ECEN 5033 Concurrent Programming

**LAB2 WRITE UP**

# **Introduction:**

This document is about the bucket sort, merge sort algorithms which are used to sort an unsorted text file and send it to a new or existing file in sorted manner. It also contains counter program which updates the value of count for each thread every time till the number of iterations requested and prints the final count value from all threads in the output text file.

**# Description of code organization:**

Consists of main.counter,cpp, bucket\_main.cpp and Makefile

* The lab1 code has been organized in three files which is the main\_counter.cpp, bucket\_main.cpp and Makefile. These two programs are different compiled with Makefile. It generates two executables mysort and counter on running make.

**# Description of every file**

* Makefile compiles the code main\_counter.cpp, bucket\_main.cpp and generates executable that can be run by typing make.
* main\_counter.cpp and bucket\_main.cpp contains the main two executables of the project.
* The bucketsort program file takes the inputs as - - name, -o sortedfile.txt - - alg=fj or - - alg=bucket, -t thread\_count and –lock={tas, ttas, mcs, ticket, pthread} and –bar={sense, pthread}. --name argument prints the full name of the person and alg argument takes options as fj and bucket which sort the file according to the option specified. the locks options need to be one of the selected lock among the options above. The barrier type will be one of the selected barriers out of the required options.
* The counter program file takes the inputs as - - name, -t thread\_count, -o out.txt -i iterations and - -lock={tas, ttas, mcs, ticket, pthread} or –bar={sense, pthread}. --name argument prints the full name of the person. The concurrency primitives need to be one of the selected lock or one of the selected barriers out of the required options, both cannot be specified together.
* For both -t option takes input as number of threads which needs to be run in parallel and for counter program -i option for each thread is the number of iterations to be incremented by.
* Bucketsort program takes a non-option argument which is the unsortedfile.txt. This file will have multiple lines which each line having integer of size 0 till INT MAX.
* Both the program takes -o argument which in case of mysort program takes option as the sorted file eg.sortedfile.txt with sorted output form unsorted file. The numbers will be sorted and arranged line by line in the sorted file, in this case sortedfile.txt. For counter program, final value of count incremented by all threads will be put in the output text file.
* The bucket\_main.cpp contains the merge sort algorithm functions, bucket sort algorithm function and function to read and write to the file. Reading the file is done by using char \*\*array and converting the array to an integer 1d array. This array will be passed to the mergesort and bucketsort functions depending on the request from the user.
* The sorted array will then be written to a new or an existing file line by line.
* The file is read using getline which takes row of the file as row of the array and columns of the file as columns of the array. This read array is converted using strtol to an integer array. The char \*\*array which reads input of from the unsorted file uses dynamic memory allocation and reallocates for file content greater than size of initially defined array.
* The main\_counter.cpp file contains counter\_value which is incremented by each thread, i or iteration number of times. This final count value is then written to the output text file at the completion of the program.
* In both the programs the elapsed time at the end will be printed on terminal in sec and nanosec.

# **Parallelization Strategy for Merge sort**

* The input file copied into an array will be divided into sub arrays where each individual thread will work on approximately same number of array elements.
* The low array index, high array index, array to be populated, thread number is calculated and passed for each thread by using a structure of these values
* New threads are created, and their specific indexes and unique parameters are passed to the merge thread function which will sort individual sub arrays using recursive merge sort.
* The threads are joined on completion of their unique tasks.
* The individual sub arrays are then finally merged and is sent to the sorted\_file.txt

# **Parallelization Strategy for Bucket sort**

* The input file copied into an array will be divided into sub arrays where each individual thread will work on approximately same number of array elements.
* The number of buckets will be equal to number of threads.
* The low array index, high array index, array to be populated, thread number, individual array length and bucket divider is calculated and passed for each thread by using a structure of these values.
* Divider is calculated by dividing the max element of array by number of threads used to filter each element of array into separate sorted buckets, created using stl::map.
* New threads are created, and their specific indexes and unique parameters are passed to the bucket sort thread function where each thread elements are checked to see which bucket they will go to according to their values in comparison to value of max element in the total array. They are then added to individual sorted buckets.
* The threads are joined on completion of their unique tasks
* The individual sub arrays are then finally combined to the final main array to create a list of elements present in a sorted order.
* This sorted array is then written into a new or an existing text file.

**# Compilation instructions**

* Eg of how to run the bucketsort project. First compile the file using **make** which creates an executable of the name **mysort**.
* Eg of how to run the counter project. First compile the file using **make** which creates an executable of the name **counter**.

**# Execution instructions**

* Then run the mysort executable using the command by typing

**./mysort - - name unsortedfile.txt -o sortedfile.txt -t10 alg=<bucket, fj> - - lock=<tas, ttas, ticket, mcs, pthread> - - bar=<sense, pthread>** which sorts the unsorted file using sortin program with fj or bucket taking 10 number of threads using the chosen concurrency primitives.

* Then run the counter executable using the command by typing

**./counter - - name -o sortedfile.txt -t num\_threads – i num\_iteration - - lock=<tas, ttas, ticket, mcs, pthread> or - - bar=<sense, pthread>** which starts the counter value incremented by each of the num\_threads num\_iteration times and prints final counter value in the output text file.

**# Extant bugs**

* The input values are less than **INT\_MAX,** will lead to differences in the actual sorted file run using **sort -n** linux command and the **mysort** executable. Hence, the same should be considered when running the file.
* The datatype features of std::map throws erroneous results when taking very very large inputs leading to segmentaion fault.
* The std::map vector does not sort repeated numbers.
* Sense barrier stops execution for large values for counter program.
* **Counter with 10 threads and 10000 iterations with locks or barriers respectively**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Locks | L1 cache load reference | L1 cache load misses | Branch predictions | Branch misses | Page faults | Runtime  nanosec |
| tas | 98.44% | 1.56% | 99.54% | 0.46% | 145 | 6283395 |
| ttas | 98.38% | 1.62% | 99.59% | 0.49% | 147 | 6462046 |
| mcs | 97.52% | 2.48% | 99.11% | 0.89% | 937 | 13641225 |
| ticket | 97.49% | 2.51% | 98.14% | 1.86% | 147 | 14258956 |
| pthread | 94.09% | 5.91% | 97.65% | 2.35% | 147 | 6354795 |
| Barriers | L1 cache load reference | L1 cache load misses | Branch predictions | Branch misses | Page faults | Runtime  nanosec |
| phtread | 89.17% | 10.83% | 98.94% | 1.06% | 152 | 164927899 |
| sense | 98.5% | 1.50% | 99.23% | 0.77% | 151 | 167616289 |

* **Sorting with 10 threads, 382 numbers and 8 digit max number with locks and barriers respectively**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Barrier | Locks | L1 cache load reference | L1 cache load misses | Branch predictions | Branch misses | Page faults | Runtime  nanosec |
| pthread | tas | 95.24% | 4.76% | 97.54% | 2.46% | 172 | 1069078 |
| pthread | ttas | 96.83% | 3.17% | 98.45% | 1.95% | 173 | 2185719 |
| pthread | mcs | 92.34% | 7.66% | 97.21% | 2.79% | 175 | 2453902 |
| pthread | ticket | 93.35% | 6.65% | 97.27% | 2.73% | 172 | 2462539 |
| pthread | pthread | 91.98% | 8.02% | 97.1% | 2.90% | 175 | 876927 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Barrier | Locks | L1 cache load reference | L1 cache load misses | Branch predictions | Branch misses | Page faults | Runtime  nanosec |
| sense | tas | 99.92% | 0.08% | 99.96% | 0.04% | 180 | 529890 |
| sense | ttas | 99.67% | 0.33% | 99.8% | 0.20% | 172 | 619867 |
| sense | mcs | 98.01% | 1.99% | 98.98% | 1.02% | 184 | 632965 |
| sense | ticket | 99.61% | 0.39% | 99.77% | 0.23% | 175 | 1731907 |
| sense | pthread | 97.35% | 2.65% | 98.97% | 1.03% | 177 | 557325 |

**# Discussions**

* Mcs for counter program has significantly higher page faults and lowest is for tas. The sense barrier has generally more page faults for bucketsort program. The reason for mcs could be link list type implementation, using a queue for waiting threads. Branch prediction rate is highest for ttas in both programs but lowest for pthread in case of bucketsort and counter program. Ttas lock branches predicted is high.
* Ticket lock has the max runtime and tas lock has minimum runtime in both programs.
* Tas lock has maximum cache hits for counter program and ttas has maximum cache for bucket sort. The lowest cache hits is pthread lock in both programs. It could have a multiple unique acceses in its internal structure hence, pthread has such performance.
* The sense revival barrier has more cache hits for counter in counter program and bucketsort program.
* The reason for better performance for sense barrier is that each thread keeps it own, local sense to detect changes and maintain same phase for all threads.
* The analysis of all the different locks shows us that tas and ttas have higher cache hits than mcs, ticket and pthread lock generally. As mcs and ticket are FIFO locks, they result in better fairness but more cache misses than tas and ttas.
* The runtime measurement also shows that tas and ttas are comparatively faster than mcs and ticket lock because of more cache hits. In ticket lock more threads might give lock to the one descheduled and this type of FIFO lock there is cache miss on every lock release.
* LIFO locks like tas and ttas might bottleneck the system due to same core thread trying to acquire lock. We reduce coherence traffic with ttas compared to tas by avoiding cache line pinging all around the system if lock is already held and wait till it is released.
* There are some results in both the programs which are other than expected. The reason could be the handling of the program by the operating system and the hardware with different capabilities.