Multiprocessor Architecture Assignment 3

Sayak Chakraborti NetID-schakr11

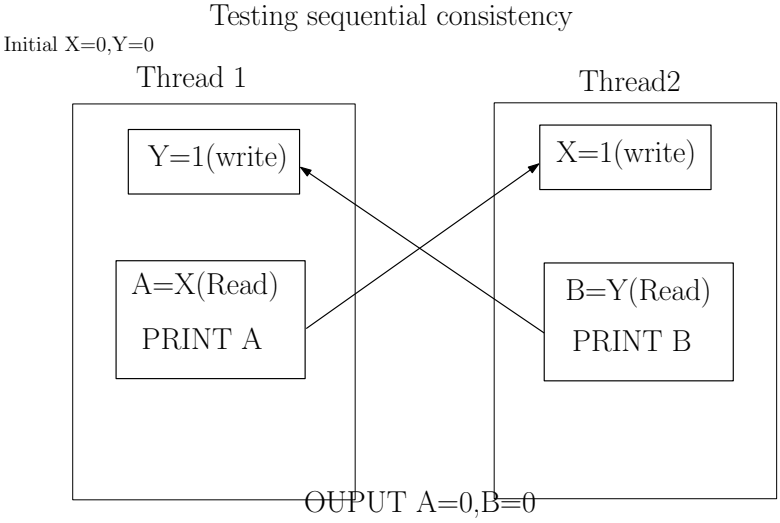
February 2017

Question 1

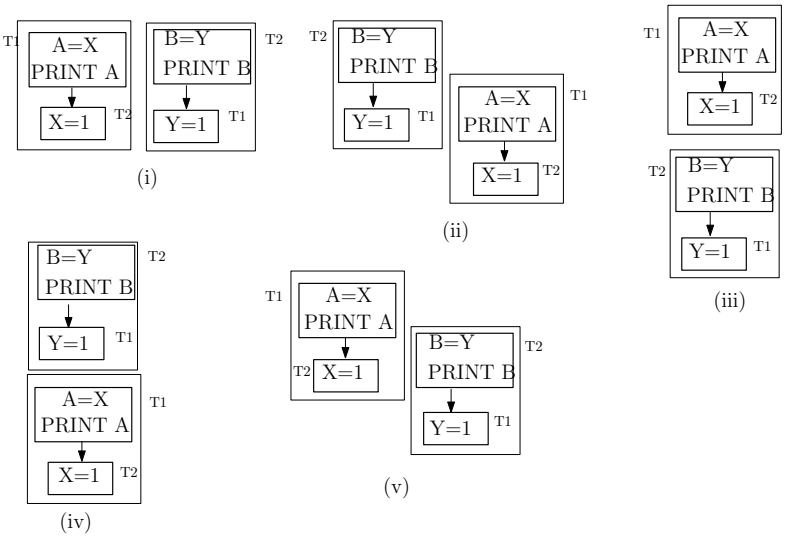
Consistency model (30pts): Write a pthread-based multi-threaded program to figure out as much as you can about the machine's underlying consistency model (without using any consistency model-specific primitives such as a write-memory-barrier). Explain your approach, your observation after running the program, and the conclusion you can draw from the experiment.

Answer 1

So the idea is to test whether the load− >load,load− >store,store− >load and store− >store orders.We know that sequential consistency maintains all the four orders for a particular program/thread on a core.Thus my first test was to see whether it withhold the store− >load order reordering which can be done using store buffer and is the most exploited reorder observed in processors to gain better performance.(shown in figure below)



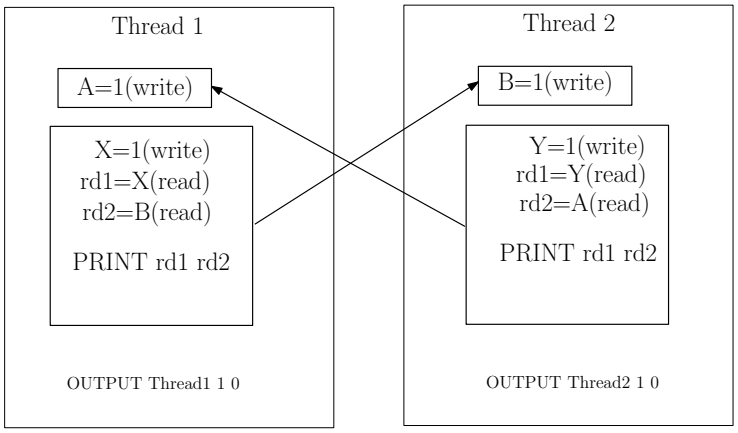
Now running the two threads and observing outputs as A=0 and B=0,intuitively gives us two happens before relations between the threads T1 and T2.These suggest that read Y(B=Y) in thread 2 happened before write Y(Y=1) in thread 1 and read X(A=X) in thread 1 happened before write X(X=1) in thread 2.We can argue on the order of these two happens before graphs being executed with respect to time(the direction of time grows downwards).So each of the possible combination(figure below) of the executions it is revealed that either one of the threads or both of them reorder the write to occur after a read.Example in case (iii) for Thread 1 write Y happens much later than read A=X,similarly for other cases it can be said that at least one of them relax the store− >load order and thus violate sequential consistency for performance benefit.



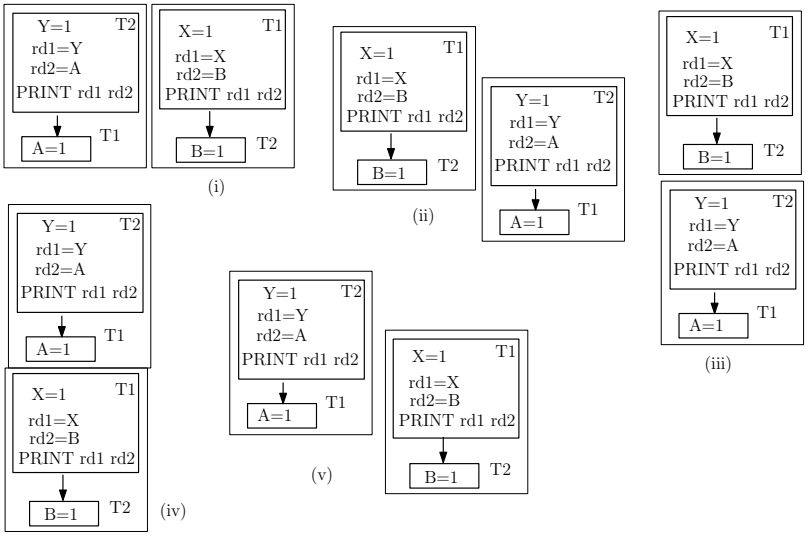
As we know that if we violate the store− >load order then it falls into the next consistency model of total store order which guarantees maintaining the store− >store order.Thus now we have ascertained that our machine does have the memory consistency model of the sequential consistency.Hence I devised the second experiment which again has two threads and two writes in them to check whether they maintain the order of the two stores or not.(shown in figure below)

Testing total store order

Initial X=0,Y=0,A=0,B=0



Running the two threads and observing an output of Thread1 rd1=1(X) rd2=0(B) and Thread2 rd1=1(Y) rd2=0(A0, again gives us two happens before relations between segments of the codes of thread1 and thread2.These suggest that Thread1 write X(X=1) and read B(rd2=B) happened before Thread2 write B(B=1) and Thread2 write Y(Y=1) and read A(rd2=A) happened before Thread2 write A(A=1).Again we can argue on the order in which these happens before graphs can be represented with respect to time(figure below).So in each of the possible combinations we see either one of the threads or both of them have reordered.For example in (iv) we see that write to Y happens before write to B ,which infers that the stores have been reordered.



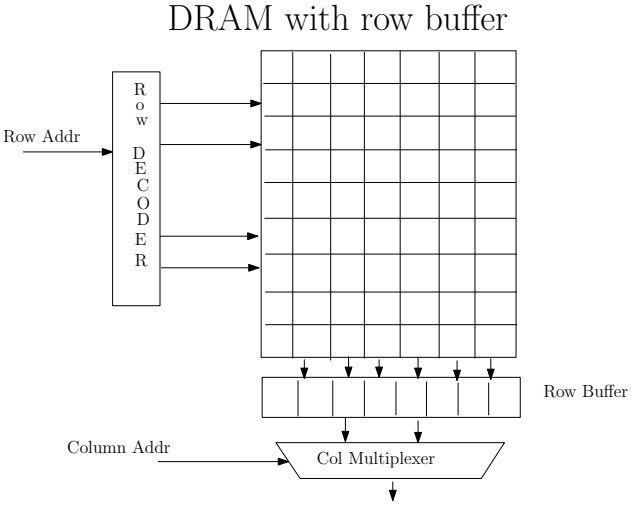
So the store− >store order is violated ,which violated the total store order(TSO) memory consistency and falls to partial store order(PSO) which allows both store− >load reorder and store− >store reorder.Now we ascertain that the memory consistency model is not even TSO but PSO.
I ran all these experiments on my laptop which has a configuration of architecture x86_64 ,4 CPUs and 3MB cache.

Question 2

False sharing (20pts): Take your matrix multiplication program from Assignment 1, instead of distributing the workload by blocks of lines, do so by elements. For example, thread 0 will calculate the end result of [0,0], thread 1 [0,1], thread 2 [0, 2], and so on. Compare the execution time of the two versions and explain the difference.

Answer 2

Sequential access to the arrays when each thread is given the task of computing n set of rows causes each thread to access memory locations contiguously ,which causes more hits in the cache due to spatial locality and for matrix the size 4000x4000 which can't fit inside the cache ,the access to memory(DRAM) is faster when accessing memory locations that are contiguous row buffer(see diagram below).Thus the less the row buffer has to be flushed the better it is(less time required).



But in the new implementation where the threads are accessing locations or rows that are not contiguous (element counter modulo thread number),there

is very less spacial locality that can be exploited in the cache and thus cause cache misses which result in cache penalty(time).Also these kind of accesses cause repeated flushing of the row buffer in the DRAM which adds on to the time spent in getting the data.Thus we see a degraded performance in the new implementation compared to the previous one.

Machine	Benchmark	Number of threads	Execution time in secs(Old)	Execution time in seconds(new)
CYCLE3	mat200x200	1	0.037009	0.086021
CYCLE3	mat200x200	4	0.027164	0.059788
CYCLE3	mat200x200	8	0.013599	0.079309
CYCLE3	mat200x200	12	0.021567	0.097113
CYCLE3	mat200x200	16	0.017279	0.101478
CYCLE3	mat200x200	20	0.018956	0.120018
CYCLE3	mat200x200	24	0.013756	0.121650
CYCLE3	mat200x200	28	0.013829	0.171799
CYCLE3	mat200x200	32	0.010909	0.178692
CYCLE3	mat200x200	36	0.069362	0.194159
CYCCLE3	mat200x200	40	0.065160	0.212446
CYCLE3	mat4000x4000	1	831.330	860.921152
CYCLE3	mat4000x4000	4	218.926790	884.856041
CYCLE3	mat4000x4000	8	170.149558	968.173
CYCLE3	mat4000x4000	12	141.658557	1182.4589
CYCLE3	mat4000x4000	16	103.261475	1106.818517
CYCLE3	mat4000x4000	20	85.035733	1193.862264
CYCLE3	mat4000x4000	24	82.589171	1372.9757
CYCLE3	mat4000x4000	28	82.047838	1467.455
CYCLE3	mat4000x4000	32	81.9641	1653.283
CYCLE3	mat4000x4000	36	88.2459932	1763.96
CYCCLE3	mat4000x4000	40	141.336496	1939.402

Also in the new implementation each thread is going over the matrix to check which of those positions they need to compute for(that means they simply perform a modulo operation to assert whether they need to compute or not).By computation I mean multiplication and addition.The going over the matrix and modulo operation is adding onto the time consumed in the new implementation.

At threads 36 I am crossing the number of nodes in cycle3 and hence the irrational behavior is observed.

Increasing threads in the late(new) implementation causes the execution time to go up bit by bit instead as there is a lot of misses and competition for getting hold of an element in the same row as someone else,so now each thread is crossing paths in some way with one another and that causes the rise in execution time.

Question 3

MPI-based implementation (30pts): Take your pthread-based matrix multiplication program from Assignment 1, replace the reliance on pthread support with MPI support. Compare the execution time using up to 4 threads and running on a single machine with that of using pthread. Then, enlist at least one other

machine in your MPI implementation, maximize the performance, and measure the execution time. Discuss the result in your report.

Answer 3

Message passing Interface requires us to create multiple processes instead of multiple threads to do the same computation.Threads share the same address space hence they can work on the same data structure with different indexes and no or less passing of information among the threads.But for processes as they don't share the same address space thus everything like inputs to each of the threads has to be passed as messages to the slave processes by a master process (which in this case also does some work).The results where as follows,

Machine	Benchmark	Number of threads	Execution time in secs(1)	Execution time in seconds(MPI)
node2x18a	mat200x200	1	0.058938	0.050931
node2x18a	mat200x200	2	0.038363	0.028214
node2x18a	mat200x200	3	0.039907	0.019094
node2x18a	mat200x200	4	0.038345	0.014171
node2x18a	mat200x200	5	0.043202	0.011866
node2x18a	mat200x200	6	0.019942	0.011766
node2x18a	mat4000x4000	1	676.59403	464.73265
node2x18a	mat4000x4000	2	334.684169	243.840
node2x18a	mat4000x4000	3	229.041	164.117
node2x18a	mat4000x4000	4	171.307	118.606
node2x18a	mat4000x4000	5	141.27	103.9041
node2x18a	mat4000x4000	6	120.407	89.78

From the results we see that the implementation with Message Passing Interface yields lower execution.Message passing copies the data required by each process for computation.Thus when we send and receive messages we simply create a copy of the data that will be used by the received process and thus it can carry on it's computation without bothering any other process.While for pthreads as they share the same address space they have a single copy of the data which causes maybe reads to the same variables and data by multiple readers (threads) which may cause read races and thus **time consumed to read and write same cache lines causing multiple invalidation messages and cache coherence protocol change of state**.Which is not the case for message passing because they are different copies.
For smaller data set either of them is good enough for a bigger data set ,MPi hay have overheads of copying memory and time required to do that.

Question 4

Mixed support (extra credit *30pts): When you have both the support of MPI and pthread, what can you do to improve performance. Discuss your measured result.

Answer 4

pthreads are threads which share the same address space and thus don't require duplication of data to be made available to them for fraction of the computation.On the other hand for Message Passing Interface each process has separate address space and thus have to be communicated among using messages to get some work done.This address space sharing is implicit in threads and we don't ave to worry about communicating among threads as such.