ECEN 5033 LAB 2 Write-Up

Name: Nitik Gupta

Performance Analysis:

1. Locks performance for Bucket Sort with 10 threads and 380 different values

Locks	Barrier	Run Time(ns)	L1 Cache Hit rate	Branch prediction hit rate	Page fault
Test and set	Pthread	784853	95.64	97.91	181
Test and TAS	Pthread	753854	97.74	98.8	181
Ticket	Pthread	1054457	98.17	99.05	182
MCS	Pthread	1082896138	99.99	99.99	183
Pthread	Pthread	1809566	95.18	97.75	180
Test and set	Sense	22274685	99.74	99.9	181
Test and TAS	Sense	19258746	99.92	99.94	181
Ticket	Sense	35990287	99.97	99.98	183
MCS	Sense	993586778	99.99	99.99	186
Pthread	Sense	40018650	99.97	99.98	181

2. Locks performance for counter with 10 threads and 100 iterations

Locks Used	Run Time(ns)	L1 Cache Hit rate	Branch prediction hit rate	Page fault
Test and set	109155	91.48	96.67	152
Test and test and set	134168	91.55	96.77	153
Ticket	86798	91.67	96.59	152
MCS	414212	95.26	97.84	160
Pthread	172370	91.61	97.67	147

3. Barriers performance for counter with 10 threads and 100 iterations.

Barrier Used	Run Time(ns)	L1 Cache Hit rate	Branch prediction hit rate	Page fault
Sense Reversal	18832580	99.81	99.88	153
Pthread	124227	91.68	96.68	148

Lock Performance:

From the performance of the code, it is evident that the locks using the local variables like the MCS, Ticket or Sense Reversal have a way better cache performance than others. This can be since the local variables are stored in stack [1], so that they can be accessed quickly, so in terms of memory, they can be stored in either the registers or the L1 Cache. therefore there are more cache hits for these in comparison to others. Page faults have a decrease when we use pthread.

Barrier Performance:

In terms of branch prediction, everything almost every lock has the similar behavior, the only noticble difference it between the usage of the sense reversal and pthread barriers, and since we really dont know what happens inside the pthread barrier, we really can't predict anything what is really happening inside that. Page faults have a decrease when we use pthread.

Sorting Algorithms

I. Bucket Sort

It starts with creating an empty bucket (in our case, and int to int map). Then we will take the unsorted vector and divide it into n number of parts where n is number of threads. After that, threads are created and then in each thread, the value from the vector is inserted inside the map, which helps in sorting automatically by inserting in the first part of the map which is automatically sorted while inserting.

II. Merge Sort

It is a divide and conquer type of algorithm, where each vector is divided into small vector from the middle and the smaller vectors are into further smaller until it reaches a point where there is only 1 value in an array and then two arrays can be compared and sorted individually and in the end combined the same vector it was given.

Locking Algorithms

I. Test and Set

It uses a global atomic bool variable which will be set or cleared when acquired by a thread.

II. Test and Test and Set

A variation of TAS, just it will first test the variable and then try to acquire the lock.

III. Ticket Lock

It uses two variables, the next num and the now serving variables, the next num variable tells you about what number you have acquired for lock, the now serving variable tells about which number has acquired the lock. So, until and unless, the next num and now serving are equal, the thread can’t acquire the lock.

IV. MCS Lock

It uses a queue type of locking mechanism where the first thread will acquire the lock and the rest will keep on incrementing on the tail. As soon as the thread releases the lock, the next item in queue will acquire the lock and so on util the queue is empty.

Barrier Algorithms

I. Sense Reversal

It uses two global variables: sense and counter and a local variable my sense. The global will be initialized to number of threads, and other to 0. As each thread gets to the barrier wait, it will increment the counter and wait until the counter is equal to number of threads. As soon as they are equal, every thread will be released.

Code Organization:

The code is organized based on their common functionality and their usage. There are 5 different c++ files in the project: main.cpp, util.cpp, mergesort.cpp, counter.cpp and bucketsort.cpp. As the name suggests, util file contains all the miscellaneous functions used in the utility of the main function. Similarly, mergesort and bucketsort contains the functions used for the sorting algorithm. And the main file contains the main function. The counter.cpp contains the microbenchmark program for testing purposes. The header files for each file contains the function declarations of the functions used in C++ file. In the next topic, the functionality of each function is given in detail.

File Description:

- main.cpp: Contains the main function where the command line arguments are captured using the getopt_long function. These arguments are then processed for their respective functionality.
- makefile: makefile for the project which contains the commands to compile the code ad create the object files of each program and then combine in the mysort object file for the execution of the whole program.
- mergesort.cpp: Contains the function definitions required to perform mergesort operation on the given numbers.
- mergesort.h: Contains the function declarations required to perform mergesort operation on the given numbers.
- util.cpp: Contains the function definitions of all the miscellaneous utility functions that performed inside the main function like print, write to file etc.
- util.h: Contains the function declarations of all the miscellaneous utility functions that performed inside the main function like print, write to file etc.
- Bucketsort.cpp: Contains the function definitions required to perform bucketsort operation on the given numbers.
- Bucketsort.h: Contains the function declarations required to perform bucketsort operation on the given numbers.
- Counter.cpp: Contains the function definitions required to perform counter micro benchmarking.
- Counter.h: Contains the function declarations required to perform counter micro benchmarking.
- Locks.cpp: Contains the function definitions of all the locking functions that performs different locking mechanism required for assignment.
- Locks.h: Contains the function declaration and classes of all the locking functions and barriers that performs different locking and barrier mechanism required for assignment.

Compilation Instructions:

The makefile can used to help in compilation. Just call make or make all to compile the object file and use them to execute the program. To delete all the object files, you can use make clean in the command line.

Execution Instructions:

Mysort Object file Instructions:

- After the compilation of the program, you can use the ./mysort instruction on the command line, to execute the program. There are 7 arguments you can give to the ./mysort instruction (you must at least give the input file for successful execution of the program). These are --name, input file (without any argument variable), output file (with variable -o), number of threads(with variable -t) ,algorithm(with varible --arg) required to sort, lock type to be used (with variable --lock) and barrier to be used(with variable --bar).
- The --name argument will print out the name and that’s it. But it will also exit the file.

- The input file doesn't require a seperate variable, but without an input file, the code will not run and will exit the program.
- The output file uses the variable –o. If Output file is provided then it will write to it, if not it will print on the console.
- Number of threads uses the variable -t. If thread number is provided, then it will use the number of threads given by user to create that many numbers of threads, if not it will use 5 threads.
- Locks uses the --lock variable to define the lock to be used. If no lock is defined, then by default pthread is used.
- Barriers uses the --bar variable to define the barrier to be used. If barrier lock is defined, then by default pthread is used.
- Last is the algorithm with variable --alg, it can take two values, one is merge and other is quick, according to the value they will perform fj and bucket. If something else is given, or nothing is given, it will perform mergesort by default.

Counter Object file Instructions:

- After the compilation of the program, you can use the ./counter instruction on the command line, to execute the program. There are 5 arguments you can give to the ./counter instruction. There are no compulsory arguments.These are --name, output file (with variable -o), number of iterations (with variable -I), number of threads(with variable -t) ,algorithm(with varible --arg) required to sort, lock type to be used (with variable --lock) and barrier to be used(with variable --bar).
- The --name argument will print out the name and that’s it. But it will also exit the file.
- The output file uses the variable -o. If Output file is provided then it will write to it, if not it will print on the console.
- Number of threads uses the variable -t. If thread number is provided, then it will use the number of threads given by user to create that many numbers of threads, if not it will use 5 threads.
- Locks uses the --lock variable to define the lock to be used. If no lock is defined, then by default pthread is used.
- Barriers uses the --bar variable to define the barrier to be used. If barrier lock is defined, then by default pthread is used.
- Number of iterations use -I for counter, if nothing given it will default to 20.

Extant Bugs:

1. If you provide more command line variables than the required, then the input file might not be taken by the program, and there won't be a sort program execution.
2. If values in the input file is greater than INT_MAX(+2147483647), then the program will not be able to apprehend them and will give wrong answers.

Version Control Link:

https://github.com/IMNG7/Concurrent_Lab2