CONCURRENT PROGRAMMING

LAB 2

VIGNESH IYER

No of threads: 5

Machine tested: Intel Core i7-3537U 2 Core 2.1 GHz

Bucket Sort								
		Cache						
	Time(ms)	Load	Store	Prefetch	Load Miss	Store Miss	Prefetch Miss	Page Fault Count
TAS	0.32	-	-	-	0			94
TTAS	0.24	-	-	-	0			92
Ticket	0.3	-	-	-	0			91
MCS	0.4	-	-	-	0			100
Sense	1	-	-	-	0			120
Mutex	0.5	-	-	-	0			93

Counter								
		Cache						
	Time(ms)	Load	Store	Prefetch	Load Miss	Store Miss	Prefetch Miss	Page Fault Count
TAS	0.32	-	-	-	0	30	39	94
TTAS	0.24	-	-	-	0	10	21	92
Ticket	0.3	-		-	0	25	40	91
MCS	0.4	-		-	0	0	10	100

Sense	1	-	-	-	0	0	0	120
Mutex	0.5	-	-	-	0	0	0	93

The calculations done are based on testing and seeing missing the cache rate and done theoretically. The timing analysis are done on the machine itself i.e. the code does the calculation.

The worst timing is of the mutex locks due to the fact that the atomic operations are much faster as compared to mutex operations. The reason for this is atomics is a very costly operations are takes into options in order to avoid the data race which can be used either using sequential consistency or relaxed consistency. The best timing analysis was for the MCS locks which are more efficient as compared to TAS and TTAS as well as the ticket locks. The reason is that TAS is a FIFO lock which has high cache miss as well as for the ticket lock, it has the same issue. TTAS is LIFO lock which has a high cache hit rate but is unfair to threads. The MCS lock is an efficient lock which allows the high cache hit rate and average fast. Mutex locks are the worst due to the fact that contention is high and slower due to the fact that the atomic operations are faster than normal operations. Thus it is an issue and based on the average analysis, it is implied that the MCS lock is the most optimized lock to be used.

The bucket sort algorithm, divides the given array in the buckets which are equal to the number of threads. Then each of the bucket is allocated a memory according to the given structure which gives the thread number and is implemented as a list. The bucket size is determined based on the modulus operation of the total elements to the number of thread. If the remainder is zero then the bucket size is equal to the ratio of the total elements to the number of threads/buckets. If the modulus is a non-zero number, then the bucket size is equal the number of elements to (no of threads-1). Thus each bucket is then sorted using the insertion sort algorithm due to the fact that it is more stable and occupies less memory as compared to quick sort as is faster. As well as due to fact that insertion sort is better when the elements are added in a sequential manner. The elements are sorted in each buckets using the total buckets divided by the element and the ratio value in integer determines the

bucket number. The insertion sort will first insert the first element and proceed to compare the incoming element with the first and will put it if the incoming element is less than the first one. Again it recursively checks for the next element and compares with the elements inside the bucket. Then a lock is put in each thread to ensure no memory is accessed in the process which could cause data race. Thus then each of threads are joined and the final array is given as an output. The lock selection is done with the basis of the input of the user which could either of the five given locks i.e. TAS, TTAS, Mutex, MCS or Ticket. If the user argument is given as a sense reversal barrier the locks argument is ignored. The timing is calculated for the thread id =0 and the cache miss is calculated on a theoretical basis.

The counter takes the number of iterations as an input from the user and the value is incremented by the counter on the basis of the user input. The lock selection is done with the basis of the input of the user which could either of the five given locks i.e. TAS, TTAS, Mutex, MCS or Ticket. If the user argument is given as a sense reversal barrier the locks argument is ignored. The timing is calculated for the thread id =0 and the cache miss is calculated on a theoretical basis.

Steps to run:

1. Run make all
2. The execution is: ./mysort –name inputfile –o outputfile –t Num_Threads –alg=bk –bar=<sense,pthread> --lock=<tas,ttas,ticket,mcs,pthread> → For bucket sort.
3. The execution is: ./counter --name –t Num_Threads –-i=NUM_Iterations --bar=<sense,pthread> --lock=<tas,ttas,ticket,mcs,pthread> –o outputfile → For counter.

Difficulties Faced:

The problems faced are the implementation of the MCS lock and was not properly working. Also the code is not modular enough to be easy to read. The branch-prediction rate was not

done and I am not sure how it is done. The file format is not supported, also when calculated cache misses were zero for some reason which I am not sure of.

Also the usage of the atomic relaxed consistency has some issues. I was not able to find any bugs in the code (But it doesn't mean it is not there). Also consistency issues were also faced. The Threading policy was also an issue but was resolved.

The other difficulties faced were the use of header files for make due to some weird issues it was not compiling in the beginning, but then the value of the input file was found to null and void. Thus the code worked later on.

References:

[1] https://stackoverflow.com/questions/34143724/error-maybe-you-meant-to-use

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

[3] http://libfbp.blogspot.com/2018/01/c-mellor-crummey-scott-mcs-lock.html

[4] https://bartoszmilewski.com/2008/12/01/c-atomics-and-memory-ordering/

[5] http://concurrencyfreaks.blogspot.com/2014/05/relaxed-atomics-optimizations-for.html

[6] https://lwn.net/Articles/590243/