Report

Experiment - 1) **Time vs. Sudoku Size (Fixed Threads)**

Note : All times are in microseconds, taskInc = 20, Num of threads = 8

X - axis	TAS total time	CAS total time	Bounded CAS total time	Sequential total time
20^2	38636.2	37775.0	36808.6	32837.4
30^2	118156.2	103986.6	103531.6	98167.60
40^2	327116.2	306750.2	286917.4	231292.8
50^2	741147.2	678890.6	694265.4	551037.4
60^2	1467418.0	1476622.0	1548680.0	937410.4

- We can observe that, as size is increasing, the total time taken by any method increases as now any thread has to check for many rows, columns and grids.
- Sequential is faster as there is no waiting for threads to synchronise.
- Time taken for diff methods by fixing size is :
  Sequential < Bounded CAS **<** CAS < TAS (when size is small)
- For larger sizes, CAS and Bounded CAS perform better.
- Among TAS, CAS, Bounded CAS; bounded cas is performing well at larger sudoku size as it is able to check for next waiting thread from waiting array and give chance for it.

X - axis	TAS		CAS		Bounded CAS	
	Avg Entry Time	Avg Exit Time	Avg Entry Time	Avg Exit Time	Avg Entry Time	Avg Exit Time
20^2	13.38292	6.8954	13.2535	5.9193	19.19824	6.634226
30^2	13.1245	5.2526	9.950884	4.410552	12.31884	5.608392
40^2	10.4513	4.855362	11.03278	4.464338	10.271414	4.62017
50^2	11.037184	4.516678	11.872166	4.031088	10.604251	6.64529
60^2	10.069506	4.20066	9.772166	5.210928	13.78436	7.337308

- Average entry time is larger than average exit time in every case due to the fact that, in exiting from the critical section the thread has to just modify the isLocked variable to false. But while entering, threads have to wait till another thread exits from the critical section.
- Entry and exit times are decreasing as sudoku size is increased.As the work is more i.e, more rows, columns, grids to check for, threads stay busy, so there aren't any sitting idle.
- Since the bounded cas has to check for next thread that is waiting to enter into CS, it's exiting time is more than other methods.
- The reason for CAS and TAS exit times are similar is due to the same exit section code, i.e., it just has to set isLocked variable to false.

X - axis	TAS		CAS		Bounded CAS	
	Worst-Case Entry Time	Worst-Case Exit Time	Worst-Case Entry Time	Worst-Case Exit Time	Worst-Case Entry Time	Worst-Case Exit Time
20^2	117.6	57.4	88.4	41.4	194.2	45.4
30^2	134.2	65.8	95.2	90.4	151.2	78.6
40^2	178.0	156.4	224.6	104.8	236.4	122.4
50^2	284.8	198.8	276.6	222.2	226.4	298.6
60^2	515.4	279.4	406.4	370.2	591.8	597.0

- There is a sharp increase in time as sudoku size increased for TAS, CAS compared to Bounded CAS.
- As the size increases, the worst case time increases.
- We can observe that in overall time taken, sequential takes less time because there is no overhead in creation of threads and waiting for that shared variable that allots the work that thread has to do.

Experiment - 2) **Time vs. Task Increment (Fixed Sudoku Size)**

Sequential Time Taken for 8100 sized sudoku = 1901930 micro sec

X - axis	TAS total time	CAS total time	Bounded CAS total time
10	2502000.0	2538480.0	2527970.0

20	2562804.0	2529194.0	2509044.0
30	2563296.0	2597348.0	2594010.0
40	2673178.0	2644700.0	2638340.0
50	2730980.0	2731930.0	2710546.0

X - axis	**TAS**		**CAS**		**Bounded CAS**	
	Avg Entry Time	**Avg Exit Time**	**Avg Entry Time**	**Avg Exit Time**	**Avg Entry Time**	**Avg Exit Time**
10	3.78	1.79	4.05	1.62	4.02	2.42
20	3.69	1.89	3.80	2.12	4.77	2.22
30	4.76	2.38	4.80	1.54	5.58	2.45
40	5.04	2.28	4.47	2.02	5.16	1.90
50	3.96	1.82	4.39	2.14	4.46	3.01

X - axis	**TAS**		**CAS**		**Bounded CAS**	
	Worst-Case Entry Time	**Worst-Case Exit Time**	**Worst-Case Entry Time**	**Worst-Case Exit Time**	**Worst-Case Entry Time**	**Worst-Case Exit Time**
10	496.8	417.8	703.2	303.4	519.2	452.4
20	373.6	252.4	266.2	367.6	438.4	290.8
30	341.6	383.6	513.9	249.4	370.2	273.2
40	427.6	225.4	303.2	170.0	474.6	137.6
50	389.4	145.8	332.4	258.2	252.0	333.2

Analysis:

- Since, sudoku size is not changing, sequential time taken would be constant.
- We can observe that in overall time taken, sequential takes less time because there is no overhead in creation of threads and waiting for that shared variable that allots the work that thread has to do.
- As tasInc is increased, the total time taken is slightly increasing for any type of mutual exclusion method.

- We can observe that TAS total time taken is slightly better than CAS. Bounded - CAS is performing better than CAS, as it is directly checking and giving a chance for the next waiting thread than doing unlocking i.e., setting isLocked to false. Also, it restricts the other threads to wait for a long time.
- TAS is performing better than CAS, Bounded-CAS in both average and worst cases time taken to enter and exit CS.
- It is obvious that entry time is more than exit time, as in case of exiting the only job to be done is to set isLocked to false but while entering it has to wait until the other thread leaves.

Experiment - 3) **Time vs. Number of Threads (Fixed Sudoku Size)**

Sequential Time Taken for 8100 sized sudoku = 1901930 micro sec

X - axis	TAS total time	CAS total time	Bounded CAS total time
1	1833436.0	1789054.0	1836838.0
2	1761678.0	1768454.0	1768904.0
4	1921050.0	1900958.0	1933510.0
8	2573356.0	2505006.0	2523670.0
16	4023522.0	3971926.0	3998230.0
32	4939460.0	4903728.0	9880598.0

X - axis	TAS		CAS		Bounded CAS	
	Avg Entry Time	Avg Exit Time	Avg Entry Time	Avg Exit Time	Avg Entry Time	Avg Exit Time
1	0.25	0.16	0.27	0.15	0.30	0.16
2	0.59	0.36	0.57	0.32	0.61	0.38
4	1.40	0.60	1.31	0.64	1.45	0.83
8	3.55	1.83	3.76	1.55	3.72	2.30
16	14.92	6.26	14.36	6.58	18.2	9.29

32	1336.75	18.07	1352.57	45.72	147909.94	29.36

X - axis	TAS		CAS		Bounded CAS	
	Worst-Case Entry Time	Worst-Case Exit Time	Worst-Case Entry Time	Worst-Case Exit Time	Worst-Case Entry Time	Worst-Case Exit Time
1	2.8	1.2	4	1	2.8	1
2	82.8	87.4	82.8	73.2	80.4	72.6
4	103.8	107.6	105.6	106.0	102.4	123.4
8	427.4	215.8	303.6	232.0	300.0	275.0
16	1444.4	651.2	1320.6	895.2	3553.6	1322.6
32	135230.0	12189.2	125375.2	30266.6	649359.4	13759.8

Analysis:

- Since, sudoku size is not changing, sequential time taken would be constant.
- We can observe that in overall time taken, sequential takes less time because there is no overhead in creation of threads and waiting for that shared variable that allots the work that thread has to do.
- As the number of threads increased, total time taken for different methods also increased which should actually not happen. The reason for this is because of thread creation overhead, also the time for threads in waiting for their work, due to busy waiting of the threads.
- We can observe that time taken for bounded CAS is less compared to TAS and is correct as the next thread which is waiting can be immediately found by seeing the waiting array.
- As the threads are increasing, average time taken for entry and exit are also increasing irrespective of the mutual exclusion method. Because now many threads are competing with each other for the same shared variable which is accessed by only one thread at a time.
- As threads count increases, worst case entry and exit time is increased, this is most likely because of that thread that reached late and had to wait till all other threads utilise that shared variable.
- The reason for fluctuations in values occur in any case are because there are many other programs that run concurrently in our device affecting our program times.

Implementation:

- This program is to validate sudoku using multiple threads which uses a shared variable named ‘C’ that allots the task that thread has to do. Since it is a shared variable to avoid race conditions, they have to be synchronised. So, when one thread updates ‘C’, other threads have to wait which is called mutual exclusion and is done by TAS, CAS, and Bounded CAS methods which are actually hardware instructions but implemented here by cpp atomic library.
- User implements chooses a different method by command line interface i.e., while executing “./a.out <lockType>” where lockType is a digit which specifies different mutual exclusion methods like 0 for TAS, 1 for CAS, 2 for Bounded CAS, 3 for sequential.
- TAS is implemented by an **exchange** method from the atomic library.

```
{
    while (isLocked.exchange(true, memory_order_acquire))
        ;
```

  - Initially, **isLocked** variable is false and the first thread that requests CS acquires it and makes it true resulting in no other thread able to acquire till that thread unlocks it by making the value isLocked to false, once it finishes its work in critical section.

```
isLocked.store(false, memory_order_release);
```

  - Since now again the lock is set free(false), other threads can use it.

- CAS is implemented by a **compare_exchange_strong** method from the atomic library.

```
bool expected = false;
while (!isLocked.compare_exchange_strong(expected, true,
memory_order_acquire))
{
    expected = false;
}
```

  - If isLocked is false, it sets to true and comes out this while loop(since function returns true as isLocked is equal to expected)
  - This function returns false when isLocked(true) is different from expected(false) and sets expected to true, so that's why expected

should be changed back to false inside the while loop.

```
isLocked.store(false, memory_order_release);
```

- Once the thread finishes its work, it is set back to false.

- Bounded - CAS is implemented by a **compare_exchange_strong** method from the atomic library.

```
atomic<bool> isLocked{false};
vector<atomic<bool>> waiting = vector<atomic<bool>>(K);
```

```
waiting[idNo].store(true);
bool expected = false;
while (waiting[idNo].load())
{
    if (isLocked.compare_exchange_strong(expected, true,
    memory_order_acquire))
    {
        break;
    }
    expected = false;
}
waiting[idNo].store(false);
```

  - It uses a waiting array, which stores true when a thread makes a request to enter CS. Initially this array is set to false.
  - If isLocked is false, it sets to true and comes out this while loop(since function returns true as isLocked is equal to expected).
  - This function returns false when isLocked(true) is different from expected(false) and sets expected to true, so that's why expected should be changed back to false inside the while loop.
  - Once the thread finishes its work, it checks for the next thread after it according to the thread number that is waiting by checking in the waiting array. If there is no thread waiting by the time it has checked (by traversing the whole waiting array once), it sets the isLocked to false and comes out of the CS and executes the reminder section.

```
int j = (idNo + 1) % K;
while (j != idNo && !waiting[j].load())
{
    j = (j + 1) % K;
}
if (j == idNo)
{
    isLocked.store(false, memory_order_release);
}
else
{
    waiting[j].store(false);
}
```

- **isSudokuValid** is a global atomic variable which is initially set to true. When a thread is checking any row, column or grid and finds out that it is invalid, it sets it to false. For the program to terminate early, it has to check that variable which is done by :

```
isSudokuValid.store(false); // sudoku invalid
```

```
if (!isSudokuValid.load())
{ // Early termination check
    pthread_exit(res);
}
```