[ECE 6101] Parallel and Distributed Computer Architecture
Assignment 4

Q1. Solution)

Part – a):

Bus	Action	P1	P2	P3	P4	Comment
	Initially	S	S	S	S	Initially, P1 holds the lock
1	St1 (BusRdX)	M	I	I	I	P1 releases the lock
2	Ld2(BusRd)	S	S	I	I	P2 read misses after invalidation
3	t&s2 (BusRdx)	I	M	I	I	P2 executes test and set and issues busrdx
4	St2	I	M	I	I	P2 releases the lock, no bus request
5	Ld3(BusRd)	I	S	S	I	P3 read misses after invalidation
6	t&s3 (BusRdx)	I	I	M	I	P3 executes test and set and issues busrdx
7	St3	I	I	M	I	P3 releases the lock, no bus request
8	Ld4(BusRd)	I	I	S	S	P4 read misses after invalidation
9	t&s4 (BusRdx)	I	I	I	M	P4 executes test and set and issues busrdx
10	St4	I	I	I	M	P4 releases the lock, no bus request

Total minimum bus requests generated : 7

Bus	Action	P1	P2	P3	P4	Comment
	Initially	S	S	S	S	Initially, P1 holds the lock
1	st1 (BusRdX)	M	I	I	I	P1 releases the lock
2	ll2(BusRd)	S	S	I	I	P2 load link read misses after invalidation
3	sc2 (BusRdx)	I	M	I	I	P2 executes store conditional and issues busrdx
4	st2	I	M	I	I	P2 releases the lock, no request generated
5	ll3(BusRd)	I	S	S	I	P3 read misses after invalidation
6	sc3 (BusRdx)	I	I	M	I	P3 executes test and set and issues busrdx
7	st3	I	I	M	I	P3 releases the lock , no request generated
8	ll4(BusRd)	I	I	S	S	P4 read misses after invalidation
9	ll4 (BusRdx)	I	I	I	M	P4 executes test and set and issues busrdx
10	St4	I	I	I	M	P4 releases the lock, , no request generated

Total minimum bus requests generated : 7

Part – b):

Bus	Action	P1	P2	P3	P4	Comment
	Initially	S	S	S	S	Initially, P1 holds the lock
1	St1 (BusRdX)	M	I	I	I	P1 releases the lock
2	Ld2(BusRd)	S	S	I	I	P2 read misses after invalidation
3	Ld3(BusRd)	S	S	S	I	P3 read misses after invalidation
4	Ld4(BusRd)	S	S	S	S	P4 read misses after invalidation
5	t&s2 (BusRdx)	I	M	I	I	P2 executes test and set and issues busrdx
6	t&s3 (BusRdx)	I	I	M	I	P3 executes test and set and issues busrdx
7	t&s4 (BusRdx)	I	I	I	M	P4 executes test and set and issues busrdx
8	Ld3(BusRd)	I	I	S	S	P3 read misses after invalidation
9	Ld4	I	I	S	S	P4 read misses after invalidation, no bus request
10	st2(BusRdx)	I	M	I	I	P2 releases the lock
11	ld3(BusRd)	I	S	S	I	P3 read misses after invalidation
12	ld4(BusRd)	I	S	S	S	P4 read misses after invalidation
13	t&s3 (BusRdx)	I	I	M	I	P3 executes test and set and issues busrdx
14	t&s4 (BusRdx)	I	I	I	M	P4 read misses after invalidation
15	St3 (BusRdx)	I	I	M	I	P3 releases the lock
16	Ld4(BusRd)	I	I	S	S	P4 read misses after invalidation
17	t&s4 (BusRdx)	I	I	I	M	P4 executes test and set and issues busrdx

Max request is : 16

Bus	Action	P1	P2	P3	P4	Comment
	Initially	S	S	S	S	Initially, P1 holds the lock
1	St1 (BusRdX)	M	I	I	I	P1 releases the lock
2	ll2(BusRd)	S	S	I	I	P2 read misses after invalidation
3	ll3(BusRd)	S	S	S	I	P3 read misses after invalidation
4	ll4(BusRd)	S	S	S	S	P4 read misses after invalidation
5	sc2 (BusRdx)	I	M	I	I	P2 executes store linked and issues busrdx
6	sc3	I	M	I	I	P3 failed on store conditional, no request

7	sc4	I	M	I	I	P4 failed on store conditional, no request
8	ll3(BusRd)	I	I	S	I	P3 read misses after invalidation
9	ll4(BusRd)	I	I	S	S	P4 read misses after invalidation
10	st2(BusRdx)	I	M	I	I	P2 releases the lock
11	sc3	I	M	I	I	P3 failed on store conditional, no request
12	sc4	I	M	I	I	P4 failed on store conditional, no request
13	ll3(BusRd)	I	S	S	I	P3 read misses after invalidation
14	ll4(BusRd)	I	S	S	S	P4 read misses after invalidation
15	sc3 (BusRdx)	I	I	M	I	P3 executes store linked and issues busrdx
16	sc4	I	I	M	I	P4 failed on store conditional, no request
17	ll4(BusRd)	I	I	S	S	P4 read misses after invalidation
18	st3(BusRdx)	I	I	M	I	P3 releases the lock
19	sc4	I	I	M	I	P4 failed on store conditional, no request
20	ll4(BusRd)	I	I	S	S	P4 read misses after invalidation
21	sc4(BusRdX)	I	I	I	M	P4 failed on store conditional, no request
22	St4	I	I	I	M	P4 releases the lock, no request

Max request is 15.

Q2:

```
int  B= 0;

BARRIER( barrier_t bar, int N) {
        If (__atomic_fetch_and_add(B,1) == N -1)
                { B = 0; }
        else { while (B != 0); }
 }
```

The above barrier will not work in two scenarios:

1. Suppose when last thread N hits the barrier and makes B =0. This causes invalidations on all the N-1 threads spinning on B. Now the last barrier moves on and hits another barrier performing a fetch and add on B(considering its BusRdx got access before contending BusRd from N-1 threads). This causes the barrier to fail as rest other threads will be still stuck in the old barrier while one or a few threads might race ahead.

2. Consider a case where the spinning thread in the barrier is context-switch out due to an interrupt or gets de-scheduled by OS and the contents of the thread register file is saved on thread control block. Now the terminating or last thread arrives at the barrier and causes B to go 0. Now when the preempted thread is context switched in it will still spin on stale value of variable B. Hence the barrier will fail.

We can avoid this by using a sense reversal barrier. Below is the code for the sense reversal barrier:

```
typedef struct _SenseReversalOpenMP_Barrier{
	long count;
	volatile int* global_sense;
	int* mysense;
	long max_count;
}SenseReversalOpenMP_Barrier;

void SenseReversalOpenMP_BarrierInit(long num_threads, SenseReversalOpenMP_Barrier* barrier){
	int i;
	barrier->count = num_threads;
	barrier->mysense = (int*)malloc(sizeof(int)*num_threads);
	barrier->max_count = num_threads;
	*(barrier->global_sense) = FALSE;
	for(i=0;i<num_threads;i++){
		barrier->mysense[i] = TRUE;
	}
}

void SenseReversalOpenMP_Barrier_Wait(SenseReversalOpenMP_Barrier* barrier,long threadid){

	assert(threadid < barrier->max_count);

	__atomic_fetch_and_add(&barrier->count,1);

	if(barrier->count==0){
		barrier->count = barrier->max_count;
		*(barrier->global_sense) = barrier->mysense[threadid];
	}
	else{
		while(*(barrier->global_sense) !=  barrier->mysense[threadid]);
	}

	if(barrier->mysense[threadid]==1){
		barrier->mysense[threadid] = 0;
	}
	else{
		barrier->mysense[threadid] =1;
	}
}
```

Above each thread spins on its local sense. Now when the terminating thread toggles the global sense and all threads are released. While leaving the barrier all threads toggle their local sense so as to be ready for next barrier. Volatile qualifier ensures that global sense is always read from memory. Hence, we avoid both the scenarios as mentioned above.

Q3:

Below is the requested analysis of all the three locks:

No. of Threads	Omp Lock	Array Based Lock	Ticket Lock
1	0.4052	0.3959	0.40489
2	0.26188	0.2615	0.25977
3	0.1896	0.19101	0.18818
4	0.150321	0.15089	0.1500818
5	0.1276	0.131202	0.126593
6	0.120016	0.122218	0.11848741
7	0.1088245	0.112798	0.1078648
8	0.10263064	0.107908	0.1016187
9	0.093185	0.1019857	0.093798
10	0.0905404299	0.1004749	0.09296221
11	0.088939	0.09638655	0.088391
12	0.0856562	0.09633738	0.0868928

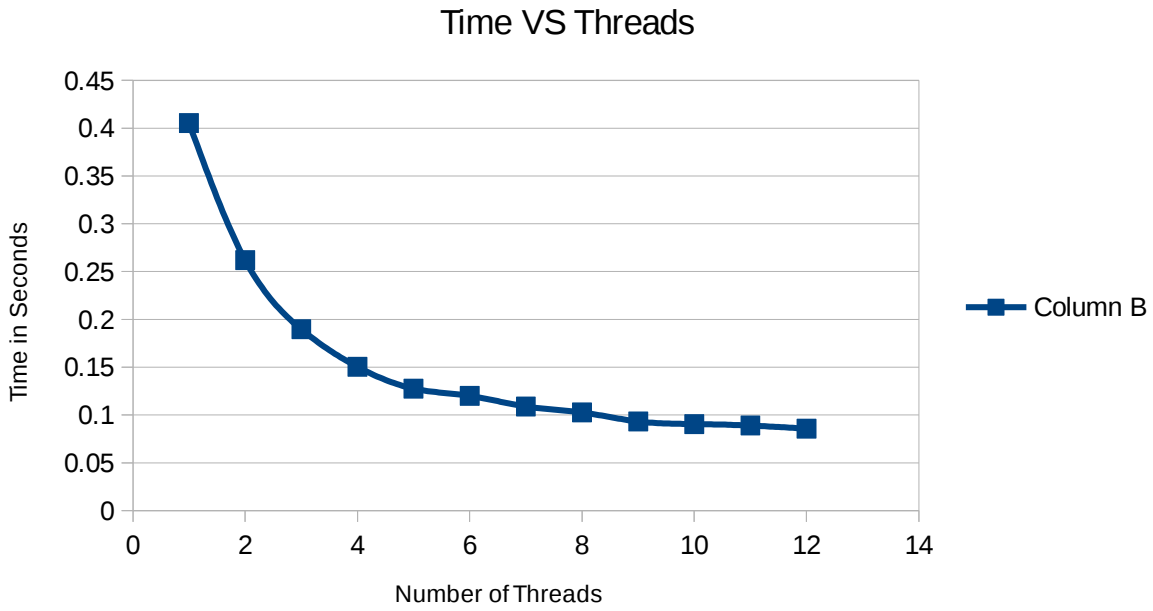

Figure 1: Omp Lock - Time vs threads

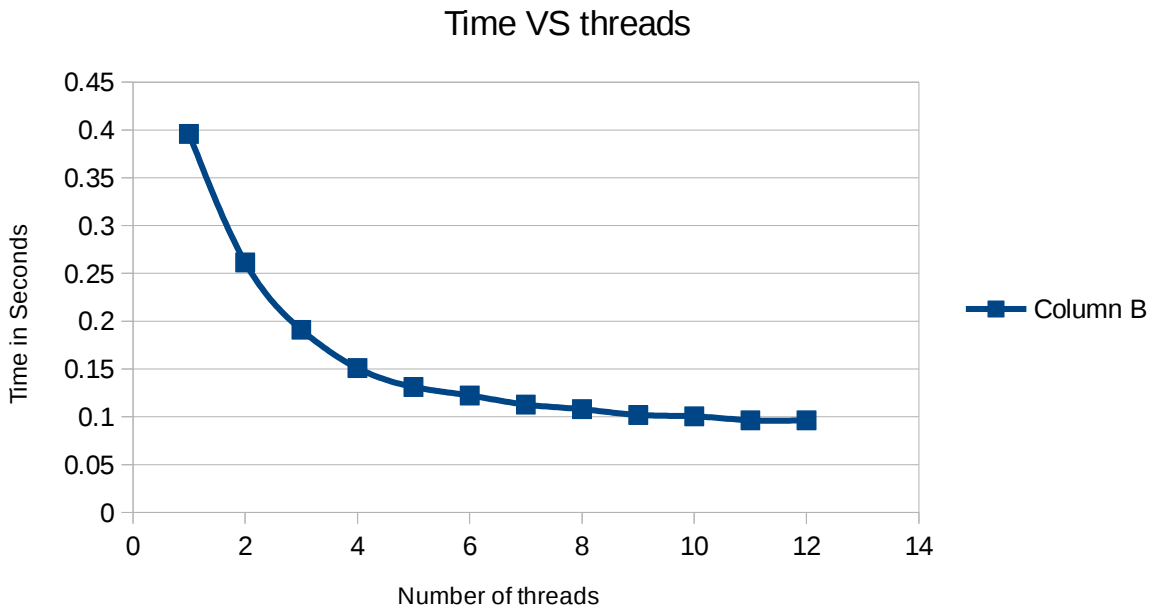

Figure 2: Array Based Lock - Time vs threads

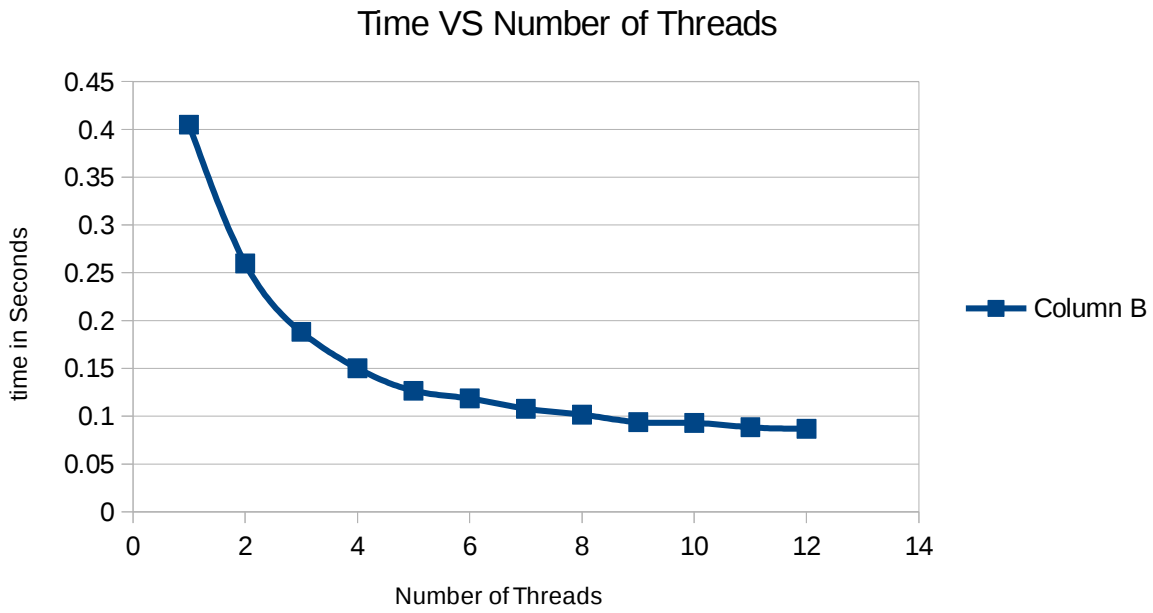

Figure 3: Ticket Based Lock - Time vs threads

System Configurations:

1. The above code was run on 2 6-core Intel Westmere processors, on Gatech's jinx cluster.
2. Each run was done 100 times and averaged to get desired results. The data set was of 10000 elements

and random seed was 100.

3. The threads were twice as the number of cores, i.e., for 6 cores I used 12 threads.

Observations:

1. Nothing conclusive can be said about the performance of the three locks. The average time fails to show any pattern apart from scaling down with the number of threads.

2. Average time shows no constant improvement over any lock, i.e., it increases and decreases for multiple threads.

Conclusions:

1. If we understand the architecture, we can categorize the three locks as shown below:

a. Omp Lock: As per the documentation of OpenMP, it is implemented using a simple spin lock using test and set having a contention of the order of $O(P^2)$, *where P is the number of processors.* Thus omp lock should perform the worst. But as the given workload is not timed to cause maximum contention, our results fail to prove this. Also, omp lock is not fair but has a small memory footprint.

b. Ticket Lock: Ticket lock is fair, has a heavier memory footprint than omp lock but has similar contention as compared to omp lock, of the order of $O(P^2)$. Here too our results fail to prove any correlation to normal architectural beliefs.

c. Array Lock: Array lock has the heaviest memory footprint(of the order of $O(P)$), is fair and has contention of the order of $O(P)$. But due to aforementioned reasons our results fail to prove this.

The given workload fails to provide any conclusive results to validate the architectural beliefs of constructing the above locks.

Source Code Details;

The source code is provided in the given tarball. There are three folders:

/givenlock

/arraylock

/ticketlock

Use make in each folder to compile. Run using ./out <Data size> <Random Seed> <No. of threads>

Use Regress.py to evaluate regression results:

python Regress.py -r <rcmd(./out)> -i <input binary folder> -o <output folder> -d <data element size> -m <no of threads> -t <timout>