Concurrent Programming

Lab 2 Report

Author: Sarayu Managoli

Program Overview:

The program is intended to implement concurrency primitives. These are applied to a stand-alone program named counter and also to the existing bucket sort algorithm. The execution of these programs is dependent upon the inputs provided by the user. The analysis of these programs is done using the "perf" command.

Brief Introduction to the Concurrency Primitives implemented:

Test and Set Lock – It tests a lock whose value is returned, and it sets it. When the returned value us false, the thread can acquire the lock. If the value is already true, it can be assumed that another thread holds the lock.

Test and Test and Set Lock – It is an enhancement to TAS. Reading from memory does not require exclusive access on the cache. First, the cache is tested and if the lock cannot be acquired, it is repeatedly checked for in the cache. On lock release all other cached copies of the lock are invalidated.

Ticket Lock – It is a FIFO lock. The thread holding the number same as that being served gets hold of the lock. This lock is considered to be fairer.

MCS Lock – This type of lock depends on two atomic read-write and modify operation. This is the most useful when there is contention soon after the lock is released. MCS forms a queue of waiting threads. Every thread is just polling the thread that is in the queue before itself.

Pthread Mutex Lock – Man pages quotes that "If the mutex is already locked, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by mutex in the locked state with the calling thread as its owner."

Sense-reversal Barrier – This is another type of synchronization that is used by multithreaded programs. The counter is an atomic variable, and the sense variable is local to the thread. There is one copy of this variable per thread. However, it seems like a global or a static variable.

Pthread Barrier - It can be initialized using the function pthread_barrier_init(). Using pthread_barrier_wait(), threads are made to wait until all the threads call this function.

Brief Introduction to Bucket Sort:[From Lab 1]

Bucket Sort functions by distributing the elements of an array into various number of buckets. Each of the buckets is then sorted individually, either by using an inbuilt function, STL Map, different sorting algorithm, or by recursively applying the bucket sorting algorithm. Bucket sort can be exceptionally fast because of the way elements are assigned to buckets, typically using an array where the index is the value. The average time complexity for Bucket Sort is O(n + k). The worst time complexity is $O(n^2)$. The space complexity for Bucket Sort is O(n+k). Bucket sort is mainly useful when input is uniformly distributed over a range. For this Lab, I have used maps to sort the bucket

Brief Introduction to Pthreads:[From Lab 1]

Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library - though this library may be part of another library, such as libc, in some implementations. A single process can contain multiple threads, all of which are executing the same program. These threads share the same global memory (data and heap segments), but each thread has its own stack (automatic variables). Most pthreads functions return 0 on success, and an error number of failures.

Analysis of Counter program with various concurrency primitives:

Lock	**Runtime (s) as seen in perf**	**L1 Cache Hit Rate**	**Branch Prediction Hit Rate**	**Page Fault Count**
TAS	0.004094841	96.01%	98.10%	161
TTAS	0.008664765	98.31%	99.13%	153
Ticket	0.012181611	99.22%	99.76%	159
MCS	0.003903073	99.28%	99.64%	158
Pthread	0.002749257	91.96%	96.36%	155
Lock	**Runtime (s)**	**L1 Cache Hit Rate**	**Branch Prediction Hit Rate**	**Page Fault Count**
Sense	0.012254473	98.63%	99.37%	168
Pthread	0.002977495	91.53%	96.98%	161

All calculations made for 10 threads with 10 iterations each.

Analysis of Sorting program with various concurrency primitives:

Lock	**Runtime (s) as seen in perf**	**L1 Cache Hit Rate**	**Branch Prediction Hit Rate**	**Page Fault Count**
TAS	0.004452975	94.86%	98.20%	162
TTAS	0.005000234	97.38%	99.66%	163
Ticket	0.012438170	98.93%	99.43%	160
MCS	0.003581865	98.13%	98.82%	162
Pthread	0.011511511	91.76%	97.19%	155
Lock	**Runtime (s)**	**L1 Cache Hit Rate**	**Branch Prediction Hit Rate**	**Page Fault Count**
Sense	0.024773378	99.87%	99.97%	163
Pthread	0.003722841	90.18%	97.21%	166

All calculations made for 100 values and 10 threads

Explanation:

- Although TAS takes lesser time, it can be seen that it has lesser cache hit rate. This is because TAS is more prone to starvation. Every TAS attempt to acquire a lock results in coherence transition. It is more likely to cause a cache miss which can be seen above. TTAS provides an improvement in terms of cache hit rate. The contention in TAS ends up slowing down the movement of control between the cache lines.
- TTAS prevents the above-mentioned contention problems. Hence it can be seen that in both the implementations, TTAS has better cache hit rate. TTAS lock will not spin in test-and-set but increase the likelihood of successful test-and-set by using an entry protocol that spins on normal memory operations
- It can also be seen that mysort TAS has lesser cache hit rate compared to counter TAS. This is because counter is less CPU-intensive compared to mysort. This can be observed for TTAS as well. However, in both, TTAS has a better cache hit rate compared to TAS.
- Branch prediction too is observed to be better was TTAS compared to TAS. However, ticket and MCS have similar prediction rates.
- Ticket is a FIFO lock. This guarantees each thread a lock. It can be seen that its hit rate is better than both TAS and TTAS. However, there might be cache misses each time a lock is released. MCS is hence a better FIFO algorithm.
- Ticket lock has the highest uncontented latency thus justifying starvation. TAS and TTAS have lower value of the same.
- In the both the implementations, cache hit rate for ticket lock is more than TAS and TTAS since storage is not essential since it spins on only one variable (now_serving).
- It can be seen from the hit rates that MCS and ticket provide fairness among the locks implemented. However, it has to be noted that fairness can waste CPU resources if there happens to be an event of preemption.
- MCS is more preferrable [shorter runtime] compared to ticket since each lock release invalidates only the parent's cache entry.
- It can be seen that though sense barrier takes longer time, it has lesser cache misses compared to pthread barrier and it is simple to implement. However, one disadvantage of sense reversal barrier is that it spin-waits on one global flag (sense).
- Also, it can be seen that pthread lock takes lesser time compared to ticket lock. However, it has more cache misses in comparison to all other algorithms. A cache miss issued by an L1 is always one cache line size. As discussed with the professor, 90% cache hit might not be a huge difference as compared to other locking algorithms. The cache rate here indicates the overall misses that occur in the program where in the threads might be completing the iteration one after the other. This might result in cache contention.
- A page fault happens when the virtual memory is not mapped to physical memory. A page fault occurs when a program attempts to access data or code that is in its address space but is not currently located in the system RAM.
- It can be seen that pthread lock has relatively lesser number of faults compared to TAS lock.

Deliverables

As a part of the deliverables, the following artefacts are present in the ZIP folder:

1. Lab write-up
2. Source Files (in folder Code)
3. Makefile (in folder Code)

Source File

This file contains the source code for the implementation of Lab 0. The following functions are included as a part of the source file:

a. mysort.cpp

1. read_from_file(char *read_file_name,vector<int> &readarray) - Reads the integer values from the file specified in the command line argument.
2. write_to_file(char *write_file_name,vector<int> &writearray) - Writes the integer values from the arrays and stores it in a file as mentioned.
3. main(int argc, char **argv) - Initialises the variables and calls other functions.
4. bucket_store(void* arguments) – Thread handler for bucket sort. Uses a map to sort the array.
5. getMax(vector<int>& arr, int size) – Returns the maximum value out of all elements array.

b. mergesort.cpp

6. mergefunction(vector<int>& mergearray, int left, int mid, int right) - Merges the left and the right subarrays onto the main array.
7. sort_merge(vector<int>& mergearray, int left, int right) - Implements merge sort by arranging the elements in an ascending order.

c. algorithm.cpp

8. tasLock(void) – Implements test-and-set-lock
9. tasUnlock(void) – Implements TAS unlock
10. ttasLock(void) – Implements test-and-test-and-set-lock
11. ttasUnlock(void) – Implements TTAS unlock
12. ticketLock(void) – Implements ticket lock
13. ticketUnlock(void) – Implements ticket unlock
14. sense_reverse(void) – Implements sense reverse barrier algorithm

d. counter.cpp

15. write_to_file(char *write_file_name,vector<int> &writearray) - Writes the integer values from the arrays and stores it in a file as mentioned.
16. void* thread_count(void* args) – handler to implement count functionality

Makefile

Makefile consists of two targets, all and clean. The steps to execute are also printed in the file.

Compilation steps:

Open command prompt in the folder Lab2/Code, type "**make**". This will create the executables "counter" and "mysort". To delete the executable and out.txt, type "**make clean**".

Execution steps:

Mysort -

To print the author's name, type "**./mysort --name**"

To run the program in normal mode, type "**./mysort [sourcefile.txt] [-o outfile.txt] [-t NUM_THREADS] [--alg=<fjmerge,lkbucket>]**"

Counter-

To print the author's name in the counter program, type **"./counter [--name]"**

To run the counter program in normal mode, type **"./counter [--name] [-t NUMTHREADS] [-i NUMITERATIONS] [--bar=<sense,pthread>] [--lock=<tas,ttas,ticket,mcs,pthread>] [-o out.txt]"**

Error handling and extant bug:

1. Error handling has been done in mysort if incremental inputs are missing.
2. Also, mysort program exits if the algorithm is selected to be "fj". This has been discussed with the TA.
3. In counter, if lock and barrier are both provided as arguments, it exits.

Extant bugs:

1. Mysort program does not handle the errors related to the inputs from the user being out of order. This can be considered as future scope of work.
2. If the locking or barrier primitive is incorrectly given, TAS lock is automatically used since lock lag is set to 0 as a part of its declaration.

Output:

```
jovyan@jupyter-sama2321:~/Lab2$ make
g++ -pthread -o mysort mysort.cpp mergesort.cpp algorithm.cpp
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
To run the code, please follow the usage as mentioned below.
To print the author's name in the sorting program,
Usage: ./mysort [--name]
To run the sorting program in normal mode,
Usage: ./mysort [sourcefile.txt] [-o outfile.txt] [-t NUM_THREADS] [--alg=<fj,bucket>] [--bar=<sense,pthread>] [--lock=<tas,ttas,ticket,mcs,pthread>]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
g++ -pthread -o counter counter.cpp algorithm.cpp
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
To run the code, please follow the usage as mentioned below.
To print the author's name in the counter program,
Usage: ./counter [--name]
To run the counter program in normal mode,
Usage: ./counter [--name] [-t NUMTHREADS] [-i NUMITERATIONS] [--bar=<sense,pthread>] [--lock=<tas,ttas,ticket,mcs,pthread>] [-o out.txt]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
jovyan@jupyter-sama2321:~/Lab2$
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
Output successfully written to the specified file!
jovyan@jupyter-sama2321:~/Lab2$ ./counter -t 10 -i 12 --lock=tas -o out.txt
Thread count is 10
Number of iterations in each thread is 12

jovyan@jupyter-sama2321:~/Lab2$ cat out.txt
120
jovyan@jupyter-sama2321:~/Lab2$
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
jovyan@jupyter-sama2321:~/Lab2$ ./counter -t 10 -i 10 --lock=mcs -o out.txt
Thread count is 10
Number of iterations in each thread is 10
```

```
jovyan@jupyter-sama2321:~/Lab2$ cat out.txt
100
jovyan@jupyter-sama2321:~/Lab2$
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
jovyan@jupyter-sama2321:~/Lab2$ ./counter -t 10 -i 5 --bar=sense -o out.txt
Thread count is 10
Number of iterations in each thread is 5

jovyan@jupyter-sama2321:~/Lab2$ cat out.txt
50
jovyan@jupyter-sama2321:~/Lab2$
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
jovyan@jupyter-sama2321:~/Lab2$ ./mysort test_case2.txt -o outsort.txt -t 10 --alg=bucket --lock=ttas
Thread count is 10
Num count = 10

Lock flag = 1

jovyan@jupyter-sama2321:~/Lab2$ cat outsort.txt
2
3
4
7
9
10
13
14
16
17
jovyan@jupyter-sama2321:~/Lab2$
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
jovyan@jupyter-sama2321:~/Lab2$ ./mysort test_case1.txt -o outsort.txt -t 3 --alg=bucket --lock=ticket
Thread count is 3
Num count = 5

Lock flag = 2
```

```
jovyan@jupyter-sama2321:~/Lab2$ cat outsort.txt
1
2
3
4
9
jovyan@jupyter-sama2321:~/Lab2$
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
jovyan@jupyter-sama2321:~/Lab2$ ./mysort test_case1.txt -o outsort.txt -t 3 --alg=bucket --bar=pthread
Thread count is 3
Num count = 5

Lock flag = 6
```

```
jovyan@jupyter-sama2321:~/Lab2$ cat outsort.txt
1
2
3
4
9
jovyan@jupyter-sama2321:~/Lab2$
```

~~~~~~

```
jovyan@jupyter-sama2321:~/Lab2$ ./mysort test_case3.txt -o outsort.txt -t 20 --alg=bucket --lock=pthread
Thread count is 20
Num count = 15

Lock flag = 3
```

```
jovyan@jupyter-sama2321:~/Lab2$ cat outsort.txt
2
3
4
5
14
16
19
20
22
23
24
25
26
29
30
jovyan@jupyter-sama2321:~/Lab2$
```

~~~~~~

```
jovyan@jupyter-sama2321:~/Lab2$ ./mysort test_case3.txt -o outsort.txt -t 20 --alg=fj --lock=pthread
Thread count is 20
Num count = 15

Lock flag = 3
Unsorted array is:
19
30
29
14
3
16
20
25
2
4
24
23
22
5
26
--alg=fj is an invalid argument, please choose --alg=bucket
jovyan@jupyter-sama2321:~/Lab2$
```

~~~~~~

References

https://stackoverflow.com/questions/25833541/reading-numbers-from-file-c
~~~~~~

https://codeyarns.com/2015/01/30/how-to-parse-program-options-in-c-using-getopt_long/

https://www.geeksforgeeks.org/bucket-sort-2/

https://stackoverflow.com/questions/48965540/how-do-i-insert-arrays-as-values-into-map-in-c

https://dyclassroom.com/sorting-algorithm/bucket-sort

https://mfukar.github.io/2017/09/26/mcs.html

https://geidav.wordpress.com/2016/03/23/test-and-set-spinlocks/

https://stackoverflow.com/questions/33048079/test-and-test-and-set-in-c

Lecture slides