INTRODUCTION

As per Assignment we need to implement sudoku Validator using dynamic Task Allocation (not fix which thread performs what task/check which operation) using parallelism and handle synchronization(race conditions) Using TAS , CAS , BOUNDED CAS and run sequential too to compare.

How to run them is given in Assgn2Readme-CO23BTECH11022.txt

WORKING OF CODE AND ALGORITHM

Common Algorithm For TAS,CAS,Bounded CAS implementation

1. Take Some global Variables so they can be accessed by anyone who wants to perform some operation. Ex:-
   K(Number of threads)
   N(Dimension of sudoku)
   tasInc(How much increment value of C where C is Global shared Variable)
   n(Dimension of sudoku)
   sudoku(2-D grid),
   Sudoku_is_Valid (To tell sudoku is valid or not)
   and some time calculating Total time and Total Critical section entry(Help to find average (Total/occurence)).
   Take Atomic variable:- C , Lock . So no one else can modify them during their execution.

   Take BASE_START_TIME and initialise it when the main function starts actual work (before creating threads), to find how much time operations are taking with respect to our Base time.
   Means , C++ has library chrono by using that to find current time and subtract it from base_time , we get execution time .

Taking all Times in Microseconds because that's the smallest convenient range where we identify difference of time.

2. From main function take input K(Number of thread) , N(dimension of matrix), taskInc(How much add to C) and Sudoku From inp.txt and store in global variables by use of read_object of Ifstream class and close object .

3. To store id of each thread in an array use pthread_t threads[k] and create all K threads to execute check_sudoku function having thread id as argument(For 1 base indexing add +1 in id) .

4. Now create K threads using **pthread_create**(parameters:- thread_id,NULL, function which each thread executes- **check_sudoku** , function parameters) and suppose when they all executed join them using **pthread_join**(thrad_ids, NULL) - It will wait for all threads to finish.
   All computational tasks completed now ,so store ending *time .*
   *(ending -base*_start_time = Total execution time).

   Print whatever asked Using Global variables which store all things.

   Now we see the check_sudoku function skeleton .

   check_sudoku(){

   ```
   while(CONDITION){
       // ENTRY  SECTION

       //LOCK (to satisfy mutual exclusion)

       //CRITICAL SECTION

       //UNLOCK (when critical section work is done)
   ```

```
            //REMAINDER SECTION (to perform non critical task )

        }
        pthread_exit(NULL)  //to terminate calling threads
}
```

ENTRY SECTION: -

Threads will come here and request to Enter in the Critical section . We will store Request_Time and print which threads are requesting for CS as asked in the assignment.

LOCK : -

It makes Sure that at a time only one Thread can go in the Critical Section and the rest all have to wait until CS gets free(Therefore providing mutual Exclusion and avoiding race conditions) . This can be done By:

TEST_AND_SET LOCK() ;- Use C++ Wrapper . test_and_set() gives previous lock value and update lock value to 1 , so it will run again and again until lock sets to 0, which is called spinlock too.
It will ensure Mutual Exclusion and progress but cannot handle Bounded Waiting means some threads may go to starvation .

COMPARE_AND_SWAP() : - Use C++ Wrapper.
compare_exchange_strong() atomically compares lock variables with expected value .if they match ,replace with current value of lock. It also has a spinlock . It will also ensure Mutual Exclusion and progress but cannot handle Bounded Waiting .

So to add Bounded Waiting (Each will get chance/no starvation) we used

BOUNDED_CAS():- Handle bounded waiting so each thread get chance to execute with use of :
Atomic variable (BCAS_LOCK) initialized with zero, used as lock.
Boolean array (waiting) where each thread marks itself as waiting.

Locking Algorithm:

function lock_BCAS(take thread id)
Sets waiting for that thread/mark True.
Repeatedly try to acquire lock using(compare_excahge_strong) Of CAS.
Once lock is acquired make Waiting as False.

Unlocking Algorithm:

Find next thread which is waiting in circular order(modulo by k Or size of the waiting array) .
If found, execute it , else release the lock.

CRITICAL SECTION(section accessing shared resources)

We have atomic C which is a shared variable, incremented by taskInc each time when some thread enters CS.
We use Global variable C_value which stores C value before incrementing C,(because it helps for task allocation) .

Calculate C increment time , Worst entry time, Total entry time of CS, Total CS entries and store them globally so we use them for Questions asked in assignment.

Handling Edge Condition (when shared counter C >= 3*N)

Means our all Task/operations (Total 3*N) are assigned dynamically. So we have to stop incrementing C , C maximum value is minimum of C+taskInc and 3*N . (Handling it also)

Some thread still come under entry section and as soon as current Thread leave CS and release lock , Those Entry section threads get Entry in CS section . So, to stop them for increment of C → Terminate them and before that release lock so rest thread can also Come and we terminate them also.

Caution : If we do not do the above solution , we will face Deadlock condition (If not unlock before terminating).

Now leave CS (do task distribution in remainder section) so that we Can perform many operations and make CS efficient.

UNLOCK ;-

TAS and CAS unlock : - just make lock available / or make it free(0) so threads waiting in the entry section to get CS.

BOUNDED CAS : - above explained

REMAINDER SECTION: -

In this, perform Task allocation by shared C value .
Logic : Number of Task to execute go from C_previous value to min(C_prev+taskInc,3*N) .
taskType is finded by(Task/N). 0 means row, 1 means column and 2 means subgrid.
Getting index of taskType by (Task%N) ,because for validation we N entities .

Check According to taskType And index by help of validating row, Column, subgrid functions.

CONDITION (in while)

We should run program till C(shared variable) is less than 3*N , not Equal to because we initialise C with 0 (not 1).
So if C>=3N it will enter CS and increase C too , that also we don't Want and try to perform tasks which are already executed.

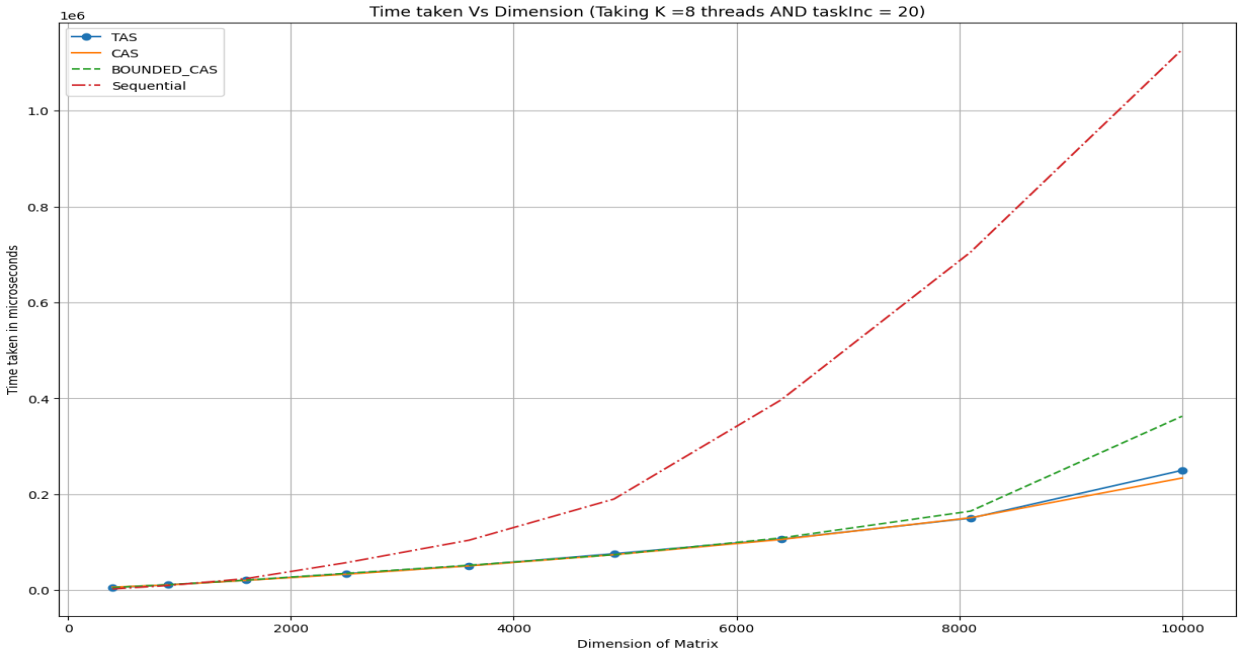
EARLY TERMINATION ON TAS, CAS , BOUNDED CAS

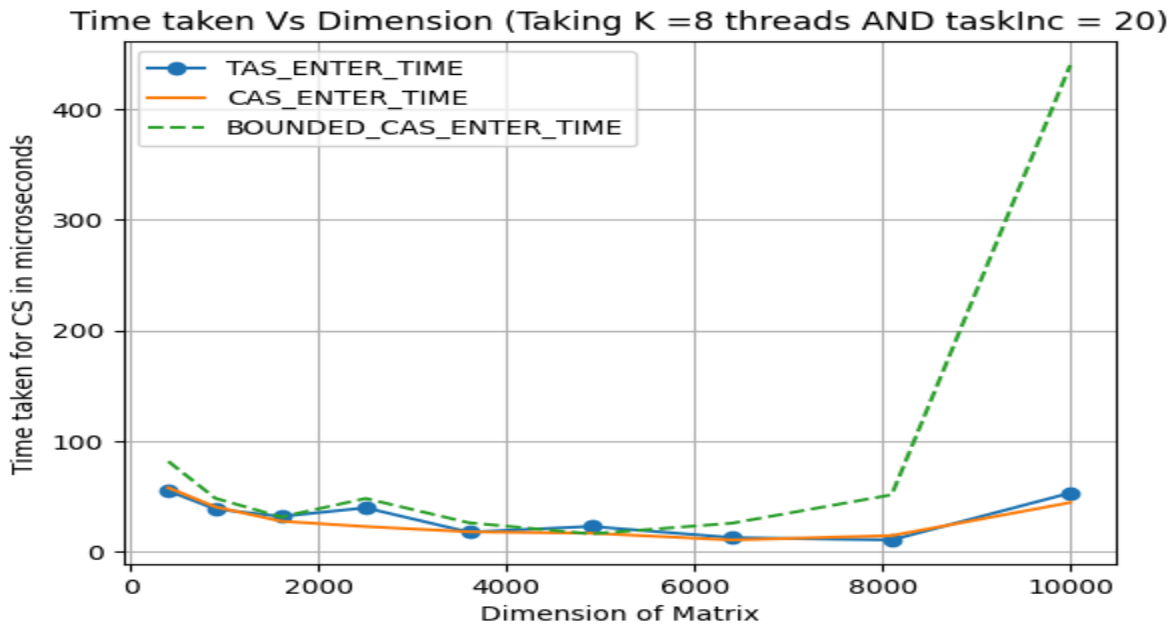
In all 3 above method we will perform early termination means as Soon as we Know there is some invalid row/coloumn/subgrid it will Stop executing further by the Help of Global variable Sudoku_isvalid Initialize with true initially.

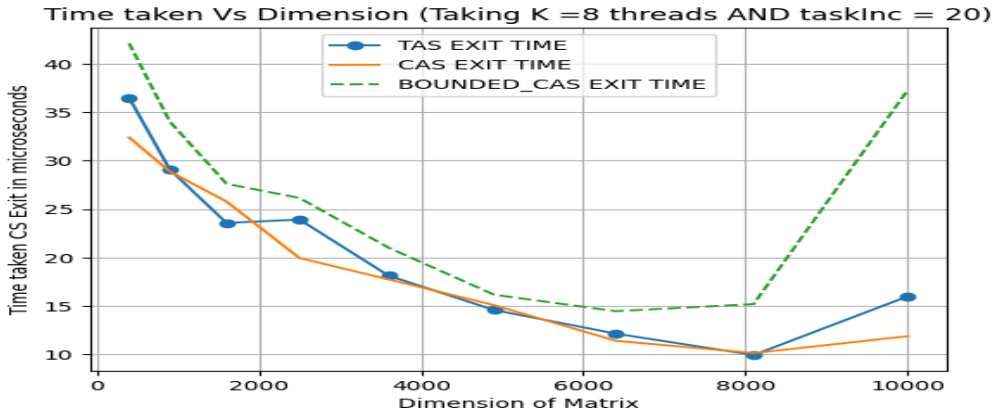
To early terminate we have to update while CONDITION, EDGE CASE and One more check in remainder Section , just by that Global variable .
Modify While CONDITION : while(sudoku_isvalid==false && C<3XN)

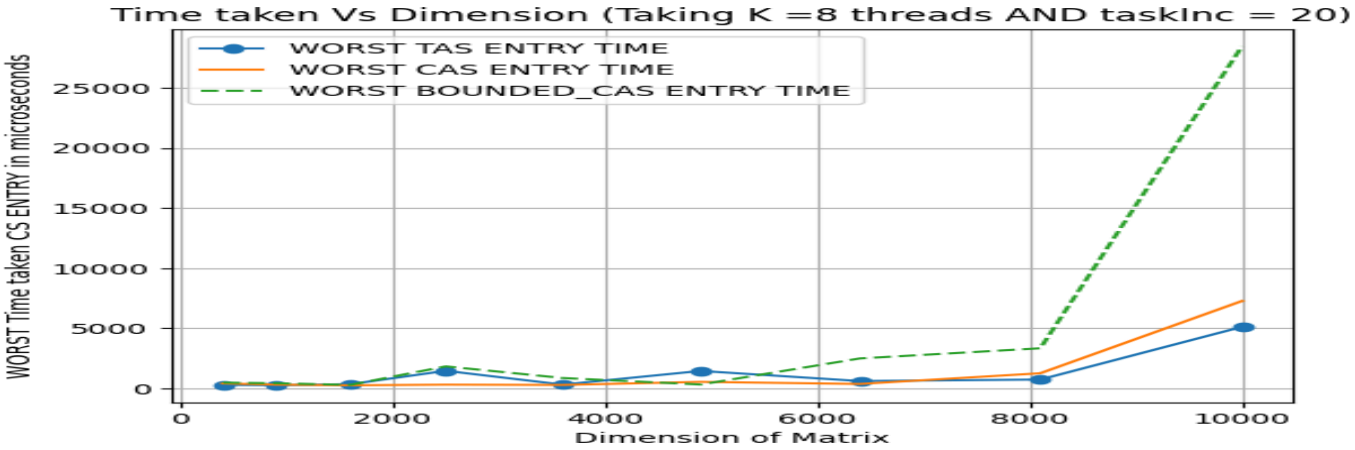
For sequential , while checking validation if found invalid , just break From all operations .
Sequential is different from single threaded program because in Sequential we do not use any thread just normally run it .
But for single threaded , we have to create thread and import specific Libraries too.

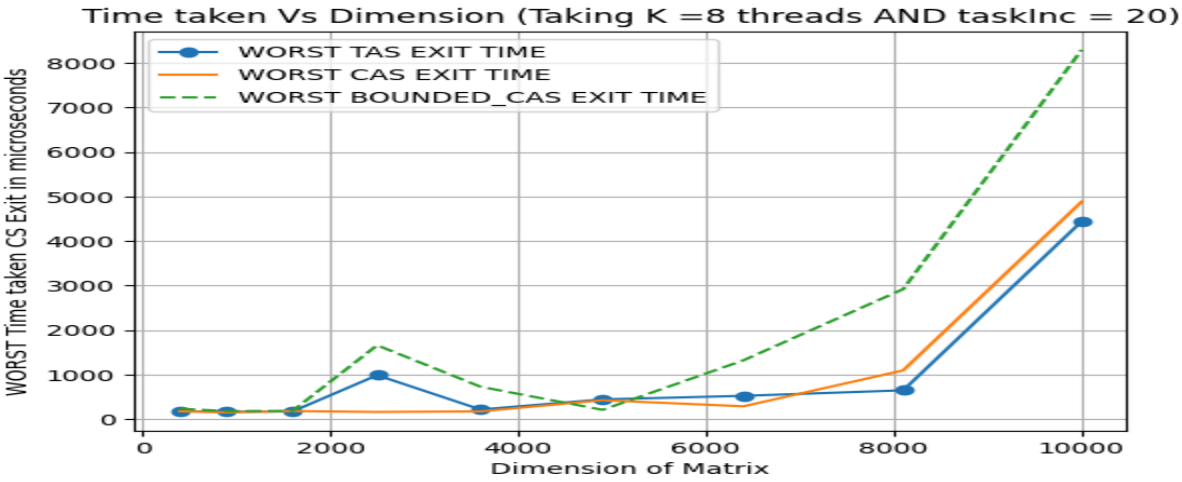
EXPERIMENT 1


Time taken Vs Dimension (Taking K =8 threads AND taskInc = 20)
1e6
TAS
CAS
BOUNDED_CAS
Sequential
Time taken in microseconds
Dimension of Matrix
0.0
0.2
0.4
0.6
0.8
1.0
0
2000
4000
6000
8000
10000


Time taken Vs Dimension (Taking K =8 threads AND taskInc = 20)
TAS_ENTER_TIME
CAS_ENTER_TIME
BOUNDED_CAS_ENTER_TIME
Time taken for CS in microseconds
Dimension of Matrix
0
100
200
300
400
0
2000
4000
6000
8000
10000


Time taken Vs Dimension (Taking K =8 threads AND taskInc = 20)
TAS EXIT TIME
CAS EXIT TIME
BOUNDED_CAS EXIT TIME
Time taken CS Exit in microseconds
Dimension of Matrix
40
35
30
25
20
15
10
0
2000
4000
6000
8000
10000


Time taken Vs Dimension (Taking K =8 threads AND taskInc = 20)
WORST TAS ENTRY TIME
WORST CAS ENTRY TIME
WORST BOUNDED_CAS ENTRY TIME
WORST Time taken CS ENTRY in microseconds
Dimension of Matrix
25000
20000
15000
10000
5000
0
0
2000
4000
6000
8000
10000


Time taken Vs Dimension (Taking K =8 threads AND taskInc = 20)
WORST TAS EXIT TIME
WORST CAS EXIT TIME
WORST BOUNDED_CAS EXIT TIME
WORST Time taken CS Exit in microseconds
Dimension of Matrix
8000
7000
6000
5000
4000
3000
2000
1000
0
0
2000
4000
6000
8000
10000

EXPERIMENT - 1(TIME VS SUDOKU SIZE) (FIXED: NUM OF THREADS = 8, taskInc = 20)										
S.No	X-Axis	TAS Avg CS Entry Time	CAS Avg CS Entry Time	Bounded CAS Avg CS Entry Time	TAS Avg CS Exit Time	CAS Avg CS Exit Time	Bounded CAS Avg CS Exit Time	CAS Total time taken	Sequential Total TIme	
1	20x20	54.8	57.63	81.47	36.488	32.39	42.11	5272.8	2502.2	
2	30x30	38.45	40.38	48.13	29.1	28.89	34.05	11158.2	9739.2	
3	40x40	31.73	27.41	31.28	23.576	25.78	27.61	20646.6	24136.3	
4	50x50	39.52	22.57	47.96	23.92	19.95	26.15	34460	57475	
5	60x60	17.54	17.87	26.01	18.12	17.74	21.01	50997.4	104221.2	
6	70x70	22.7	16.58	15.99	14.58	15.08	16.16	7600.2	190023.4	
7	80X80	12.58	10.47	25.62	12.14	11.4	14.46	106578.6	396819.2	
8	90x90	10.485	14.39	51.294	9.93	10.13	15.19	150242.4	705005.3	
9	100x100	52.9	44.39	439.6	15.988	11.88	37.29	249982.8	1127970	
S.No	X-Axis	TAS Worst case CS Entry Time	CAS Worst case CS Entry Time	Bounded CAS Worst case CS Entry Time	TAS Worst case CS Exit Time	CAS Worst case CS Exit Time	Bounded CAS Worst Case CS Exit Time	CAS Total time taken	Bounded CAS Total time taken	
1	20x20	298.2	417.4	484.8	175.6	175.6	235	6299.4	5485	
2	30x30	269.2	316.2	434	166.4	151.6	180.2	10945.6	11358.4	
3	40x40	357.2	263	296.4	172.6	181.6	184.4	20495	21044.7	
4	50x50	1468	316.2	1836.2	986.4	161	1659.4	33252.8	35090.8	
5	60x60	321.4	293.2	869.7	216	173.8	727.7	50664.8	51972.8	
6	70x70	1457.8	548.6	327.6	443.4	424.6	208.4	73936.4	73996.9	
7	80X80	617.5	375	2497.4	523.4	287.8	1321.2	105706.5	108903.6	
8	90x90	741.6	1264.8	3342	646.8	1094.6	2919.8	151048.2	164858.8	

9	100x100	5125	7278.2	28461.4	4442.6	4883.8	8295.3	233950.2	362869.2	

OBSERVATIONS1

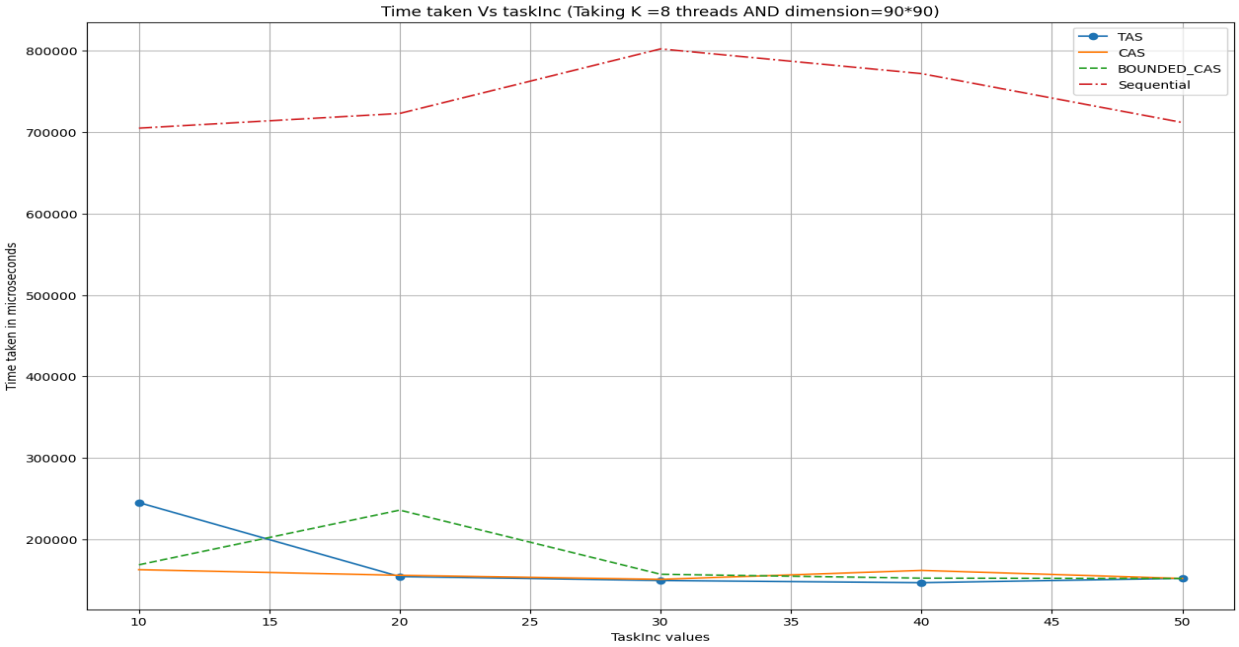
1. As size of sudoku /dimension increases, Sequential time is increasing exponentially whereas multithreaded and parallel methods using Locks time increases linearly which means multithreaded program taking less time for large dimensions , but near about or more in small dimensions because of overhead and creation time is more and become dominant factor for smaller sudoku's .

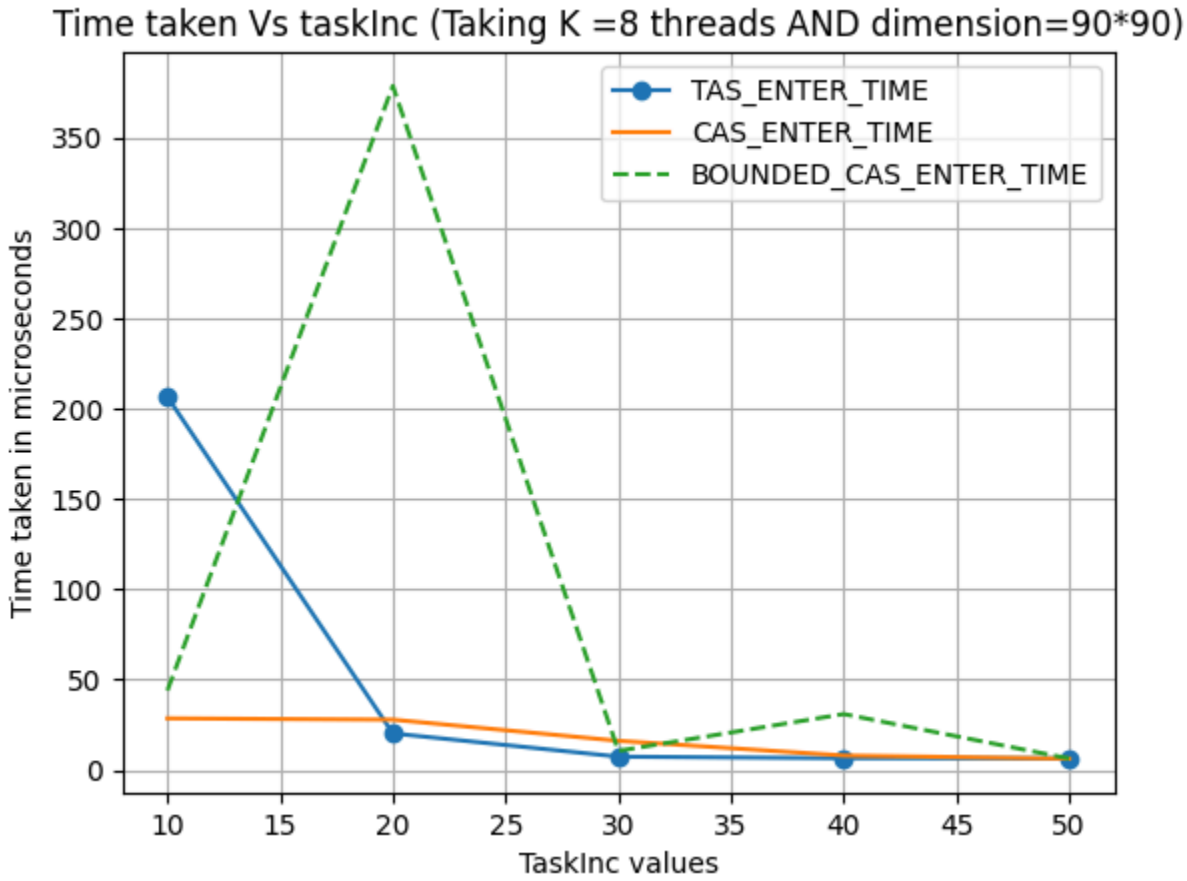
2. Among multithreaded methods , BOUNDED CAS is taking more time as size increases due to waiting array traversal (checking of array in cyclic order so each thread gets a chance to execute and there should be no starvation) .

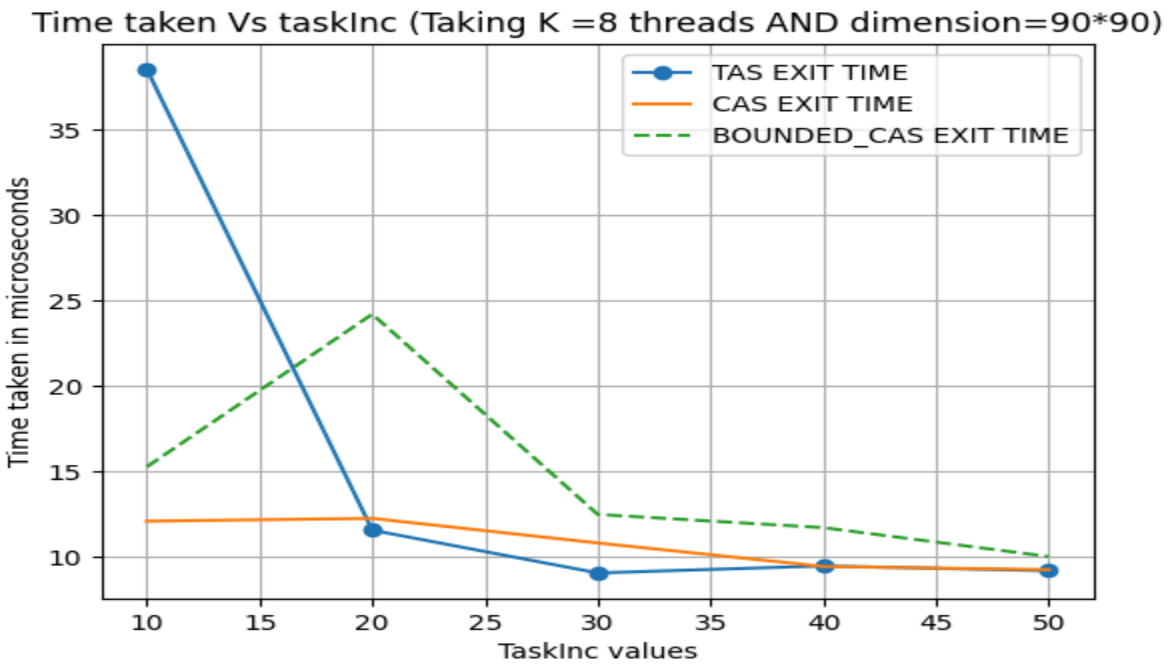
3. In TAS and CAS , TAS is little better for larger dimensions but for mid or less value both are performing almost the same because their internal application concept of Locking is the same.

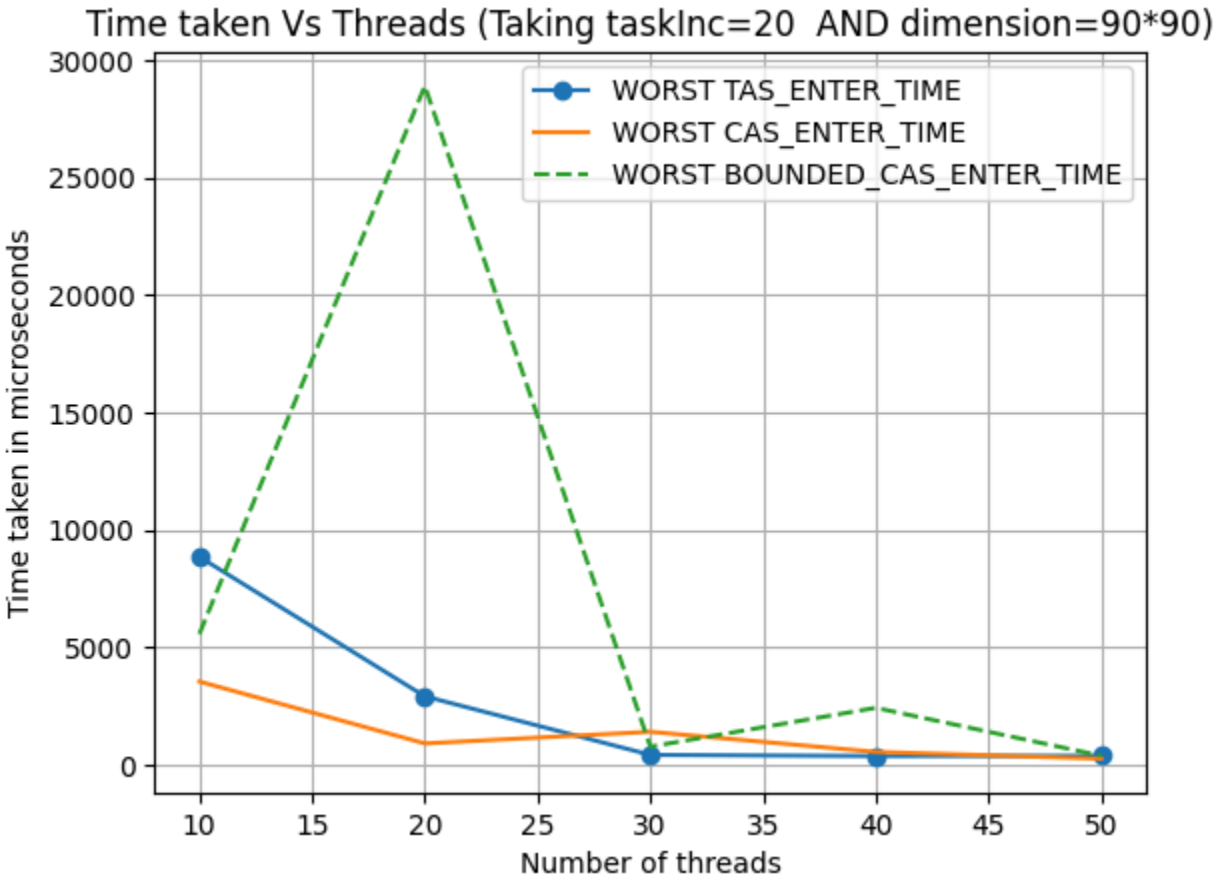
4. Average CS entry and exit time
   Till near 9000 dimensions of sudoku , parallel methods have a dip and after that it is increasing very fast due to overhead of lock .Similar pattern can be seen in Average Worst CS entry and exit time too.

EXPERIMENT 2


Time taken Vs taskInc (Taking K =8 threads AND dimension=90*90)
TAS
CAS
BOUNDED_CAS
Sequential
Time taken in microseconds
TaskInc values


Time taken Vs taskInc (Taking K =8 threads AND dimension=90*90)
TAS_ENTER_TIME
CAS_ENTER_TIME
BOUNDED_CAS_ENTER_TIME
Time taken in microseconds
TaskInc values


Time taken Vs taskInc (Taking K =8 threads AND dimension=90*90)
TAS EXIT TIME
CAS EXIT TIME
BOUNDED_CAS EXIT TIME
Time taken in microseconds
10
15
20
25
30
35
10
15
20
25
30
35
40
45
50
TaskInc values


Time taken Vs Threads (Taking taskInc=20 AND dimension=90*90)
WORST TAS_ENTER_TIME
WORST CAS_ENTER_TIME
WORST BOUNDED_CAS_ENTER_TIME
Time taken in microseconds
0
5000
10000
15000
20000
25000
30000
10
15
20
25
30
35
40
45
50
Number of threads

EXPERIMENT - 2(TIME VS TASK INCREMENT) (FIXED: NUM OF THREADS = 8 ,SUDOKU SIZE = 60*60)											
S.No	X-Axis	TAS Avg CS Entry Time	CAS Avg CS Entry Time	Bounded CAS Avg CS Entry Time	TAS Avg CS Exit Time	CAS Avg CS Exit Time	Bounded CAS Avg CS Exit Time	TAS Total time taken			
1	10	206.5	28.51	43.955	38.51	12.1	15.28	245185.2			
2	20	20.2	27.82	378.55	11.57	12.26	24.2	154371.4			
3	30	7.34	16.11	10.53	9.07	10.822	12.49	146876.2			
4	40	6.46	7.95	30.91	9.48	9.45	11.72	149876.2			
5	50	6.367	6.21	6.28	9.2	9.26	10.03	152173			

S.No	X-Axis	TAS Worst case CS Entry Time	CAS Worst case CS Entry Time	Bounded CAS Worst case CS Entry Time	TAS Worst case CS Exit Time	CAS Worst case CS Exit Time	Bounded CAS Worst Case CS Exit Time	CAS Total time taken	BOUNDED CAS Total time taken		
1	10	8843.8	3632.2	5562.8	5966.4	2827.4	4345.2	162845.9	168837.2		
2	20	2902	897.2	28898.6	1865.2	3364.3	3143.4	155948.4	235967		
3	30	409.6	1393.4	760	628.8	897.2	736.8	150963.3	157207.8		
4	40	360.4	536.8	2413.6	273.6	322	1146.8	161952	152424		
5	50	370.2	248	374.4	380	514.8	695.46	152047.7	152021.6		

OBSERVATION 2

1. As TasInc increases means each thread has to perform more task but will not affect sequential as Sequential don't use thread and we observe parallel methods are performing well wrt sequential . Because They are dividing tasks and executing them in different cores which enhance their speed.

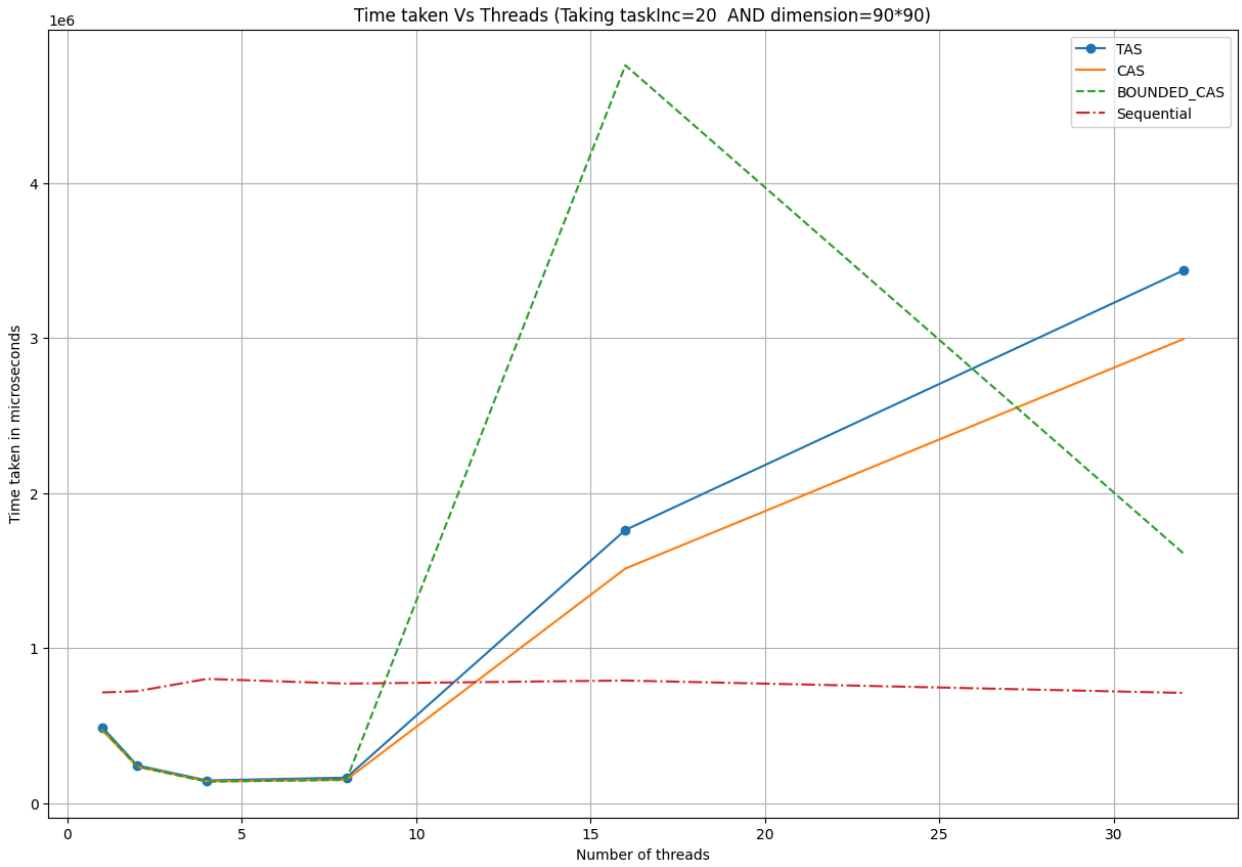
2. Bounded is performing little less effectively wrt TAS, CAS (they both are taking same time) maybe due to checking of waiting array and overhead problems.

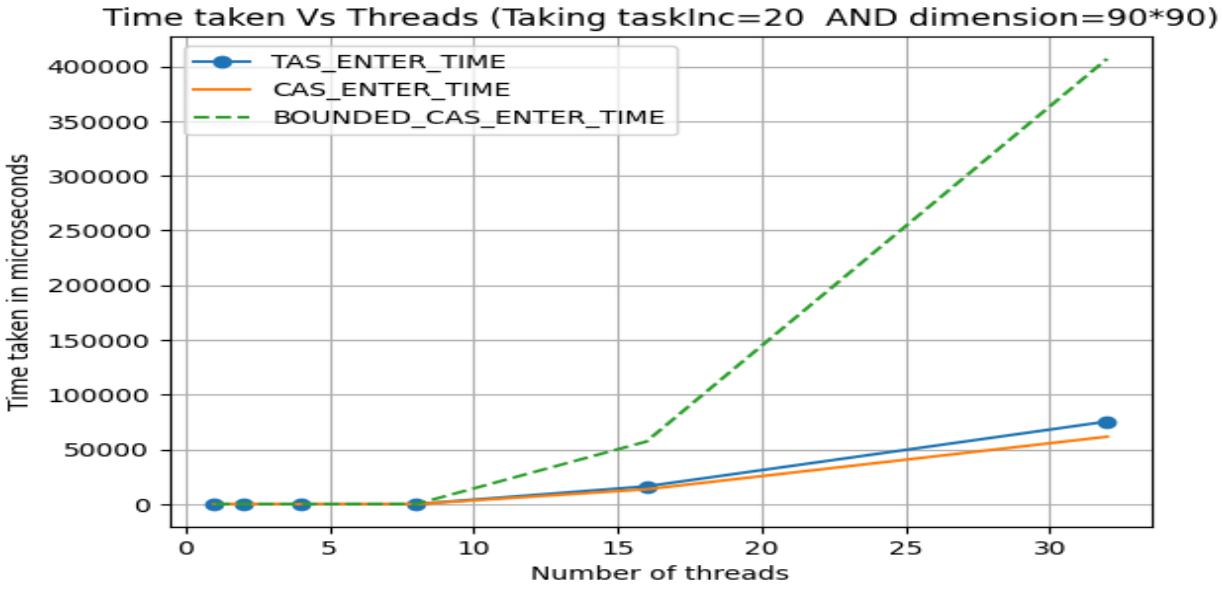
3. After taskInc>=30 , Average CS entry and exit time decreases because as effective lots of Task = (Total task / taskInc) . so, there are less lots and they are assigned to all threads more quickly .
   This might be the reason and similarly for Worst CS Entry and Exit time .Since, running them changes value frequently it’s quite tough to find a conclusion from it.

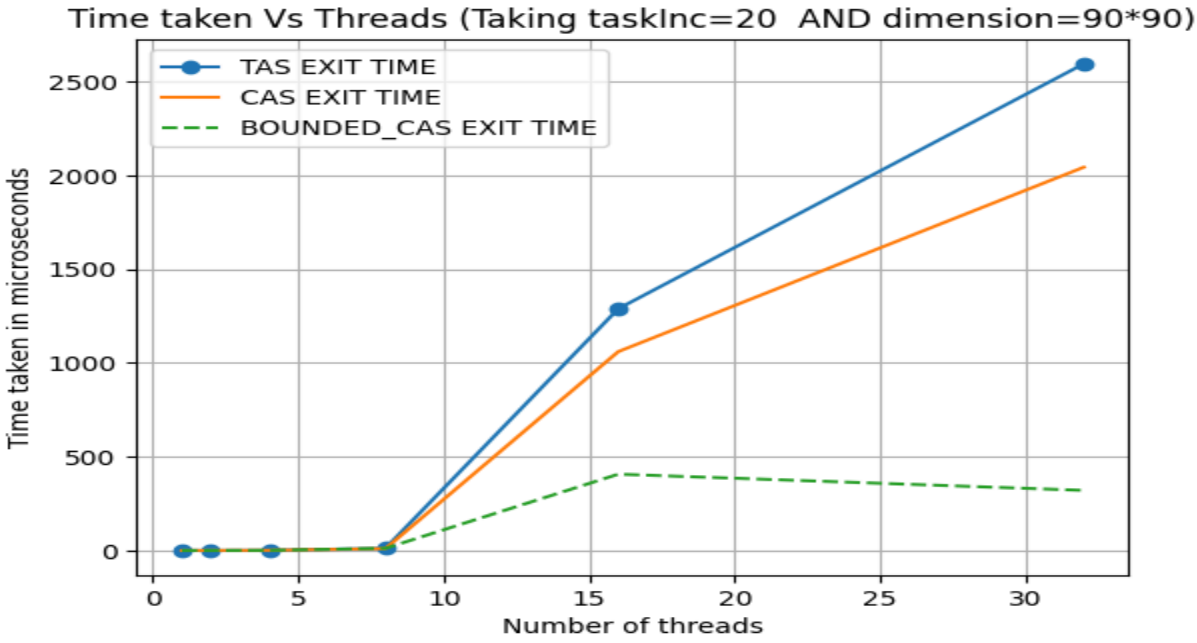
EXPERIMENT 3

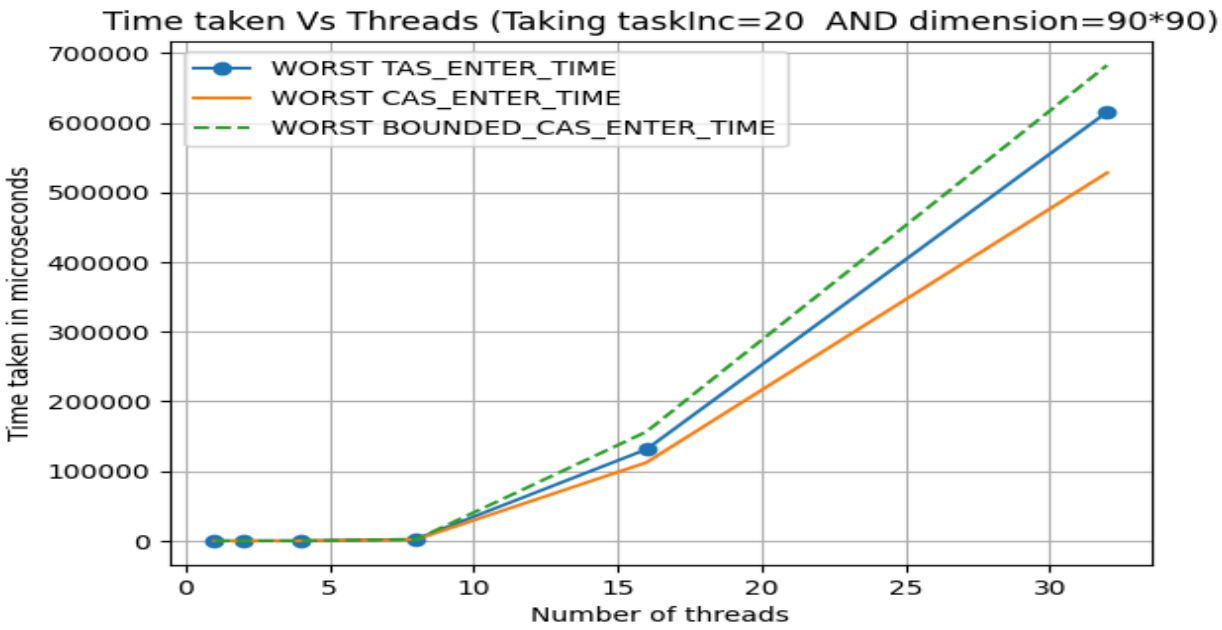
EXPERIMENT - 3(TIME VS NUMBER OF THREADS) (FIXED: SUDOKU SIZE = 90X90, taskInc = 20)										
S.No	X-Axis	TAS Avg CS Entry Time	CAS Avg CS Entry Time	Bounded CAS Avg CS Entry Time	TAS Avg CS Exit Time	CAS Avg CS Exit Time	Bounded CAS Avg CS Exit Time	TAS Total time taken		
1	1	0.709	0.071	0.064	0.149	0.122	0.125	490831.2		
2	2	0.109	0.113	0.136	0.2199	0.691	0.228	245050.6		
3	4	0.427	1.38	0.38	0.827	1.292	0.89	147750.8		
4	8	27.703	15.54	19.37	12.27	10.39	11.77	165424.4		
5	16	16238.37	13563.42	57188.4	1289.438	1060.68	407.07	1762454		
6	32	75480.26	61600.18	406863.4	2595.58	2044.18	320.87	3435349.4		
S.No	X-Axis	TAS Worst case CS Entry Time	CAS Worst case CS Entry Time	Bounded CAS Worst case CS Entry Time	TAS Worst case CS Exit Time	CAS Worst case CS Exit Time	Bounded CAS Worst Case CS Exit Time	CAS Total time taken	BOUNDED CAS Total time taken	
1	1	78.4	25.6	24.6	45.4	17	14.4	47101.4	482157.6	
2	2	21.4	25.6	30	15	20.23	15.2	237047	237762.2	
3	4	40.4	34	33.1	56.4	37.4	27.2	141982.8	139840.6	
4	8	2387.6	1826.4	1872.8	2140.4	1321.6	1795.4	152455.2	151237.6	
5	16	131255	112377.8	156529.4	28095.2	32858.6	30124	1513811.2	4757953.4	
6	32	615274.	528138.	682455.	58891.2	53753	48309.6	299284	1611660	

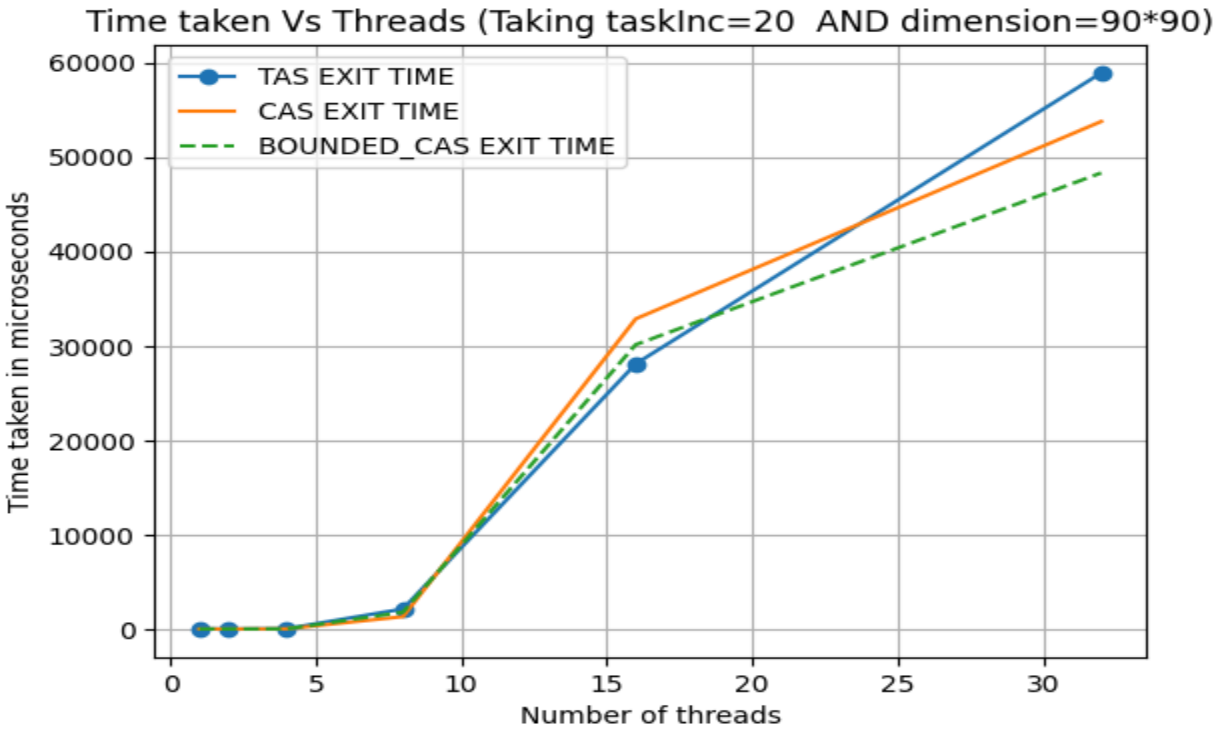
		6	2	6				4	.4	


Time taken Vs Threads (Taking taskInc=20 AND dimension=90*90)
1e6
TAS
CAS
BOUNDED_CAS
Sequential
Time taken in microseconds
Number of threads
0
1
2
3
4
0
5
10
15
20
25
30


Time taken Vs Threads (Taking taskInc=20 AND dimension=90*90)
TAS_ENTER_TIME
CAS_ENTER_TIME
BOUNDED_CAS_ENTER_TIME
Time taken in microseconds
Number of threads
0
50000
100000
150000
200000
250000
300000
350000
400000
0
5
10
15
20
25
30


Time taken Vs Threads (Taking taskInc=20 AND dimension=90*90)
TAS EXIT TIME
CAS EXIT TIME
BOUNDED_CAS EXIT TIME
Time taken in microseconds
Number of threads
0
500
1000
1500
2000
2500
0
5
10
15
20
25
30





OBSERVATION 3

1. For the number of threads near or less than 10 our multithreaded programs are working well , better than sequentially but after some threshold number of thread value near 10 execution time of parallel methods increases massively due to scheduling overheads , lock contention, and system core boundation .

2. Average CS entry and exit time also increases very fast as Number of threads increases because of the Spinlock condition in the inside implementation of LOCK Methods means more threads are spinning again and again to acquire locks increasing CPU utilization, wasting CPU cycles too .

3. Worst CS entry exit time also increases with the number of threads due to above same reasons , all Average and Worst have one peak point at 16 threads , after that may be due to system core, too much contention and CPU cycles make it a poor performer.
   Bounded CAS is also using the CAS algorithm for lock and cycling check for thread , that's why it is also struggling and performing badly.