### ECEN 4313 Concurrent Programming

### LAB 2 Write Up

### Elena Murray

# Lab Overview

## Code Organization

The code is organized into three parts: concurrency primitives, counter, and bucketsort. The bucketsort contain most of the same files from Lab 1 with some modifications for handling custom concurrency primitives. Each concurrency primitive was implemented as a class and has its own header and source file. This made it easy to add these primitives to both the counter and bucketsort independently. Both bucketsort and counter have separate main files, executables, and scripts for testing functionality and also collecting performance information.

## Counter Approach

The overall structure is modeled off of test.c which was provided in Lab 1. However, I implemented each concurrency primitive’s variation of counter as its own function. I used enums and function pointer to help determine which function to run and what global variables to initialize. This offloaded the matching of primitive types to inputs from the main function that forked and joined all the threads, which simplified this function.

## Mysort Approach

Most of the structure for bucketsort is the same from last lab, however, like counter each primitive had its own function. Unlike counter, the function was based on a combination of a barrier and a lock. I did this because I only used barriers for timing in bucketsort. In order for me to test my sense barrier with bucketsort, I created functions for each lock using my sense barrier. I also had a function for each lock using the pthread barrier. Overall, I had eight functions four of which used my sense barrier for timing and the rest used the pthread.

# File Structure

* conc\_primitives/
  + sense\_barrier:
    - Contains my implementation of the sense barrier.
  + tas\_lock:
    - Contains my implementation of the tas lock. Uses atomic bools versus atomic\_flags. As a result, the test-and-set function is implemented as exchange().
  + ttas\_lock:
    - Contains my implementation of the ttas lock. Uses atomic bools versus atomic\_flags. As a result, the test-and-set function is implemented as exchange().
  + ticket\_lock:
    - Contains my implementation of the ticket lock.
* counter\_files/
  + counter\_main:
    - Parses in input from the command line, determines which concurrency primitive to use with counter, and outputs the final count to the provided output file.
  + threaded\_counter:
    - Contains all the functions to run a parallelized counter given a concurrency primitive, a number of iterations, and a number of threads.
* mysort\_files/
  + bucketsort\_main:
    - Parses in input from the command line, determines which concurrency primitives to use with the bucketsort algorithm, and outputs the sorted data to the provided output file.
  + threaded\_bucketsort:
    - Contains all the functions to run a parallelized counter given a barrier implementation, lock implementation, a number of iterations, and a number of threads. Barriers are for timing purposes while locks are used to protect buckets.
  + array\_splitter:
    - Contains function for splitting a vector array into subarrays and functions for
    - calulating the standard deviation, max value, and number of buckets.
* test\_cases/
  + counter\_output, counter\_soln:
    - Files used for testing counter.
  + test\_caseN, test\_solnN:
    - Files used for testing mysort.
* Makefile:
  + The Makefile used for compiling and creating the executables.
* counter\_script:
  + Used to test functionality all concurrency primitives for counter with a known thread count and number of iterations.
* counter\_script\_perf:
  + Used to test functionality and collect performance information all concurrency primitives for counter with a known thread count and number of iterations. The performance data includes the level 1 load cache miss rate, level 1 loads, branch misses, and branches. Default number of threads is 10 while the number of iterations is 20.
* mysort\_script:
  + Used to test functionality for mysort with a known thread count and input file. The performance data includes the level 1 load cache miss rate, level 1 loads, branch misses, and branches. Default number of threads is 10 while the default input file is test\_case1.txt.
* mysort\_script\_perf:
  + Used to test functionality and collect performance information all barrier and lock combinations for mysort with a known thread count and input file. The performance data includes the level 1 load cache miss rate, level 1 loads, branch misses, and branches. Default number of threads is 10 while the default input file is test\_case1.txt.

## Concurrency Primitive Analysis

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Counter with 10 Threads and Varying Iterations | | | | | | | | | | | |
| Iteration | Primitive | L1-dache-load-misses | L1-dache-loads | L1 Cache Hits | % L1 Cache Hits | branch-misses | branches | Branch Hits | %Branch Hits | Thread Run Time (s) | Total Run Time (s) |
| 50000 | pthread barrier | 1.22E+09 | 1.25E+10 | 1.12E+10 | 90.20% | 75364888 | 8.61E+09 | 8.54E+09 | 99.12% | 10.00324 | 10.0077 |
| 80000 | sense barrier | 33665696 | 2.54E+09 | 2.51E+09 | 98.68% | 9196754 | 1.17E+09 | 1.16E+09 | 99.21% | 3.998967 | 4.004597 |
| 8000000 | pthread lock | 1.06E+09 | 1.19E+10 | 1.09E+10 | 91.15% | 1.55E+08 | 8.13E+09 | 7.97E+09 | 98.09% | 11.85184 | 11.85628 |
| 8000000 | tas lock | 5.24E+08 | 9.65E+09 | 9.12E+09 | 94.57% | 60438904 | 3.86E+09 | 3.8E+09 | 98.43% | 28.88268 | 28.88588 |
| 500000 | ttas lock | 64416733 | 2.7E+09 | 2.64E+09 | 97.61% | 28927421 | 1.68E+09 | 1.65E+09 | 98.27% | 9.133228 | 9.136935 |
| 500000 | ticket lock | 70096641 | 6.28E+09 | 6.21E+09 | 98.88% | 5506551 | 2.88E+09 | 2.88E+09 | 99.81% | 7.209425 | 7.215357 |

*Table 1: Counter comparison data with 10 threads on a Jupyter machine and varying iterations.*

### Lock Analysis

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Lock Comparison | | | | | | | | |
| Lock | Barrier | Test | L1 Cache Hits | % L1 Cache Hits | Branch Hits | %Branch Hits | Thread Run Time (s) | Total Run Time (s) |
| pthread | N/A | counter | 1139897 | 92.85% | 829520 | 96.45% | 0.000086 | 0.00495422 |
| pthread | mysort | 1,092,797 | 93.43% | 786,649 | 95.98% | 0.000073 | 0.00672696 |
| sense | 1,079,245 | 93.71% | 774,514 | 96.20% | 0.000104 | 0.00587584 |
| Average | | 1,103,979.667 | 93.33% | 796,894.3333 | 96.21% | 8.7667E-05 | 0.00585234 |
| TAS | N/A | counter | 1,131,727 | 92.66% | 823954 | 96.51% | 0.000226 | 0.00521473 |
| pthread | mysort | 1,073,698 | 93.88% | 769,587 | 96.35% | 0.000131 | 0.00551191 |
| sense | 1,079,245 | 93.71% | 934,399 | 97.04% | 1.077E-06 | 0.00519334 |
| Average | | 1,094,890 | 93.41% | 842,646.6667 | 96.64% | 0.00011936 | 0.00530666 |
| TTAS | N/A | counter | 1,132,153 | 91.96% | 826769 | 96.32% | 0.000102 | 0.00478214 |
| pthread | mysort | 1,072,542 | 93.24% | 771,931 | 96.17% | 0.000082 | 0.00620199 |
| sense | 1,315,487 | 94.85% | 883,668 | 96.87% | 1.071E-06 | 0.00600119 |
| Average | | 1,173,394 | 93.35% | 827,456 | 96.45% | 6.169E-05 | 0.00566177 |
| Ticket | N/A | counter | 1183166 | 92.73% | 848348 | 96.43% | 0.000101 | 0.00484747 |
| pthread | mysort | 1,079,188 | 93.81% | 775,724 | 96.41% | 0.000053 | 0.00552391 |
| sense | 1,397,776 | 94.64% | 928,302 | 97.01% | 0.000006 | 0.00545661 |
| Average | | 1,220,043.333 | 93.72% | 850,791.3333 | 96.62% | 5.3333E-05 | 0.005276 |

*Table 2: Lock comparison data with 10 threads on a Jupyter machine, iterations is 20 and input file is test\_case9.txt.*

* Ticket lock
  + Ticket lock has the highest cache hit rate[[1]](#footnote-1) and lowest execution time[[2]](#footnote-2) of all four lock implementations. While there are cache misses due to context switches there are less caused by contention when threads spin, which means overall there are less misses.
* TAS
  + TAS has the lowest cache hit rate[[3]](#footnote-3) of all four lock implementations. This is because contention issues that arise from only having a test-and-set function in the acquire lock loop. Every test-and-set call requires a modifiable copy of the flag variable, which means that all other threads’ local copies of flag are invalidated.
  + TAS does have a high run time[[4]](#footnote-4) compared to other locks. This is because the lock tends to favor the current holder. As a result, during the critical section there are less cache misses, which means that they will execute quicker than a fairer lock.
* TTAS
  + TTAS has a higher cache hit rate than TAS because it resolves the spinning contention issue with flag by spinning on a local copy which gets updated when the lock is released.
  + Like TSA it is faster than the pthread lock and this is because it too favors the favor the current holder.

### Barrier Analysis

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Barrier Comparison | | | | | | | | |
| Barrier | Lock | Test | L1 Cache Hits | % L1 Cache Hits | Branch Hits | %Branch Hits | Thread Run Time (s) | Total Run Time (s) |
| pthread | N/A | counter | 543,9510 | 90.19% | 4112987 | 98.45% | 0.007424 | 0.01362823 |
| pthread | mysort | 1,092,797 | 93.43% | 786,649 | 95.98% | 0.000073 | 0.00672696 |
| TAS | 1,073,698 | 93.88% | 769,587 | 96.35% | 0.000131 | 0.00551191 |
| TTAS | 1,072,542 | 93.24% | 771,931 | 96.17% | 0.000082 | 0.00620199 |
| Ticket | 1,079,188 | 93.81% | 775,724 | 96.41% | 0.000053 | 0.00552391 |
| Average | | 1,951,547 | 92.91% | 1,443,375.6 | 96.67% | 0.0015526 | 0.0075186 |
| sense | N/A | counter | 1,881,366 | 94.94% | 1172951 | 97.23% | 0.000159 | 0.00526573 |
| pthread | mysort | 1,079,245 | 93.71% | 774,514 | 96.20% | 0.000104 | 0.00587584 |
| TAS | 1,424,365 | 95.42% | 934,399 | 97.04% | 1.077E-06 | 0.00519334 |
| TTAS | 1,315,487 | 94.85% | 883,668 | 96.87% | 1.071E-06 | 0.00600119 |
| Ticket | 1,397,776 | 94.64% | 928,302 | 97.01% | 0.000006 | 0.00545661 |
| Average | | 1,419,647.8 | 94.71% | 938,766.8 | 96.87% | 5.423E-05 | 0.00555854 |

*Table 2: Barrier comparison data with 10 threads on a Jupyter machine, iterations is 20 and input file is test\_case1.txt.*

* Sense:
  + Sense has a lower cache hit rate than pthread in Table 3. This could be explained by the fetch-and-increment that all threads call when waiting. The fetch-and-increment call is a source of contention which leads to cache misses. However, this is not the case in Table 1, where the hit rate is much higher for the sense barrier. Sense also has a shorter execution time than pthread. This may be a result of different implementations that prioritize different things.

# Compilation Instructions

1. Run the Makefile using: make
   1. This will generate all object files and the executable.
2. Clean the project using: make clean

# Execution Instructions

* ./counter
  1. Print name:

./counter --name

* 1. Run counter with a barrier:

./counter -t T -i I --bar=bar\_imp -o your\_output.txt

* 1. Run counter with a lock:

./counter -t T -i I --lock=lock\_imp -o your\_output.txt

* 1. counter\_script:

./counter\_script.sh -t T -I I

* 1. counter\_script\_perf:

./counter\_script.sh -t T -I I

* ./mysort
  1. Print name:

./mysort --name

* 1. Run mysort:

./mysort your\_input.txt -t T -i I --bar=bar\_imp –lock=lock\_imp –alg=bucket -o you\_output.txt

* 1. mysort\_script: ./mysort \_script.sh -t T -c C
  2. mysort \_script\_perf: ./mysort \_script.sh -t T -c C

# Known Bugs and Assumptions

1. All files must be text files.
2. Passing fj in the alg flag will still run bucketsort.
3. The tas lock uses atomic bool versus an atomic\_flag because the atomic flag was lock free which caused issues. As a result tas() was done with exchange.
4. If the number of threads exceeds the amount of data, the number of threads will be set to the number of elements. A warning will appear but mysort will still run.
5. There must be at least one thread for both counter and mysort.
6. The number of iterations must be greater than or equal to 1 for counter.

1. Table 1 & 2 [↑](#footnote-ref-1)
2. Table 2 [↑](#footnote-ref-2)
3. Table 1 & 2 [↑](#footnote-ref-3)
4. Table 2 [↑](#footnote-ref-4)