# Locks and Barriers Perf Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Run Time | L1 Cache Hit Rate | Branch Prediction hit rate | Page fault count |
| Tas | 54432ns | 94.2% | 96.8% | 121 |
| Ttas | 168832ns | 88.7% | 96.9% | 133 |
| Ticket | 167588ns | 93.3% | 96.7% | 136 |
| Pthread | 173131ns | 92.1% | 96.6% | 133 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Run Time | L1 Cache Hit | Branch Prediction hit rate | Page fault count |
| Sense Reversal | 59071ns | 96.6% | 97.7% | 140 |
| Pthread | 629496ns | 90.1% | 95.4% | 134 |

(all for counter that increments to 10 with 5 threads)

These results showed that pthread had the longest run time. Everyock and barrier had similar cache hit rates and branch prediction percentages and page fault counts. However, Pthread had the longest run times. This is likely due to the complicated back end that is involved inside of pthread that we don’t know about. This likely makes pthread more robust, but also heavier on the system

# Descriptions

My counter.cpp algorithm does exactly as described in the document: it increments a counter using different primitives that are chosen based on command line input. A control function called handle\_concurrency\_primitive() takes in a position Boolean and inserts a lock or a barrier necessary based on Boolean variables set by the user. That way, the function to lock or barrier wait is a lot cleaner. My locksbarriers.cpp file is a file that contains all the classes and functions necessary to implement my own concurrency primitives. These functions either wait or unlock based on what is needed in the concurrency primitive. The two main functions that are used in this cpp file are fetch\_add() and tas() which are fetch and increment and test and set, respectively. These functions allow atomic reads and writes when dealing with concurrency primitives. The user may choose to sense reversal barrier wait, pthread barrier wait, test and set lock, test and test and set lock, pthread lock, and ticket lock. Of course one may also unlock from all of these.

Mysort is very similar to what was done in Lab1. Instead of having bucketsort and mergesort be controlled by pthread and mutexes, we implemented the primitives inside of locksbarriers.cpp that we also used in counter. This includes barrier wait, pthread wait, test and set, test and test and set, ticket lock, pthread mutex lock, and of course unlock. After reading the user command line inputs, threads are launched and either bucket sort or fork join will launch. Inside of these programs, we call handle\_concurrency\_primitive() which handles the case for locking, and handle\_concurrency\_primitive1() which handles the case for barriers. We need both these functions because locking and barriers need to happen at different points in the execution of threads.

# Code Organization

My code is organized into mysort.cpp and counter.cpp. Inside the makefile, two programs are compiled: counter and mysort. Both of these cpp files include “locksbarriers.h” which corresponds to locksbarriers.c which contains all of the concurrency primitives I described earlier in this document. Inside both of these programs, format of execution is the same. There is some sort of reading from command line input, and then pthread create is called with the void\* function that is the initialization point for all threads. From there, usually one or more functions is called depending on the algorithm that is necessary.

# Description of Files:

AndrewHsu.docx: This writeup document

Counter.cpp: the code that handles the incrementing of the counter, implements my own concurrency primitives as well as pthread.

Counter.h: header file necessary for counter.cpp

Locksbarriers.cpp: contains all the code for concurrency primitives: locks, barriers, etc

Locksbarriers.h: contains all necessary declarations and cleans up locksbarriers.cpp

Makefile: makefile compiles two programs and generates two executebles

Mysort.cpp: sorting program that uses forkjoin or mergesort to sort an input list of integers. Uses concurrency primitives described in locksbarriers.cpp as well as pthread

Mysort.h: header file contains necessary declarations and defines for mysort.cpp

Out.txt: standard output for testfile.txt in mysort

Out1.txt: standard output for counter to 10 using 5 threads counting to 10

Pthreadbarrier.h: copied from github, necessary to use pthread\_barrier\_t on MACOS since POSIX is not enabled on MACOS. This file is not necessary on linux

Testfile.txt: my input test file to be sorted inside of mysort.cpp

# Compilation Instructions

“make” should compile two programs and generate two executebles

“make clean” should remove executebles

# Execution Instructions

A close up of a logo

Description automatically generatedA close up of a mans face

Description automatically generated

Execution is exactly as described in the write up. However, I did not write a mcs lock so choosing this lock will throw an error.

# Bugs

I could not find any bugs as of right now. There are likely some user input bugs that one could try and make things break by forcing the command line inputs to do weird things. This part of the code is not very robust.