Validating Sudoku Solution - 2

Bhumin Hirpara - CS23BTECH11009

March 2, 2025

Introduction

This assignment implements a dynamic multithreaded Sudoku validator using three mutual exclusion algorithms: Test-and-Set (TAS), Compare-and-Swap (CAS), and Bounded CAS. Unlike static task allocation in previous work, threads dynamically claim work units through a shared counter while maintaining validation correctness. We analyze performance characteristics through three experiments comparing synchronization mechanisms under different problem parameters.

Implementation Overview

Sequential

Code Overview

This file implements a **sequential Sudoku validator** that checks:

- Row validity
- Column validity
- Sub-grid validity

without multithreading. Below are the key design decisions and code flow.

Key Functions

- `checkRows()`:
  - Uses a boolean `seen[]` array to detect duplicates
  - Validates all rows sequentially
  - Writes row-wise results to output file
- `checkColumns()`:
  - Mirrors `checkRows()` logic for columns
  - Checks uniqueness in column-major order
- `checkSubGrids()`:
  - Divides grid into $\sqrt{N} \times \sqrt{N}$ sub-grids
  - Uses nested loops to validate each sub-grid
- `readFile()`:
  - Reads `inp.txt` with dynamic memory allocation
  - Stores Sudoku in 2D array `sudoku[][]`

Design Decisions

- **Validation Strategy**:
  - Uses three independent checks (rows/columns/subgrids) for modularity
  - Combines results with logical AND: `res = checkRows() && checkColumns() && checkSubGrids()`
- **Duplicate Detection**:
  - Boolean `seen[]` array tracks encountered values (1–N)
  - Space complexity: $O(N)$ per row/column/subgrid
- **I/O Handling**:
  - Writes incremental results to `output.txt` during validation
  - Uses `fprintf` for thread-safe file operations (future-proofing)
- **Performance Measurement**:
  - Uses `gettimeofday()` for microsecond-precision timing
  - Reports total execution time in output file

Code Flow

1. **Initialization**:
   - `readFile()` parses `input.txt` into `sudoku[][]`
   - Allocates memory dynamically using `malloc()`
2. **Validation Phase**:
   - `checkRows()` → `checkColumns()` → `checkSubGrids()`
   - Each function writes partial results to `output.txt`
3. **Result Consolidation**:
   - Combines results with logical AND to determine overall validity
   - Writes final verdict ("valid"/"invalid") and timing data

TAS

Code Overview

This file implements a multithreaded Sudoku validator using **Test-and-Set (TAS)** for dynamic task allocation:

- Threads dynamically claim chunks of tasks (rows/columns/subgrids) via a shared counter `C` guarded by a TAS lock.
- Early termination via a global validity flag (`isValid`) if any thread detects an invalid section.
- Detailed logging of thread actions, critical section (CS) entry/exit times, and performance metrics.

Key Components

1. Data Structures

- `Data`: Stores thread ID for parameter passing.
- `Log`: Captures timestamps and messages for thread-safe logging.
- **Vectors**:
  - `logs`: Per-thread log entries to avoid race conditions.
  - `inTimes/exTimes`: Track CS entry/exit times (sum and max per thread).

2. Core Validation Logic

- `checkRow(index)`, `checkColumn(index)`, `checkSubGrid(index)`:
  - Validate uniqueness in rows, columns, and subgrids using boolean tracking arrays.
  - Return `false` on duplicate detection.

3. Task Processing

- `processTask(oldC, newC, threadId)`:
  - Validates tasks (rows/columns/subgrids) assigned via `oldC` and `newC` range.
  - Logs validation results and updates `isValid` for early termination.

4. Critical Section Management

- `getTask()`:
  - Uses `atomic_flag Lock` (TAS) to protect increments of the shared counter `C`.
  - Measures CS entry/exit times and logs task acquisition (e.g., "grabs row 5").

5. Logging & Timing Utilities

- `timeToString()`: Converts timestamps to `HH:MM:SS.microseconds` format.
- Logs are merged, sorted by timestamp, and written to `output.txt`.

Design Decisions

1. Dynamic Task Allocation

- **Shared Counter (`C`)**: Threads claim `taskInc` tasks at once by atomically incrementing `C`.
- **TAS Lock**: Ensures mutual exclusion during counter updates using `atomic_flag` (spinlock).
- **Task Division**:
  - Tasks 0–`n-1`: Rows
  - Tasks `n`–`2n-1`: Columns
  - Tasks `2n`–`3n-1`: Subgrids

2. Early Termination

- Global `isValid` flag: Set to `false` by any thread detecting invalidity.
- Threads exit loops early if `!isValid`, reducing unnecessary work.

3. **Logging Mechanism**
   - **Per-Thread Logs**: Avoids contention by storing logs in thread-specific vectors (`logs[i]`).
   - **Timestamp Sorting**: Merged logs are sorted chronologically for coherent output.
4. **Performance Metrics**
   - **Microsecond Timing**: `system_clock` tracks CS entry/exit and total execution time.
   - **Averages/Worst-Case**: Computed from per-thread time aggregates in `inTimes` and `exTimes`.

Code Flow

1. **Initialization**
   - `readFile()`: Reads `input.txt`, initializes `sudoku` grid, and resizes log/time vectors.
2. **Thread Execution**
   - **Thread Creation**: `k` threads execute `getTask()`, competing for `C` via TAS.
   - **Critical Section**:
     - Threads spin on `Lock.test_and_set()` until acquiring the lock.
     - Claim `taskInc` tasks by updating `C`, log grabbed tasks (e.g., "grabs row 3").
     - Release lock and validate claimed tasks in `processTask()`.
3. **Validation & Early Exit**
   - Invalid sections set `isValid = false`, causing all threads to terminate.
   - Results from all threads are combined via logical AND (`isValid`).
4. **Result Aggregation**
   - **Log Merging**: Sorts logs by timestamp and writes to `output.txt`.
   - **Metrics Calculation**:
     - Total time: `eTime - sTime`
     - Average/worst-case CS entry/exit times derived from `inTimes` and `exTimes`.

CAS

Code Overview

This file implements a multithreaded Sudoku validator using **Compare-and-Swap (CAS)** for dynamic task allocation:

- Threads dynamically claim chunks of tasks (rows/columns/subgrids) via a shared counter `C` guarded by a CAS-based lock.
- Early termination via a global validity flag (`isValid`) if any thread detects an invalid section.
- Detailed logging of thread actions, critical section (CS) entry/exit times, and performance metrics.

Key Components

1. Data Structures

- `Data`: Stores thread ID for parameter passing.
- `Log`: Captures timestamps and messages for thread-safe logging.
- **Vectors**:
  - `logs`: Per-thread log entries to avoid race conditions.
  - `inTimes/exTimes`: Track CS entry/exit times (sum and max per thread).

2. Core Validation Logic

- `checkRow(index)`, `checkColumn(index)`, `checkSubGrid(index)`:
  - Validate uniqueness in rows, columns, and subgrids using boolean tracking arrays.
  - Return `false` on duplicate detection.

3. Task Processing

- `processTask(oldC, newC, threadId)`:
  - Validates tasks (rows/columns/subgrids) assigned via `oldC` and `newC` range.
  - Logs validation results and updates `isValid` for early termination.

4. Critical Section Management

- `getTask()`:
  - Uses `atomic<bool> Lock` (CAS) to protect increments of the shared counter `C`.
  - Measures CS entry/exit times and logs task acquisition (e.g., "grabs row 5").

5. Logging & Timing Utilities

- `timeToString()`: Converts timestamps to `HH:MM:SS.microseconds` format.
- Logs are merged, sorted by timestamp, and written to `output.txt`.

Design Decisions

1. Dynamic Task Allocation

- **Shared Counter (`C`)**: Threads claim `taskInc` tasks at once by atomically incrementing `C`.
- **CAS Lock**: Ensures mutual exclusion during counter updates using `atomic<bool>`.
- **Task Division**:
  - Tasks 0–`n-1`: Rows
  - Tasks `n`–`2n-1`: Columns
  - Tasks `2n`–`3n-1`: Subgrids

2. Early Termination

- Global `isValid` flag: Set to `false` by any thread detecting invalidity.
- Threads exit loops early if `!isValid`, reducing unnecessary work.

3. **Logging Mechanism**
   - **Per-Thread Logs**: Avoids contention by storing logs in thread-specific vectors (`logs[i]`).
   - **Timestamp Sorting**: Merged logs are sorted chronologically for coherent output.
4. **Performance Metrics**
   - **Microsecond Timing**: `system_clock` tracks CS entry/exit and total execution time.
   - **Averages/Worst-Case**: Computed from per-thread time aggregates in `inTimes` and `exTimes`.

Code Flow

1. **Initialization**
   - `readFile()`: Reads `input.txt`, initializes `sudoku` grid, and resizes log/time vectors.
2. **Thread Execution**
   - **Thread Creation**: `k` threads execute `getTask()`, competing for `C` via CAS.
   - **Critical Section**:
     - Threads spin on `Lock.compare_exchange_strong()` until acquiring the lock.
     - Claim `taskInc` tasks by updating `C`, log grabbed tasks (e.g., "grabs row 3").
     - Release lock and validate claimed tasks in `processTask()`.
3. **Validation & Early Exit**
   - Invalid sections set `isValid = false`, causing all threads to terminate.
   - Results from all threads are combined via logical AND (`isValid`).
4. **Result Aggregation**
   - **Log Merging**: Sorts logs by timestamp and writes to `output.txt`.
   - **Metrics Calculation**:
     - Total time: `eTime - sTime`
     - Average/worst-case CS entry/exit times derived from `inTimes` and `exTimes`.

Key Differences from TAS Approach

- **CAS vs. TAS**: CAS uses `atomic<bool>` for lock acquisition, reducing contention compared to TAS.
- **Lock Release**: CAS uses `store(false)` instead of `clear()`.
- **Efficiency**: CAS is generally more efficient in high-contention scenarios.

Bounded CAS

Code Overview

This file implements a multithreaded Sudoku validator using **Bounded Compare-and-Swap (CAS)** for dynamic task allocation:

- Threads dynamically claim chunks of tasks (rows/columns/subgrids) via a shared counter `C` guarded by a Bounded CAS mechanism.
- Early termination via a global validity flag (`isValid`) if any thread detects an invalid section.
- Detailed logging of thread actions, critical section (CS) entry/exit times, and performance metrics.

Key Components

1. Data Structures

- `Data`: Stores thread ID for parameter passing.
- `Log`: Captures timestamps and messages for thread-safe logging.
- **Vectors**:
  - `logs`: Per-thread log entries to avoid race conditions.
  - `inTimes/exTimes`: Track CS entry/exit times (sum and max per thread).
  - `waiting`: Tracks threads waiting to enter the critical section.
- `atomic<int> current`: Tracks the thread currently allowed to enter the critical section.

2. Core Validation Logic

- `checkRow(index)`, `checkColumn(index)`, `checkSubGrid(index)`:
  - Validate uniqueness in rows, columns, and subgrids using boolean tracking arrays.
  - Return `false` on duplicate detection.

3. Task Processing

- `processTask(oldC, newC, threadId)`:
  - Validates tasks (rows/columns/subgrids) assigned via `oldC` and `newC` range.
  - Logs validation results and updates `isValid` for early termination.

4. Critical Section Management

- `getTask()`:
  - Uses `atomic<int> current` and `waiting` array to implement Bounded CAS.
  - Threads spin until they are granted access to the critical section.
  - Measures CS entry/exit times and logs task acquisition (e.g., "grabs row 5").

5. Logging & Timing Utilities

- `timeToString()`: Converts timestamps to `HH:MM:SS.microseconds` format.
- Logs are merged, sorted by timestamp, and written to `output.txt`.

Design Decisions

1. Bounded CAS Mechanism

- **Shared Counter (`C`)**: Threads claim `taskInc` tasks at once by atomically incrementing `C`.
- `atomic<int> current`: Tracks the thread currently allowed to enter the critical section.
- `waiting` array: Tracks threads waiting to enter the critical section.
- **Task Division**:
  - Tasks 0–`n-1`: Rows
  - Tasks `n`–`2n-1`: Columns
  - Tasks `2n`–`3n-1`: Subgrids

2. Early Termination

- Global `isValid` flag: Set to `false` by any thread detecting invalidity.
- Threads exit loops early if `!isValid`, reducing unnecessary work.

3. Logging Mechanism

- **Per-Thread Logs**: Avoids contention by storing logs in thread-specific vectors (`logs[i]`).
- **Timestamp Sorting**: Merged logs are sorted chronologically for coherent output.

4. Performance Metrics

- **Microsecond Timing**: `system_clock` tracks CS entry/exit and total execution time.
- **Averages/Worst-Case**: Computed from per-thread time aggregates in `inTimes` and `exTimes`.

Code Flow

1. Initialization

- `readFile()`: Reads `input.txt`, initializes `sudoku` grid, and resizes log/time vectors.

2. Thread Execution

- **Thread Creation**: `k` threads execute `getTask()`, competing for `C` via Bounded CAS.
- **Critical Section**:
  - Threads spin on `current.compare_exchange_strong()` until granted access.
  - Claim `taskInc` tasks by updating `C`, log grabbed tasks (e.g., "grabs row 3").
  - Release lock and validate claimed tasks in `processTask()`.

3. Validation & Early Exit

- Invalid sections set `isValid = false`, causing all threads to terminate.
- Results from all threads are combined via logical AND (`isValid`).

4. Result Aggregation

- **Log Merging**: Sorts logs by timestamp and writes to `output.txt`.
- **Metrics Calculation**:
  - Total time: `eTime - sTime`
  - Average/worst-case CS entry/exit times derived from `inTimes` and `exTimes`.

Key Differences from CAS Approach

- **Bounded CAS vs. CAS**: Bounded CAS uses a waiting array and explicit thread handoff, reducing contention.
- **Thread Handoff**: The current thread explicitly passes the lock to the next waiting thread.
- **Efficiency**: Bounded CAS reduces unnecessary spinning, improving performance in high-contention scenarios.

Extra

Code Overview

This file extends `chunk.c` to include **early termination**, where:

- Threads stop processing as soon as the Sudoku is found invalid
- Global result variable (`res`) is checked periodically
- Reduces unnecessary computation for invalid Sudokus

Key Components

- `Data Struct`:
  - Stores thread ID and starting index for chunk processing
  - Enables parameter passing to thread functions
- **Validation Functions**:
  - `checkRow()`/`checkColumn()`/`checkSubGrid()`: Core validation logic
  - Use boolean arrays to detect duplicates (1–N)
- `checkRows()`/`checkColumns()`/`checkSubGrids()`:
  - Thread entry points processing assigned chunks
  - Periodically check `res` for early termination
  - Write results directly to output file
- `readFile()`:
  - Reads `inp.txt` with dynamic memory allocation
  - Stores Sudoku in 2D array `sudoku[][]`

Design Decisions

- **Early Termination**:
  - Threads check `res` before processing each chunk
  - If `res` is false, threads exit immediately
  - Reduces redundant computation for invalid Sudokus
- **Thread Grouping**:
  - Total threads split equally between rows/columns/subgrids
  - `k -= k%3` ensures equal distribution
  - Example: 12 threads → 4 per task type
- **Chunk Allocation**:
  - `chunkSize = n/(k/3)` calculates work per thread
  - Handles remainder with `if(n%(k/3)!=0) chunkSize++`
  - Threads process indices `[index, index+chunkSize)`
- **I/O Handling**:
  - Writes incremental results to `output.txt` during validation
  - Uses `fprintf` for thread-safe file operations (future-proofing)
- **Performance Measurement**:
  - Uses `gettimeofday()` for microsecond-precision timing
  - Reports total execution time in output file

Code Flow

1. **Initialization**:
   - `readFile()` loads Sudoku grid
   - Thread count adjusted to multiple of 3
2. **Thread Creation**:
   - Three thread batches for rows/columns/subgrids
   - Each thread receives starting index via `Data` struct
3. **Parallel Validation**:
   - Threads process assigned chunks independently
   - Periodically check `res` for early termination
   - Results written immediately to output file
4. **Result Aggregation**:
   - Global `res` combines results via logical AND
   - Timing data appended to output

1 Experiment Results

1.1 Experiment 1: Time vs. Sudoku Size (Fixed Threads)

- **Average Entry Time Comparison**

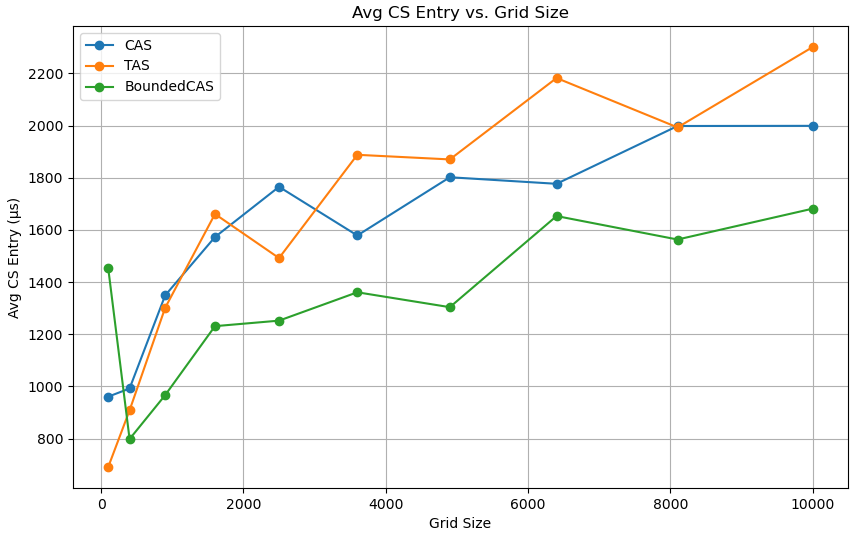

Figure 1: Average CS entry time vs Sudoku size (8 threads)

Observations: TAS has a higher entry time due to contention. CAS would follow linear growth, while Bounded CAS has a lower gradient due to cyclic CS accesses. Sequential has no entry time.

- **Average Exit Time Comparison**

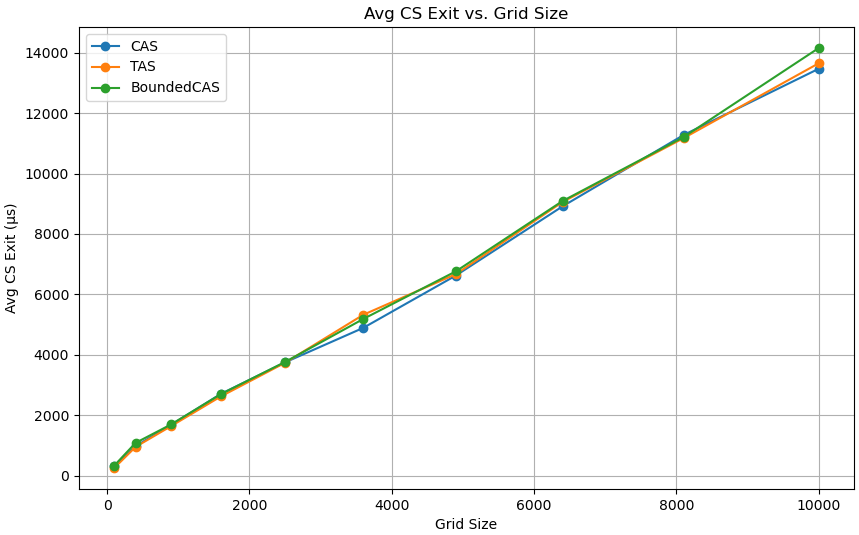

Figure 2: Average CS exit time vs Sudoku size

Observations: All methods haves stable exit times. No significant overhead of Bounded CAS is shown as thread numbers are low. Sequential has no exit time.

- **Worst-Case Entry Time**

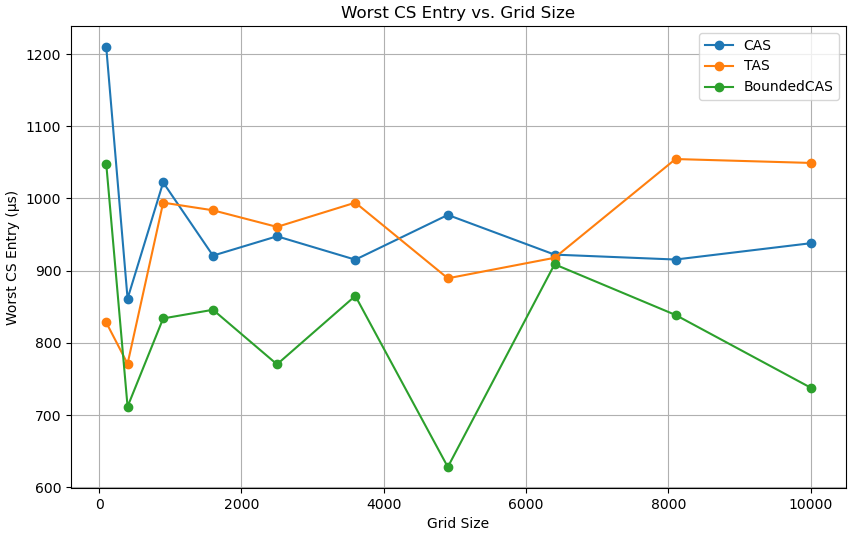

Figure 3: Worst-case CS entry time comparison

Observations: TAS shows spikes (1ms+ for large puzzles), CAS has moderate spikes, Bounded CAS has a relatively low time due to cyclic accesses and relatively low number of threads.

- **Worst-Case Exit Time**

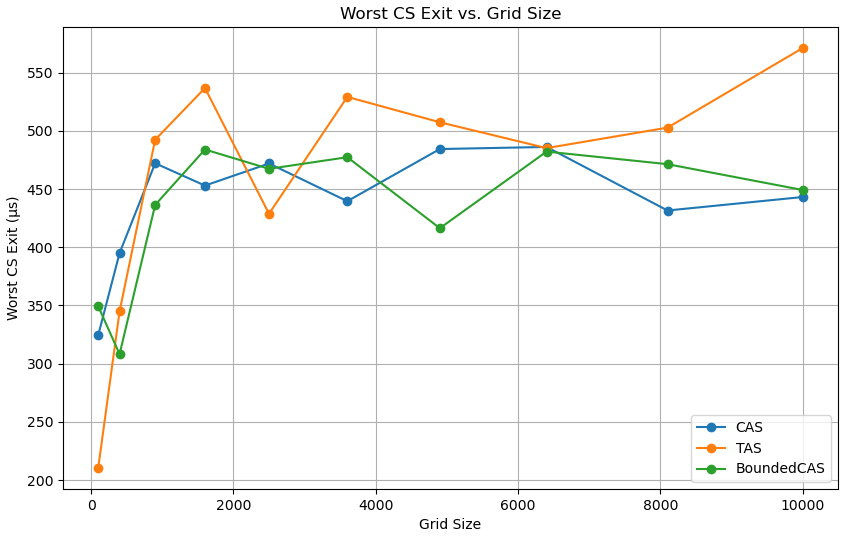

Figure 4: Worst-case CS exit time analysis

Observations: Exit time variance should be minimal for all locks (<2x average). Bounded CAS may

show slightly higher worst-case than normal CAS due to an extra iteration for finding the nearest waiting thread.

- **Total Execution Time**

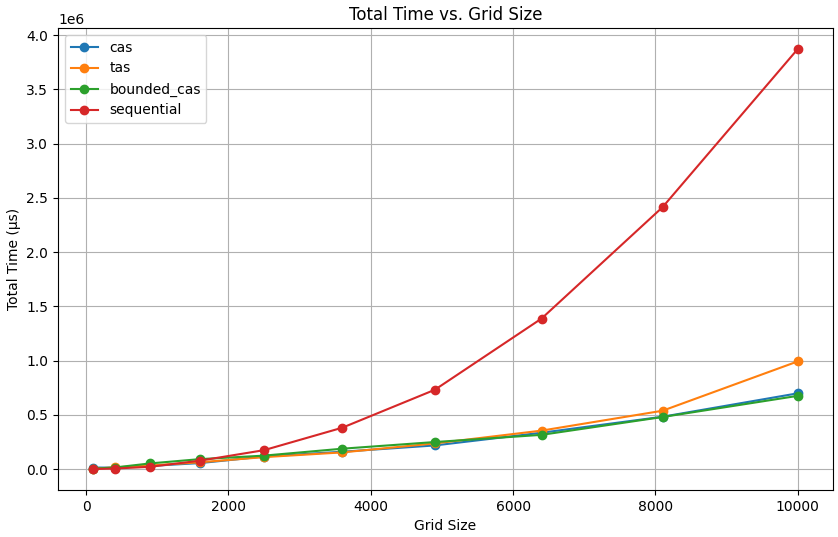

Figure 5: Total time vs puzzle size

Observations: Bounded CAS should match sequential for small sizes and scale best. TAS becomes impractical ¿40x40 grids. CAS sits between them.

Table 1: Average Case Entry/Exit Times (μs)

S.No	Grid Size	TAS Avg Entry	CAS Avg Entry	Bnd CAS Avg Entry	TAS Avg Exit	CAS Avg Exit	Bnd CAS Avg Exit
1	100	691.05	960.70	1453.05	252.75	304.52	330.95
2	400	908.37	992.02	798.23	950.60	975.05	1083.72
3	900	1301.35	1348.90	966.52	1640.33	1685.10	1695.68
4	1600	1660.98	1572.65	1231.02	2623.60	2707.93	2698.52
5	2500	1492.03	1765.25	1252.15	3735.67	3748.60	3761.40
6	3600	1887.97	1578.62	1360.90	5325.28	4891.77	5186.23
7	4900	1870.22	1801.90	1303.78	6668.80	6622.82	6761.55
8	6400	2182.33	1776.77	1653.12	9057.65	8914.47	9090.05
9	8100	1993.03	1998.40	1563.17	11 171.06	11 274.08	11 203.82
10	10000	2301.25	1998.98	1681.57	13 649.96	13 468.66	14 154.90

Table 2: Worst-Case Entry/Exit Times (μs)

S.No	Grid Size	TAS Wst Entry	CAS Wst Entry	Bnd CAS Wst Entry	TAS Wst Exit	CAS Wst Exit	Bnd CAS Wst Exit
1	100	828.20	1209.80	1048.20	210.40	324.60	349.40
2	400	771.00	861.20	711.60	345.00	395.00	308.20
3	900	994.20	1022.20	833.80	492.40	472.20	436.40
4	1600	983.60	920.80	845.80	536.80	453.00	484.00
5	2500	960.60	947.60	770.20	428.80	472.00	467.40
6	3600	994.20	915.20	865.00	529.20	439.60	477.40
7	4900	889.40	977.20	628.00	507.40	484.40	416.40
8	6400	917.80	922.20	908.80	485.20	486.20	482.20
9	8100	1054.60	915.40	838.60	502.80	431.60	471.40
10	10000	1049.20	938.00	737.80	571.20	443.20	449.20

Table 3: Total Execution Time (μs)

S.No	Grid Size	TAS Total Time	CAS Total Time	Bnd CAS Total Time
1	100	2426.00	2946.60	3082.20
2	400	5139.40	5884.80	4053.40
3	900	7430.20	7312.20	6660.20
4	1600	14 635.60	13 656.00	13 831.60
5	2500	33 486.20	33 743.00	32 094.20
6	3600	68 485.20	66 122.80	66 502.20
7	4900	1 244 646.20	1 234 481.80	1 218 270.40
8	6400	214 836.40	2 152 171.60	390 011.80
9	8100	543 639.00	559 061.00	496 225.20
10	10000	602 654.00	984 061.60	642 150.20

1.2 Experiment 2: Time vs. Task Increment (Fixed Sudoku Size)

- **Average Entry Time Comparison**

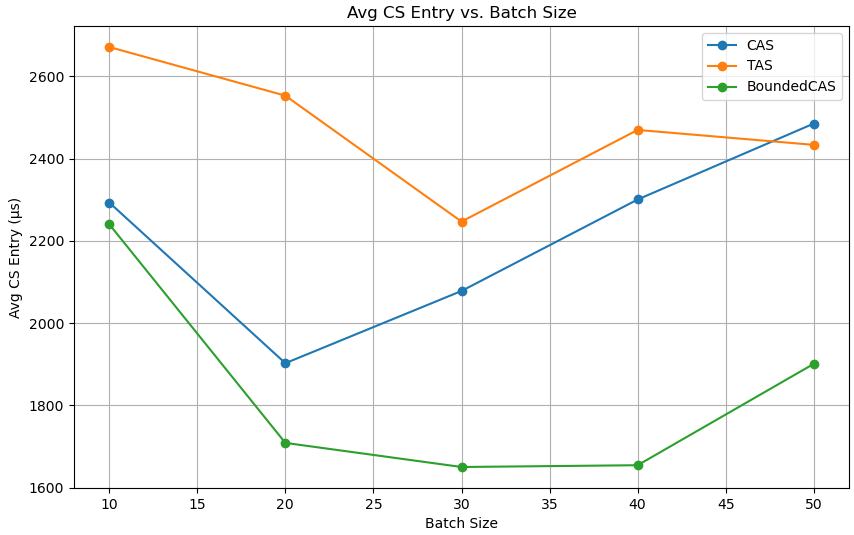

Figure 6: Entry time vs taskInc (90x90 puzzle)

Observations: TAS entry time decreases with larger taskInc (less frequent locking).Same goes for Bounded CAS. CAS shows a dip but then increases which could be due to extra time taken for grabing the taskInc tasks.

- **Average Exit Time Comparison**

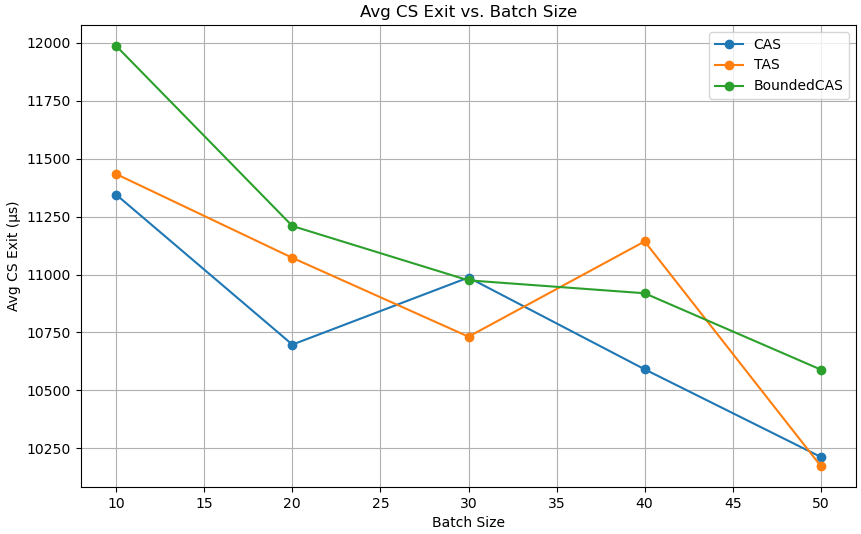

Figure 7: Exit time vs task increment

Observations: All locks show decreased exit times. Bounded CAS has slightly higher but consistent which could be due to overhead of find nearest waiting thread.

- **Worst-Case Entry Time**

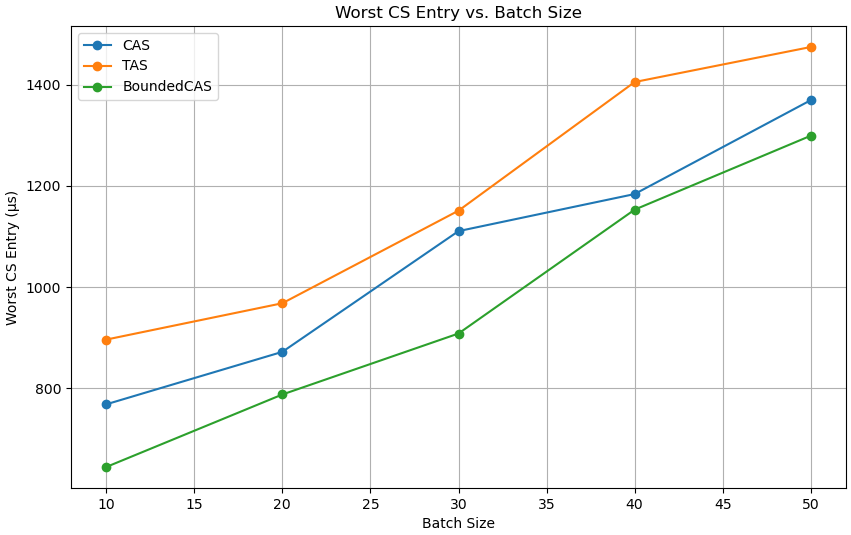

Figure 8: Lock attempts vs taskInc

Observations: There is a linear increase in the worst entry time which could be due to extra waiting as thread takes more time to grab tasks due to there increased numbers.

- **Worst-Case Exit Time**

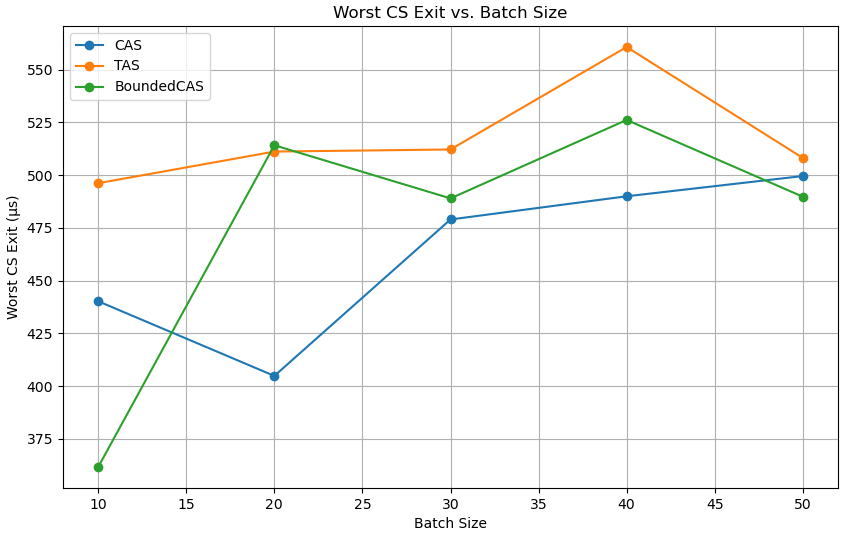

Figure 9: Tasks completed per second

Observations: Bounded CAS achieves maximum throughput at taskInc=30. TAS only becomes viable at taskInc¿40.

- **Scalability Limit**

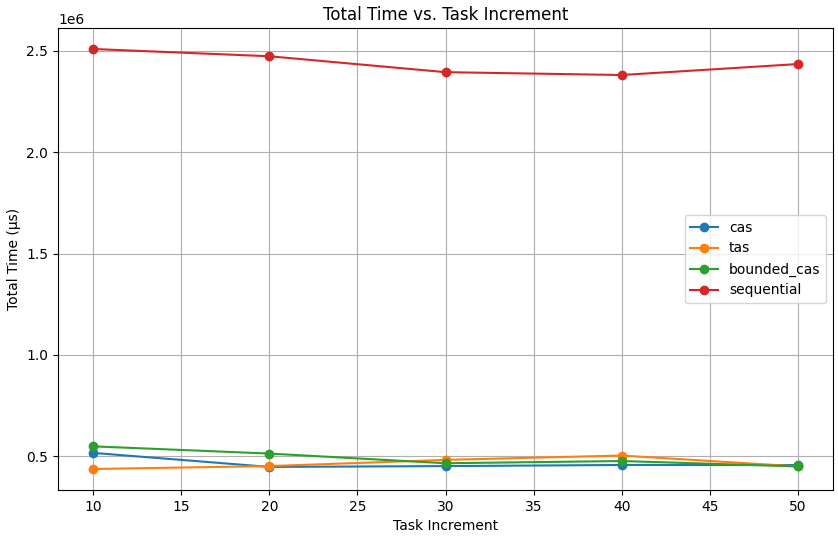

Figure 10: Total Time

Observations: Each method takes similar time as at every moment nearly same number of threads are working in each method. Sequential has a higher validation time due to lack of parallelism.

Table 4: Average Case Entry/Exit Times (μs)

S.No	TaskInc	TAS Avg Entry	CAS Avg Entry	Bnd CAS Avg Entry	TAS Avg Exit	CAS Avg Exit	Bnd CAS Avg Exit
1	10	2671.60	2293.37	2240.55	11 433.20	11 344.28	11 985.68
2	20	2553.22	1902.48	1708.65	11 071.90	10 697.70	11 209.30
3	30	2246.93	2078.17	1650.03	10 731.84	10 987.08	10 974.82
4	40	2469.88	2300.67	1654.35	11 142.82	10 591.24	10 919.20
5	50	2433.57	2485.40	1900.95	10 174.07	10 213.10	10 589.96

Table 5: Worst-Case Entry/Exit Times (μs)

S.No	TaskInc	TAS Wst Entry	CAS Wst Entry	Bnd CAS Wst Entry	TAS Wst Exit	CAS Wst Exit	Bnd CAS Wst Exit
1	10	896.40	768.60	644.80	496.20	440.20	361.60
2	20	968.20	872.20	788.00	511.20	404.80	514.20
3	30	1151.00	1110.60	908.40	512.20	479.00	489.00
4	40	1405.00	1183.80	1153.00	560.80	490.00	526.20
5	50	1474.20	1369.20	1299.00	508.20	499.60	489.80

Table 6: Total Execution Time (μs)

S.No	TaskInc	TAS Total Time	CAS Total Time	Bnd CAS Total Time
1	10	544 580.20	421 239.20	540 392.20
2	20	593 953.60	471 405.60	475 909.40
3	30	428 748.00	4 412 954.40	4 399 458.40
4	40	487 141.40	393 534.40	416 422.00
5	50	420 945.00	443 490.20	441 919.80

1.3 Experiment 3: Time vs. Number of Threads

- **Average Entry Time Comparison**

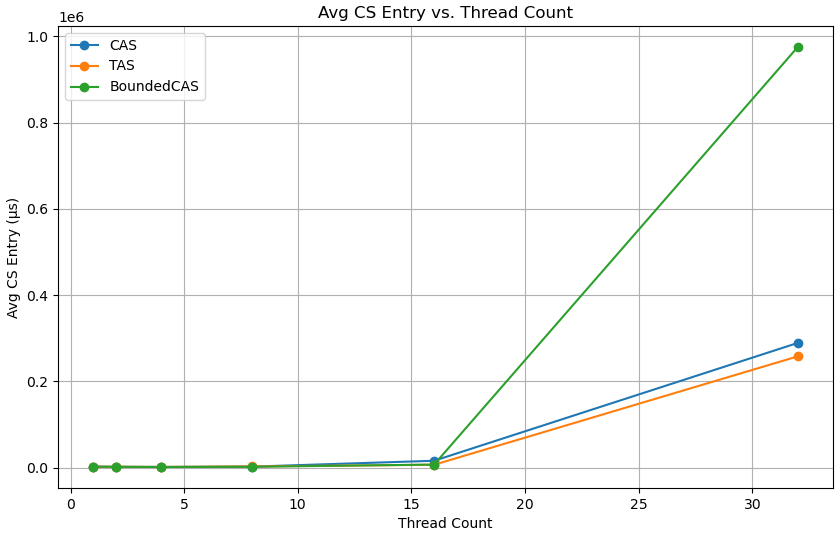

Figure 11: Entry time vs thread count

Observations: As the number of threads increases and only one thread can enter the CS so the average wait time increases. Bounded CAS has a greater average entry time as prevention of starvation comes with a higher waiting time of others.

- **Average Exit Time Comparison**

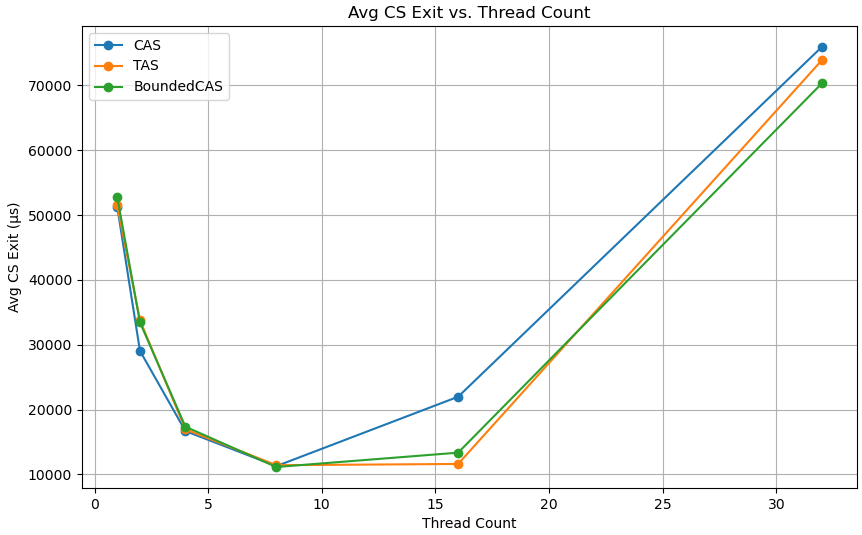

Figure 12: Exit time vs threads

Observations: Almost each thread shows a dip at initial increase in the thread count but increase again in a similar fashion.

- **Worst-Case Entry Time**

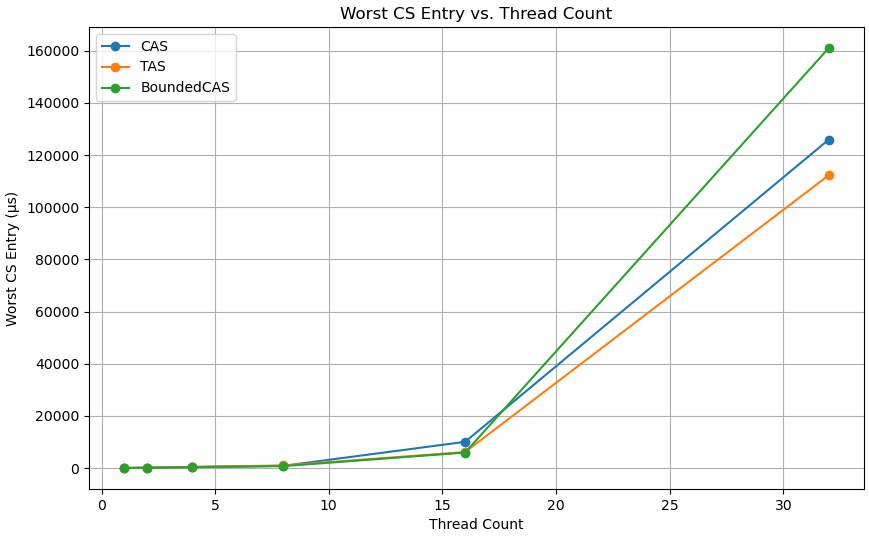

Figure 13: Worst-Case Exit Time

Observations: Worst case CS entry time increase as the number of threads increase so one particular thread has to wait for all others to enter the critical thread and as the number of thread increases that ovrhead also increases.

- **Fairness Analysis**

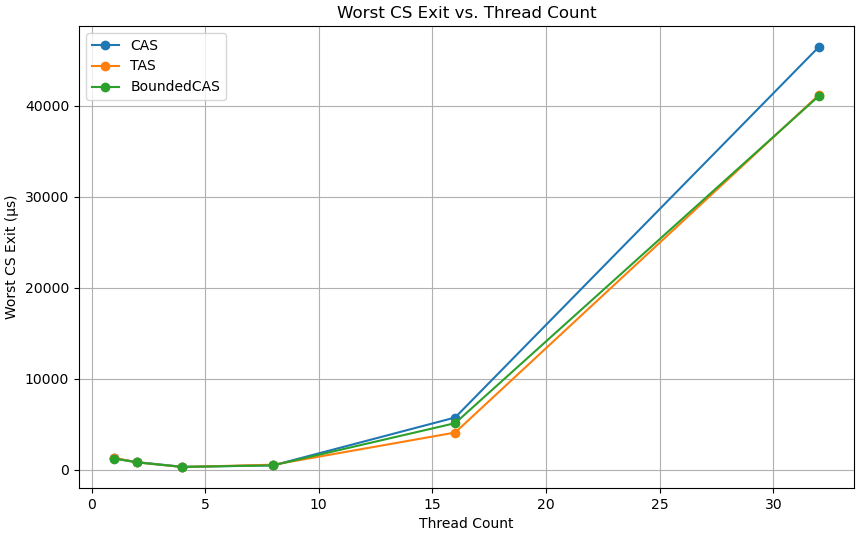

Figure 14: Total time

Observations: Same argument can be given in the time of exit. I was expecting bounded to have increased exit time as first it will check for all n to find a waiting thread as n would increae the time for checking will also be large but that trend is not followed here.

- **Core Utilization**

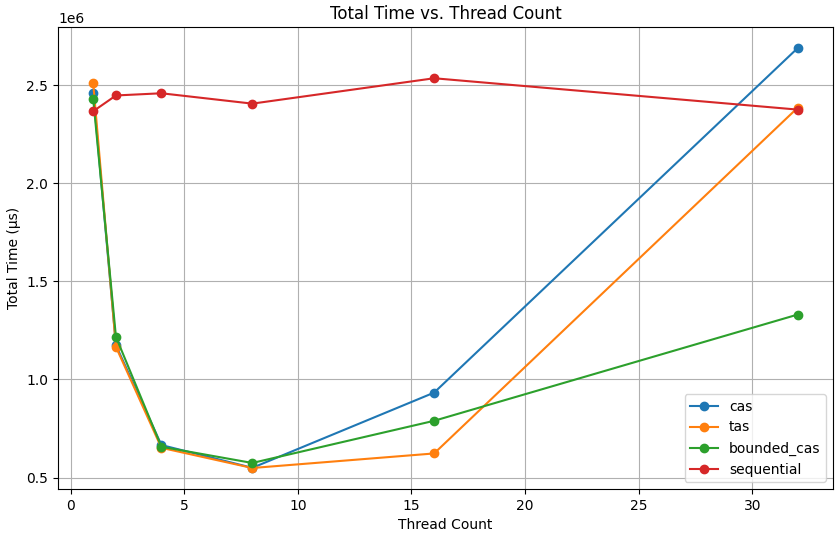

Figure 15: CPU utilization efficiency

Observations: Bounded CAS achieves a lower execution time due to lack of starvation and more threads able to work together which lacks in the other to methods.

Table 7: Average Case Entry/Exit Times (μs)

S.No	Threads	TAS Avg Entry	CAS Avg Entry	Bnd CAS Avg Entry	TAS Avg Exit	CAS Avg Exit	Bnd CAS Avg Exit
1	1	1748.80	1796.00	1991.60	51 525.60	51 299.20	52 811.40
2	2	1698.50	1500.20	1665.60	33 766.00	29 061.10	33 460.70
3	4	1333.70	1207.35	1242.20	16 962.88	16 683.70	17 356.34
4	8	2488.62	2100.72	1716.80	11 416.92	11 273.66	11 140.62
5	16	6424.22	15 685.15	6850.89	11 611.08	21 978.16	13 352.86
6	32	257 737.00	288 873.80	975 078.60	73 881.00	75 928.86	70 304.70

Table 8: Worst-Case Entry/Exit Times (μs)

S.No	Threads	TAS Wst Entry	CAS Wst Entry	Bnd CAS Wst Entry	TAS Wst Exit	CAS Wst Exit	Bnd CAS Wst Exit
1	1	63.00	62.60	79.60	1274.00	1232.40	1197.40
2	2	173.20	147.60	166.40	784.20	818.20	781.40
3	4	458.00	466.00	377.20	275.80	296.40	280.60
4	8	1022.00	899.80	786.20	527.80	461.00	460.80
5	16	6182.00	10 090.60	5983.20	4065.20	5720.40	5097.40
6	32	112 305.60	125 962.20	161 073.60	41 188.80	46 472.80	41 077.60

Table 9: Total Execution Time (μs)

S.No	Threads	TAS Total Time	CAS Total Time	Bnd CAS Total Time
1	1	2 644 030.40	2 430 368.40	2 290 993.80
2	2	16 689 301.60	1 250 323.20	1 261 465.20
3	4	880 019.20	975 868.40	733 657.00
4	8	661 090.80	739 294.20	7 452 138.40
5	16	820 645.60	769 452.20	929 932.00
6	32	1 478 726.60	2 368 995.20	2 794 767.40