CS3523: Programming Assignment-3

DOKKU HEMANADH
CS22BTECH11018

March 4 , 2024

1 Introduction

Parallel Matrix Multiplication with Dynamic Mechanism in C++ This assignment extends the task of parallel matrix multiplication in C++. The goal is to compute the square of a given square matrix A in parallel using dynamic mechanisms. The assignment involves dynamically allocating rows to threads for computation. Each thread increments a shared counter C by a value $rowInc$, representing the number of rows allocated to each thread. Synchronization issues arise when threads compete to increment C, which are addressed through various mutual exclusion algorithms including Test-and-Set (TAS), Compare-and-Swap (CAS), Bounded CAS, and atomic increment provided by the C++ atomic library. The main program reads parameters such as N (number of rows of A), K (number of threads), $rowInc$, and the matrix A from an input file in row-major order.

2 Aims and Objectives of the Provided C++ Code with Multiple Synchronization Algorithms

2.1 Aim

The primary aim of the code is to perform efficient matrix multiplication using parallel processing techniques while ensuring proper synchronization to prevent race conditions.

2.2 Objectives

1. **Matrix Multiplication**:
   Efficiently perform matrix multiplication leveraging parallel processing techniques.

2. **Parallelization**:
   Utilize parallel processing to distribute the matrix multiplication task among multiple threads, exploiting the computational power of multi-core processors.

3. **Thread Synchronization**:
   Implement various synchronization algorithms to coordinate access to shared resources among multiple threads, namely:

(a) **TAS (Test and Set)**:
Utilize the Test and Set atomic operation to set a lock, allowing only one thread at a time to access critical sections.

(b) **CAS (Compare and Swap)**:
Use the Compare and Swap atomic operation to atomically update the lock variable, ensuring mutual exclusion without busy waiting.

(c) **Bounded CAS (Bounded Compare and Swap)**:
Employ a variant of CAS where the operation succeeds only if the current value of the variable matches a specified expected value, preventing ABA problems.

(d) **Atomic Increment**:
Utilize atomic operations to safely increment shared variables, ensuring correct counting and distribution of work among threads.

4. **Efficiency**:
Strive for efficient utilization of system resources by minimizing idle time and maximizing CPU usage through parallelization and synchronization techniques.

5. **Input and Output Handling**:
Properly handle input and output operations, including reading matrices from files, performing matrix multiplication, and writing the result to an output file.

6. **Performance Measurement**:
Measure the total execution time of the matrix multiplication process using system time functions, aiding in performance evaluation and optimization.

3 Code Explanation

3.1 Global Variables

- `int counter;`
- `std::atomic_flag lock = ATOMIC_FLAG_INIT;`

3.2 Function `calculate_chunk_TAS`

- `calculate_chunk_TAS` function: This function takes parameters `rowinc`, `n`, `matrix`, and `output`. It calculates a chunk of the matrix multiplication result based on the given parameters.
- `while (counter <= n)`: This loop iterates until the `counter` exceeds `n`.
- Nested `while` loop: This loop ensures that only one thread at a time can access the critical section by using a Test-and-Set locking mechanism. It repeatedly checks and waits until it successfully acquires the lock.
- Critical section: Inside the critical section, the code updates the `counter` variable and calculates the range (`a` to `b`) of rows to process.
- `lock.clear(memory_order_release)`: Releases the lock after the critical section.

- Nested loops: These loops iterate over the assigned chunk of rows and columns to compute the dot product for each element of the output matrix.
- `output[i][l] = s;`: Stores the computed result in the output matrix.

,

3.3 Function `calculate_chunk_CAS`

- `__sync_val_compare_and_swap(&lock, 0, 1)`: This is a Compare-and-Swap operation that atomically compares the value of `lock` with 0 and, if they are equal, sets `lock` to 1. This operation is used to acquire a lock for the critical section.
- `lock = 0;`: This releases the lock after the critical section is executed.
- `for (int i = a; i < b; i++)`: This loop iterates over the assigned chunk of rows to the current thread.
- `for (int l = 0; l < n; l++)`: This loop iterates over the columns of the output matrix.
- `for (int j = 0; j < n; j++)`: This loop performs the dot product calculation by iterating over the columns of the first matrix and the rows of the second matrix.
- `output[i][l] = s;`: This stores the result of the dot product in the output matrix.

3.4 Function `calculate_chunk_Bounded_CAS`

- This function implements a parallel matrix multiplication algorithm using a bounded CAS (Compare-And-Swap) synchronization primitive.
- It iterates until the `counter` reaches n.
- Inside the loop:
  - It sets `waiting[index]` to `true`, indicating that the current thread is waiting.
  - It enters a loop where it tries to acquire the lock using a CAS operation.
  - Once the lock is acquired:
    - It sets `waiting[index]` back to `false` and updates the `counter` variable.
  - It then searches for the next available thread (with index j) that is not waiting. If found, it sets its `waiting` status to `false`. If not found, it releases the lock.
  - It calculates the bounds a and b for the rows to be processed by the current thread.
  - It performs matrix multiplication for the selected rows and columns.
  - It stores the result in the output matrix.
- The function is designed to be called by multiple threads concurrently, with each thread processing a chunk of rows of the input matrix.

- The synchronization mechanism (`waiting` and `lock` variables) ensures that multiple threads do not concurrently access and modify shared resources.
- The matrix multiplication is being done naively without any optimizations like blocking or tiling.
- The `__sync_val_compare_and_swap` function is likely a compiler intrinsic for performing an atomic compare-and-swap operation.

3.5 Function `calculate_chunk_Atomic`

- `atomic<int> counter;`:
  - Declares an atomic integer variable named `counter`.
- `void calculate_chunk_Atomic(int rowinc, int n, const vector<vector<int>> &matrix, vector<vector<int>> &output)`:
  - Function signature for calculating a chunk of matrix multiplication.
- `int local_counter;`:
  - Declares a local integer variable `local_counter` to store the value fetched from the atomic `counter`.
- `while (true) { ... }`:
  - An infinite loop that breaks when the condition `local_counter >= n` is met.
- `local_counter = counter.fetch_add(rowinc);`:
  - Atomically fetches the current value of `counter`, increments it by `rowinc`, and assigns the original value to `local_counter`.
- `if (local_counter >= n) break;`:
  - If `local_counter` exceeds or equals `n`, the loop breaks, indicating that all rows have been processed.
- `int a = local_counter; int b = min(local_counter + rowinc, n);`:
  - Calculates the range of rows to process for the current chunk.
- Nested loops:
  - Iterate over the specified range of rows (`i`) and columns (`l`) to perform matrix multiplication for the chunk.
- `output[i][l] = s;`:
  - Stores the result of matrix multiplication in the `output` matrix.

3.6 Main Function

- Open the input file `inp.txt`.
- Check if the file is successfully opened. If not, print an error message and exit with an error code.
- Read matrix dimensions (`n`), number of threads (`k`), and `rowinc` from the input file.
- Initialize vectors `output` and `matrix`.
- Read values from the input file and populate the `matrix` vector.
- Measure the start time using `gettimeofday`.
- Create `k` threads, each calling the function with parameters `rowinc`, `n`, `matrix`, and `output`.
- Wait for all threads to finish using `join`.
- Measure the end time using `gettimeofday`.
- Calculate the total time taken for the computation.
- Open the output file `out.txt` to write results to.
- Check if the output file is successfully opened. If not, print an error message and exit with an error code.
- Write the total time taken to the output file.
- Write the resulting matrix to the output file.
- Return 0 to indicate successful execution.

3.7 Input and Output Handling

- The program reads input matrix data from a file named "inp.txt".
- It writes the output matrix and the total execution time to a file named "out1.txt".
- It writes the output matrix and the total execution time to a file named "out2.txt".
- It writes the output matrix and the total execution time to a file named "out3.txt".
- It writes the output matrix and the total execution time to a file named "out4.txt".

4 Graph Analysis

4.1 Experiment 1: Time vs. Size, N:

- The y-axis will show the time taken to compute the square matrix in this graph.
- The x-axis will be the values of N (size on input matrix) varying from 256 to 4096 (size of the matrix will vary as 256*256, 512*512, 1024*1024, ...) in the power of 2.
- We Fix the rowInc at 16 and K at 16 for all these algorithms.

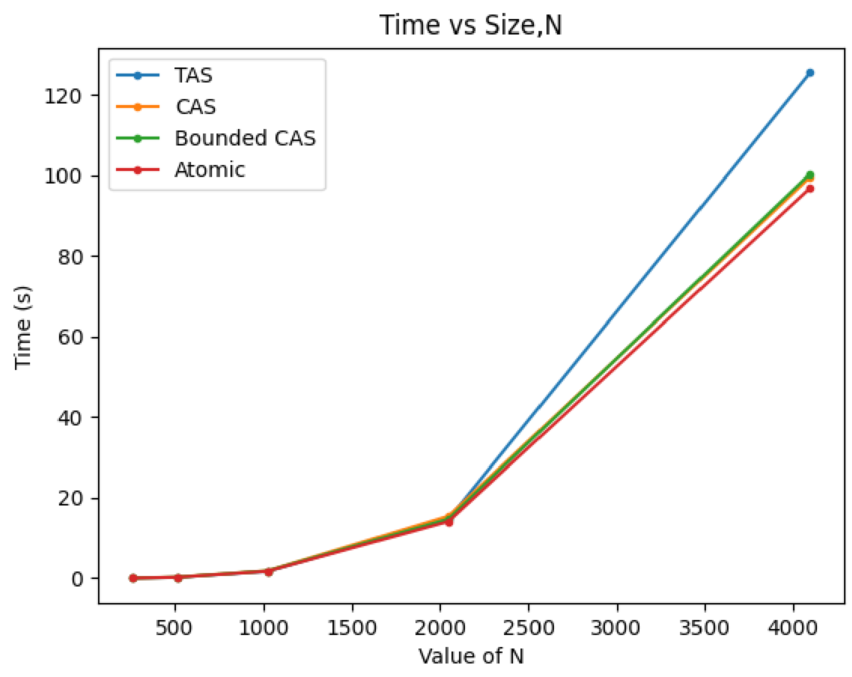

Figure 1: Time vs. Size(N)

N	TAS	CAS	BCAS	AC
256	0.034199	0.036306	0.035441	0.034578
512	0.243165	0.292414	0.243308	0.241274
1024	1.73937	1.85667	1.83271	1.74961
2048	14.6776	15.479	14.749	14.1422
4096	125.645	99.6725	100.5	96.962

Table 1: Total Time with varying N in seconds.

Analysis:

- Generally, larger matrix sizes result in longer execution times due to increased computational complexity.

- On increasing size, The number of row multiplications will increase and size of each row also increases so, the time taken will also increase gradually as we need to compute more number of rows. This is irrespective of algorithm.

4.2 Experiment 2: Time vs. rowInc, row Increment:

- The y-axis will show the time taken to compute the square matrix in this graph.
- The x-axis will be the rowInc varying from 1 to 32 (in powers of 2, i.e., 1,2,4,8,16,32).
- We fix N at 2048 and K at 16 for all these algorithms.

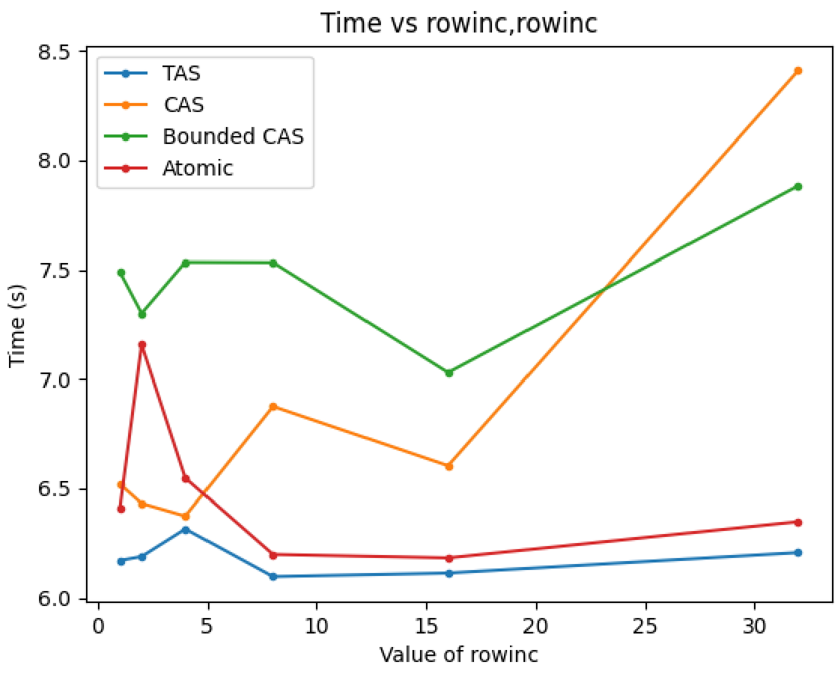

Figure 2: Time vs. rowInc.

Row Inc	TAS	CAS	B_CAS	Atomic
1	6.17214	6.52183	7.49016	6.40766
2	6.19005	6.43236	7.30067	7.15766
4	6.31618	6.37441	7.534	6.54862
8	6.09806	6.87604	7.53268	6.19986
16	6.11425	6.60458	7.02995	6.18385
32	6.20819	8.4094	7.88329	6.34945

Analysis: Here in this case there is some inconsistency for all the algorithms.

4.3 Experiment 3: Time vs. Number of threads, K:

- The y-axis will show the time taken to compute the square matrix in this graph.

- The x-axis will be the values of K, the number of threads varying from 2 to 32 (in powers of 2, i.e., 2,4,8,16,32).
- We fix N at 2048 and rowInc at 16 for all these experiments.

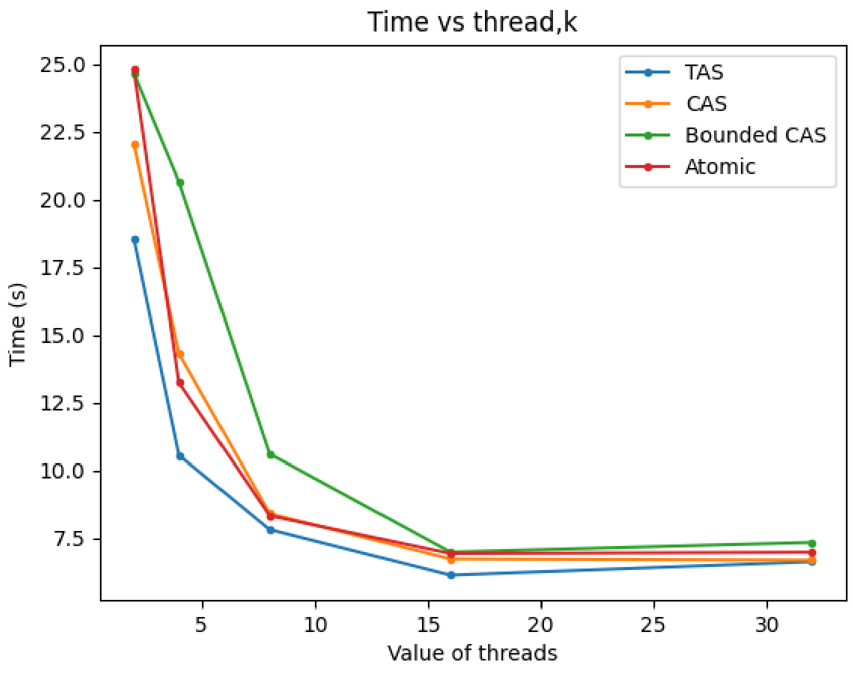

Figure 3: Time vs. Number of Threads (K).

Threads	TAS	CAS	BCAS	AC
2	18.5772	22.0679	24.6828	24.804
4	10.5757	14.3275	20.6604	13.2482
8	7.83132	8.419	10.6249	8.34749
16	6.1444	6.74288	6.99757	6.94484
32	6.64023	6.69712	7.35602	6.99089

Table 2: Data Table

Analysis:

- On increasing number of threads, The number of row computed parallelly will increase so, the time taken will decrease as we can compute more rows parallely. This is irrespective of algorithm.
- However, beyond a certain threshold, adding more threads may result in diminishing returns or even increased overhead due to contention for shared resources.

4.4 Experiment 4: Time vs. Algorithms:

- The y-axis will show the time taken to compute the square matrix in this graph.

- The x-axis will be different algorithms -
  1. Static rowInc
  2. Static mixed
  3. Dynamic with TAS
  4. Dynamic with CAS
  5. Dynamic with Bounded CAS
  6. Dynamic with Atomic.

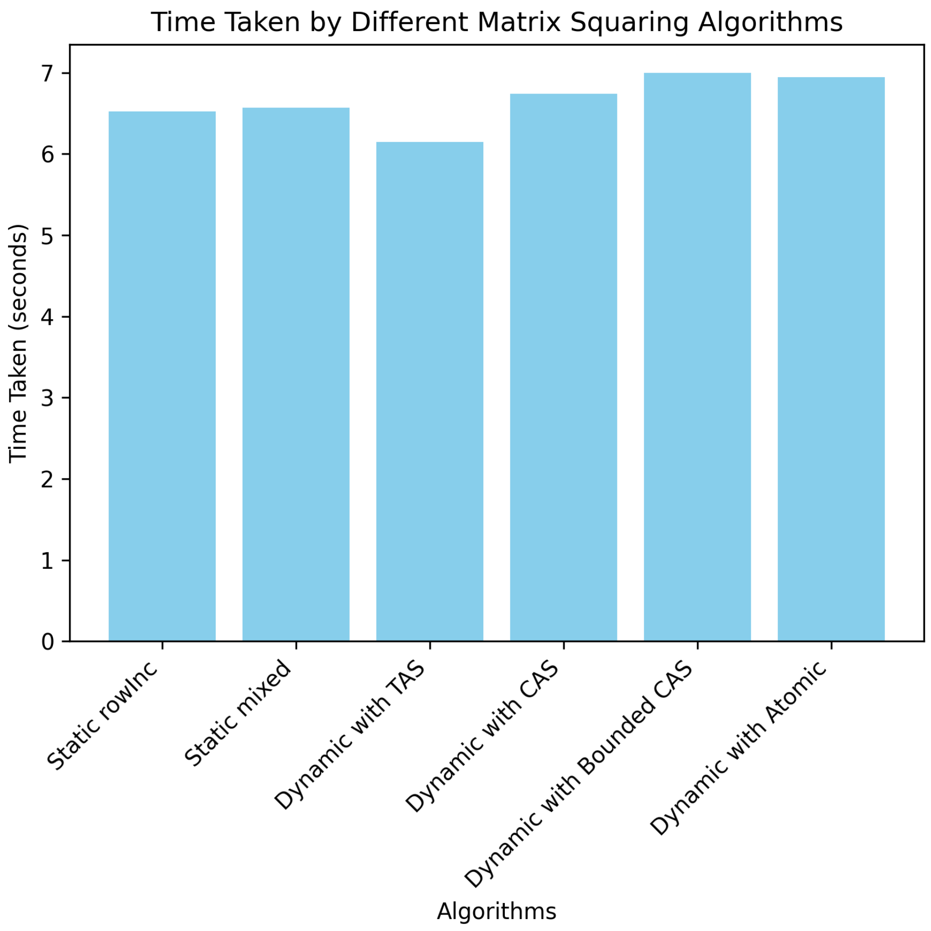

Figure 4: Time vs. Algorithms.

Algorithm	**Time taken**
Static rowInc	6.52643
Static Mixed	6.56961
Dynamic with TAS	6.1444
Dynamic with CAS	6.74288
Dynamic with B_CAS	6.99757
Dynamic with Atomic	6.94484

Table 3: Total Time for different algorithms in seconds.

5 OBSERVATIONS

Experiments

1. **Experiment 1:** The time taken increases with an increase in N. All the algorithms exhibit similar performance.

2. **Experiment 2:** Performance of algorithms shows slight randomness and unpredictability. All the algorithms perform comparably.

3. **Experiment 3:** The time taken decreases with an increase in K. An anomaly is observed - BCAS algorithms perform slightly worse for smaller values of K. This might be attributed to the fact that with a smaller number of threads, issues of thread starvation due to unbounded waiting are less pronounced. Consequently, the implementation of bounded waiting might introduce overhead.

4. **Experiment 4:** Dynamic algorithms outperform static algorithms. This may be due to the uneven work distribution of static algorithms for certain matrix inputs. In such cases, some threads finish early while others are still active, leading to inefficient resource utilization. In contrast, dynamic algorithms allocate work dynamically, allowing early finishing threads to receive new work assignments.