ECEN 5033 Concurrent Programming

LAB2 Report

Parallelization Strategies:

1. Bucket Sort
    - Again, the given array is divided into sub arrays equal to the given number of threads. Size of the subarray is total elements divided by number of threads.
    - Each thread thread then works on putting the elements into the buckets based on the derived hash value.
    - Hash value is calculated using the equation: (number of buckets * array element)/max element of the array
    - Number of buckets is equal to the number of threads. STL multisets are used as a bucket.
    - As soon as all threads joins, all the buckets are merged into main array.

Code Organization

- Handling Command-line arguments:
  getopt_long() function is used to handle required command-line arguments, --name, -t for number of threads, -o for output file and --alg for selecting the algorithm.
- As the input size (number of integers in the file) is unknown realloc is used to dynamically allocate memory. And thus handling various input sizes also becomes easier.
- After reading the input file and allocating memory for each integer, the array will be divided based on the given number of threads and will be sorted using selected sorting algorithm.
- The synchronization constructs to be used into the program is given by command line is handled through function pointers during runtime.
- Followed by printing sorted data to the output file, if given by the command line else to the stdout.

Description of files submitted

- Makefile: as the name suggests it is a Makefile to compile and build the application. Use **make** on the terminal to do the same.

- Mysort.cpp: this is the source file for the lab. It has the main() function and all other required functions to fulfill the lab requirements.
- Counter.cpp: this file is the source file for incrementing counter value using different lock and barrier.
- Random.sh: this file is a bash script used for generating a test file with random number of integers.
- Log.h: a simple logging system for different log levels. Disabled by default to save time from used by printfs.

Compilation Instructions

- After downloading the submitted .zip file, extract and go into the directory either Counter or Mysort
- Use the **make** command to build and create an executable named **counter** or **mysort.**

Execution Instructions

- ./mysort --name OR ./counter --name prints student name
- ./mysort inputfile.txt -t [num of threads] -o output.txt --alg [bucket] --lock [tas, ttas, ticket,mcs, pthread]
- ./mysort inputfile.txt -t [num of threads] -o output.txt --alg [bucket] --bar [sense, pthread]
- ./counter -t [num of threads] -i [num of iterations] -o output.txt --lock [tas, ttas, ticket,mcs]
- ./mysort -t [num of threads] -i [num of iterations] -o output.txt --bar [sense, pthread]

Performance Analysis for Counter Program

- Locks

Locks	Run Time(ns)	L1 iCache load misses	L1 dCache hit rate	Branch Prediction hit rate	Page fault rate
Test and set	26816376	131,499	99.31%	99.50%	0.735 K/sec
Test and test and set	23834792	137,378	98.12%	98.41%	0.531 K/sec
Ticket	13227728	129,677	98.81%	99.80%	0.932 K/sec
MCS	19156317	132,067	99.62	99.92%	0.658 K/sec

Pthread	16339169	2,280,470	91.33	97.21%	0.001 M/sec

- Barrier

Barrier	Run Time(ns)	L1 iCache load misses	L1 dCache hit rate	Branch Prediction hit rate	Page fault rate
Sense Reversal	57121264	169,935	98.7%	98.87%	0.229 K/sec
Pthread	1397354405	202,591,050	89.97%	97.50%	0.035 K/sec

Performance Analysis for Mysort Program

- Locks

Locks	Run Time(ns)	L1 iCache hit load misses	L1 dCache hit rate	Branch Prediction hit rate	Page fault rate
Test and set	125328637	25,498,069	97.99%	99.88%	0.536 K/sec
Test and test and set	219117	648,477	99.23%	99.41%	0.008 M/sec
Ticket	318566	850,977	99.68%	99.83%	0.004 M/sec
MCS	314312	235,089	99.63%	99.25%	0.038 M/sec
Pthread	200694	25,243,193	91.75%	98.7%	0.028 M/sec

- Barrier

My mysort program is not using Barriers for synchronization purposes.

Description of Best and Worst Performance

From the data in the table shown above, I inferred the following things about the performance.

- MCS lock has the best performance among all locks and barriers. The reason behind this is the fact that MCS lock has the property in which each lock release invalidates only the successor's cache entry. And hence, on test machine's NUMA architecture each thread controls the location on which it spins.
- When it comes to comparison between Test and Set, Test and test and set and Ticket lock, Ticket lock has the best performance while Test and Set lock has the worst performance. The reason is related with processor's caching and locality.
- The Art of Multiprocessor Programming explains that on a multiprocessor system where processors communicate with each other by a bus, both the processor and a memory controller can broadcast on the bus but only one processor can broadcast on the bus at a time. And according to the concept of cache hit and miss, ticket has the best performance because it spins on the same cache line till lock is freed.
- The worst performance in all the synchronization construct comes from the pthread lock and barrier. The reason is the pthread lock and barrier implementation is busy context switch type rather than busy wait in case of atomic.
- When it comes to the performance difference between Sense Reversal and Pthread barrier, sense reversal barrier has better performance because pthread barrier has two counters for keeping track of arriving threads and leaving threads while sense reversal has only one counter to keep track of threads. Thus pthread barrier implementation has two places where processors might run into busy wait situation compared to just one in sense reversal barrier.