atomic_flag` followed by Pthread mutex lock. This is likely because of the overhead of acquiring the other locks. In particular, the unfairness of the TTAS lock is clearly evident as the counter has a very clear access pattern that revolves through the thread ids as the iterator increases. The lock that should be acquired is the same one that will initially bounce from load() instead of flag.tas(). The L1 cache hit for the pthread mutex lock is extraordinarily high which makes me wonder if the structures are cache-line padded. The LLVM linter was warning me that my structures would be padded to match cache line size but I compiled with GCC so I am unsure if that actually happened implicitly. Otherwise, the branch miss % and page faults are all roughly equal, deviating only below in the barrier implementations.

In the case of the barriers, the sense reversal barrier absolutely flogs the pthread implementation by almost 10x. I did not use barriers in my bucket sort algorithm so I cannot test it in that case but it seems like this barrier excels with the counter for the same reason the TTAS lock failed (ordered, rolling lock accesses). One thing, however that is very surprising is how many branch misses and L1 cache misses the pthread barrier has. Neither barrier has a particularly long-winded set of methods but the sense reversal barrier has significantly more elements involved in the wait() function so it surprises me that they are more often in the cache. There is a solid chance the difference comes down to the use of atomics in my sense barrier versus what seems to be mutexes in pthreads. An interesting comparison would be a simple barrier based on atomic flags and then implementing the sense signals with `std::atomic_flag` s as well.

Code Organization

In this lab I broke out the locks and barriers into classes with their respective hpp/cpp files to make them more portable and maintainable as they are used interchangably between the `mysort` and `counter` programs. The main thing in these files is a virtual interface that all locks or barriers are forced to implement. The reason for this is so that I can create a static lockBox (or barrierBox) object, initialize it in main() and call `lockBox->lock()` (or `barrierBox->wait()`)in any

thread without having to know which the user specified from the command line.

Otherwise the programs follow a pretty ugly but linear switching pattern through main() to first parse the command line, switch on those options, initialize relevant data and determine how best to spin up the threads to sort or simply launch them for count. Count was deliberately left without the thread overhead optimizations to allow testing the effects of thread overhead and such.

There is also a single cpu_relax() function in cpu_relax.hpp that is used in the sense reversal barrier to ease the spinning while waiting that I adapted from ifknot (see credits).

File Descriptions

- **Readme.md**
  - The markdown version of this writeup (I use these in jekyll with prettier rendering is why .md)
- **DerekPrince_writeup.pdf**
  - You are here.
- **Makefile**
  - makefile to automate build with gcc.
- **source/mysort.cpp**
  - Entry point of program.
  - Contains quicksort and bucketsort implementations.
- **source/counter.cpp**
  - Contains the counter microbenchmark and nothing else
- **source/locks.cpp**
  - Contains functioins/methods of the lock classes
- **source/barriers.cpp**
  - Contains the functions/methods of the barrier classes
- **source/include/cxxopts.hpp**
  - CLI argument parser library
- **source/include/locks.hpp**
  - Class definitions of all the locks with supporting `#define` s
- **source/include/barriers.hpp**
  - Class definitions of all the barriers with supporting `#define` s
- **source/include/cpu_relax.hpp**
  - Contains support function cpu_relax() that fit nowhere else.

Compilation instructions

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

```
| Lab2/
| ------ Readme.md
| ------ Writeup.pdf
| ------ Makefile
| ------ Source/
|       |------ mysort.cpp
|       |------ counter.cpp
|       |------ locks.cpp
|       |------ barriers.cpp
|       |------ Include/
|       |       |------ cxxopts.hpp
|       |       |------ cpu_relax.hpp
|       |       |------ locks.hpp
|       |       |------ barriers.hpp
```

To build simply run `make` .

`make clean` will remove object files while `make uninstall` will remove the executables. `make remove` will delete objects and executables though the script is currently pretty stupid and will crash if any removes are missing.

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` .

Counter --help:

```
Counter micro-benchmark for lab 2.
Usage:
  <pwd>/counter [OPTION...] positional parameters
```

```
  -n, --name                    My name (for grading purposes)
  -o, --output FILE             Output file name>
  -t, --threads NUM_THREADS     Number of threads to be used
  -i, --iter Iterations desired
                                Number each thread iterates over
  -b, --bar <sense, pthread>    Selects the barrier type to use.
  -l, --lock  <tas, ttas, ticket, mcs, pthread, aflag>
                                Selects the lock type to use.
  -h, --help                    Display help options
```

mysort --help

```
Text parse and sort for lab 2.
Usage:
  <pwd>/mysort [OPTION...] positional parameters

  -n, --name                    My name (for grading purposes)
  -o, --output FILE             Output file name
  -t, --threads NUM_THREADS     Number of threads to be used
  -b, --bar <sense, pthread>    However no barriers were
                                used in this so run counter to test.
  -l, --lock <tas, ttas, ticket, mcs, pthread, aflag>
                                Lock type to use.
      --alg <fj, bucket>        Algorithm to sort with
  -h, --help                    Display help options
```

Examples:

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`.

To run mysort with a TTAS locking bucketsort on 28 threads and output to ex.txt `./mysort input.txt -o ex.txt --alg=bucket -t 28 --lock=ttas`.

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

Terminal Output

The mysort program will only print the time taken to sort a file and output the sorted list to the specified output file.

The counter program prints the time taken followed by the counter total.

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.

Credits

- cxxopts for command line parsing because nobody (save C++) needs to solve this issue again.
- Clang Format for cleaning my erratic coding styling.
- Dillinger.io for the pretty markdown while the npm and pip world failed me.
- mfukar for their resources on MCS locks
- ifknot for their mcs_lock resources and an asm memory relax instruction.

Extant Bugs

- For some reason when running the counter micro-benchmark with sense reversal barrier selected and roughly 15+ threads iterating over 1000 (or so) the program hangs on thread.join(). It is not all numbers of threads over 15 though and the iteration count seems to matter. If I had not discovered this bug right before the deadline then I would have squashed it but as it is it feels like hunting for co-prime numbers with a shotgun - i.e. entirely ineffective and random.

- 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.