|  |
| --- |
|  |
|  | **LAB2 REPORT**  **Description of locking algorithms:**   1. **Test and set lock :**The test-and-set lock function writes 1 (set) to a bool variable (which was initially 0) and return its old value as a single [atomic](https://en.wikipedia.org/wiki/Atomic_(computer_science)) (i.e., non-interruptible) operation using the function “exchange”. It keeps on spinning in the loop until the variable is made false by unlock function. 2. **Test and Test and set lock:** A better version of TAS. It polls on the bool variable to check if its false ie, the lock is available and then sets the variable to true and spins on it until the unlock function sets the variable false. 3. **Ticket lock:** Uses two atomic variables as counters for the ticket number currently served (ServingTicketNo) and the ticket number handed out to the next arriving thread (NextTicketNo). The lock serves the threads with the lock and unlock based on order of arrival. Uses atomic fetch and increment to increase the current ticket number count. Until the now serving ticket number is not equal to current ticket number spin in while loop. Serving ticket number is incremented in the unlock function. Both counters are cache line padded to prevent *false sharing*. 4. **MCS lock:  T**here is a queue, which contains a node for each CPU waiting on the lock. Every CPU which wants to wait on the lock allocates a queue node, containing a link (to the next in queue) and a boolean flag and tail which is the previous thread which got the lock. 5. **Pthread lock:** Normal pthread mutex lock and unlock as given my the pthread.h library.   All the locks have been implemented as a derived class from the base class ILOCK containing virtual functions lock and unlock which are over ridden in the lock classes. Given in locks.cpp and locks.h.  **Description of Barrier Algorithms:**   1. Pthread Barrier : Normal pthread sequential barrier APIs as given in the pthread.h library. 2. Sense Reversal Barrier : A Sense-Reversal Centralized Barrier solves the potential deadlock problem arising when sequential barriers are used. Instead of using the same value to represent pass/stop, sequential barriers use opposite values for pass/stop state. It has 2 boolean variables my\_sense and sense flag. The my sense is flipped for each arriving thread. The thread last to arrive sets the sense flag with the value of my sense, until then all the threads wait until the last thread arrives.   All the barrier have been implemented as a derived class of bass class IBarrier containing virtual functions which are overridden in the barrier classes. Given in barriers.cpp and barriers.h  **A description of your two parallelization strategies and microbenckmark**   1. **Microbenckmark:** A multithreaded program where the number of threads and iterations are as given by the user. A common global integer counter is incremented by all the threads depending on their thread id. Locks and barriers have been used to avoid data races. |
|  | 1. **Fork/join:** In this algorithm the main function calls the ‘forkjoin method’ takes in the input vector and number of threads as parameters. It then fills the thread argument structure with the values of the low and high index number of the array calculated for the thread number along with the input array. The thread entry function then retrieves the thread arguments and waits on a barrier for all the other threads to reach at the point. Then it deploys the merge sort and sorts the integers in each thread. Finally after all the threads join the ‘n’ sorted arrays are again merged together to form the final output. |
|  | 1. **Bucket sort:** The main calls ‘bucketsort method’ which takes input array and the no of threads as parameters. Again, the thread argument structure is filled. Each thread inserts some number of integers into different buckets(insertion logic). Multiset container has been used for each bucket. The insertion logic for finding the index corresponding to the integer (as given in the lab document) decides which integer goes into which bucket. The number of buckets is decided by the size of the input array. Bucket holder is the array of ‘n’ empty list. Each list is sorted (multiset). These sorted lists are then concatenated together to form the final sorted list. |
|  | Each algorithm displays the time it takes for the entire operation. |
|  | **A brief description of your code organization** |
|  | The Code organisation is as follows: |
|  | 1. bin : contains the executable of the program "mysort" and “benchmark” |
|  | 2. doc : contains lab report |
|  | 3. files : contains all the testfiles, sorted testfiles by both mergesort as well as sort -n |
|  | 4. includes : contains the header files (.h) of the Lab1 and Lab2 |
|  | 5. obj : contains the object files (.o) generated during make |
|  | 6. script : contains the bashcript to test the entire program framework. |
|  | 7. src : contains all the source files (.cpp) of Lab1 and Lab2. The src is subdivided into common, bucksortsrc and microbenchmark respectively.  8. src/common : contains all the common source files required in bucksort/forkjoin.cpp as well as counter.cpp. These include parse.cpp,locks.cpp and barriers.cpp containing parsing functions, all the locks and barriers implemented respectively.  9. src/microbenckmark: contains counter.cpp the main benchmark file.  10. src/bucksortsrc: contains the files pertaining to sorting |
|  | 11. Makefile : supports make and make clean only. Creates 2 executables in the bin folder: mysort and benchmark. |
|  | 12. Readme : lab details and perf analysis.  13.runscript.sh: script to run the program |
|  |  |
|  | **A description of every file submitted** |
|  |  |
|  | The description of the source files is as follows: |
|  | program.cpp -> contains main. |
|  | parse.cpp -> parses the command line options |
|  | forkjoin.cpp -> handles the fork/join algorithm for sorting |
|  | bucksort.cpp -> handles bucket sort algorithm.  locks.cpp -> contains all the 5 locks implemented tas, ttas, pthread, ticket and mcs.  barriers.cpp-> contains 2 types of barriers pthread barriers and sense reverse barriers. |
|  |  |
|  | **Compilation instructions** |
|  | To compile go to Lab1 main directory and simple make |
|  | $make |
|  | $make clean |
|  |  |
|  |  |
|  | **Execution instructions** |
|  | The input files need to be in a folder called files and the output sorted file is stored in the same folder. |
|  | To execute the program, to go bin directory: |
|  | For bucket sort: |
|  | $ ./mysort input.txt -o output.txt -t10 --alg=bucket --bar=sense --lock=mcs  The other options for lock are tas,ttas,ticket,pthread and bar is pthread. |
|  | For microbenchmark: |
|  | ./benchmark -t2 -i5 --bar=pthread -o out.txt  Or  ./benchmark -t2 -i5 --lock=ttas -o out.txt  Both lock and bar is not recommended. |
|  |  |
|  | **Any extant bugs** |
|  | 1.The number of threads after the ‘-t’ argument cannot be more than 370 or less than 1. |
|  | 2.If the number of elements in the file is less than number of threads given in the argument the program will not proceed.  3. The timing calculation includes the time for launch and join as well as sorting/incrementing counter.  4. The perf Analysis in the excel sheet attached does not seem to support the established behavior of the locks and shows vague results due to internal cache operations that I may be unaware of.  References:   1. <https://en.wikipedia.org/wiki/Barrier_(computer_science)> 2. <https://mfukar.github.io/2017/09/26/mcs.html> 3. <https://github.com/geidav/spinlocks-bench/blob/master/rwlocks.hpp> 4. Concurrent Lectures and example code. |