Operating Systems–2: Spring 2024
Programming Assignment 3:
Dynamic Matrix Squaring
Report

Prabhat - CO22BTECH11009

Program Overview:

Below is an overview of the program:

- ❒ In this programming assignment, we will build upon our previous work measuring the performance of calculating the square of matrix A in parallel in C++. Previously, we allocated the rows of the matrix to threads statically (in either mixed or chunk mode).
- ❒ In this assignment, the sequence of rows will be allocated dynamically to the threads.
- ❒ Each thread increments a shared counter C by a value of rowInc (row Increment) to claim the set of rows of the C matrix it is responsible for. The value rowInc can be seen similarly to the chunk size of assignment 1, where the chunk size was fixed statically.
- ❒ There will be synchronization issues when threads compete to increment the counter C. To solve this problem, I have implemented different mutual exclusion algorithms to increment C, which are: (a) TAS, (b) CAS, (c) Bounded CAS, and (d) atomic increment (provided by the C++ atomic library).

Specifications of the System:

❒ Hardware Information:

1. Hardware Model: ASUSTeK COMPUTER INC. ASUS TUF Gaming A15 FA506ICB_FA506ICB
2. Memory: 16.0 GiB
3. Processor: AMD Ryzen™ 7 4800H with Radeon™ Graphics × 16
4. Graphics: AMD Radeon™ Graphics
5. Disk Capacity: 512.1 GiB
6. CPU Cores: 8 (Octa core)
7. Threads: 16

❒ Software Information:

1. Firmware Version: FA506ICB.304
2. OS Name: Ubuntu 23.10
3. OS Build: (null)
4. OS Type: 64-bit
5. Windowing System: Wayland
6. Kernel Version: Linux 6.5.0-15-generic

TAS Design:

This program performs matrix multiplication with Test-and-Set (TAS) synchronization. The low-level design for the chunk program is as follows:

1. Atomic Variables:

    - ❒ atomic int counter(0): An atomic integer counter initialized to 0, used to keep track of the current row being processed by threads.
    - ❒ atomic_flag lock = ATOMIC_FLAG_INIT: An atomic flag 'lock' initialized to clear state, used for mutual exclusion to prevent multiple threads from accessing the critical section simultaneously.

2. Matrix Multiplication Function ('multiply'):

    - ❒ This function is called by each thread to compute a portion of the resulting matrix.
    - ❒ It takes references to matrices A (input matrix) and C (result matrix), the size of the matrix N, and the row increment rowInc.
    - ❒ It uses a while loop to repeatedly acquire the lock using test_and_set() function to enter the critical section.
    - ❒ Inside the critical section, it updates start_row and end_row based on the current value of counter, increments counter, and releases the lock.
    - ❒ It then performs matrix multiplication for the assigned rows.
    - ❒ The loop breaks when all rows have been processed.

3. Thread Assignment Function ('TAS'):

    - ❒ This function creates and manages multiple threads to perform matrix multiplication concurrently.
    - ❒ It takes references to matrices A and C, the size of the matrix N, the number of threads K, and the row increment rowInc.
    - ❒ It creates K threads, each calling the 'multiply' function with appropriate parameters.
    - ❒ After creating all threads, it waits for each thread to finish using 'join()'

4. Main Function ('main'):

    - ❒ Reads input parameters (N, K, rowInc) and matrix A from a file (inp.txt).
    - ❒ Calls the TAS function to perform matrix multiplication.
    - ❒ Measures the execution time using the chrono library.
    - ❒ Writes the result matrix C and the execution time to an output file (out_TAS.txt).

5. Input and Output:

   - The input matrix A, size parameters N, K, rowInc are read from a file (inp.txt).
   - The resulting matrix C and the execution time are written to an output file (out_TAS.txt).

6. Synchronization:

   - Atomic operations ('test_and_set()', 'clear()') are used for mutual exclusion to ensure that only one thread accesses the critical section at a time.

Overall, the program efficiently utilizes multiple threads to perform matrix multiplication concurrently while ensuring synchronization using atomic operations.

CAS Design:

This program performs matrix multiplication with Compare-and-Swap (CAS) synchronization. The low-level design for the chunk program is as follows:

1. Atomic Variables:
   - atomic int counter(0): An atomic integer counter initialized to 0, used to keep track of the current row being processed by threads.
   - atomic bool lock(false): An atomic boolean variable initialized to false, used for mutual exclusion to prevent multiple threads from accessing the critical section simultaneously.
2. Matrix Multiplication Function ('multiply'):
   - This function is called by each thread to compute a portion of the resulting matrix.
   - It takes references to matrices A (input matrix) and C (result matrix), the size of the matrix N, and the row increment rowInc.
   - It employs a while loop to repeatedly acquire the lock using CAS operation to enter the critical section.
   - Inside the critical section, it updates the start row and end row based on the current value of the counter, increments counter, and releases the lock.
   - It then performs matrix multiplication for the assigned rows.
   - The loop breaks when all rows have been processed.
3. Thread Assignment Function ('CAS'):
   - This function creates and manages multiple threads to perform matrix multiplication concurrently.
   - It takes references to matrices A and C, the size of the matrix N, the number of threads K, and the row increment rowInc.
   - It creates K threads, each calling the multiply function with appropriate parameters.
   - After creating all threads, it waits for each thread to finish using 'join()'
4. Main Function ('main'):
   - Reads input parameters (N, K, rowInc) and matrix A from a file (inp.txt).
   - Calls the CAS function to perform matrix multiplication.
   - Measures the execution time using the chrono library.
   - Writes the result matrix C and the execution time to an output file (out_CAS.txt).

5. Input and Output:

   - The input matrix A, size parameters N, K, rowInc are read from a file (inp.txt).
   - The resulting matrix C and the execution time are written to an output file (out_CAS.txt).

6. Synchronization:

   - CAS (Compare-and-Swap) operation is used for mutual exclusion to ensure that only one thread accesses the critical section at a time.

Overall, the program efficiently utilizes multiple threads to perform matrix multiplication concurrently while ensuring synchronization using CAS operation for locking.

Bounded CAS Design:

This program performs matrix multiplication with Bounded Compare-and-Swap (CAS) synchronization. The low-level design for the chunk program is as follows:

1. Atomic Variables:
   - atomic int counter(0): An atomic integer counter initialized to 0, used to keep track of the current row being processed by threads.
   - atomic bool lock(false): An atomic boolean variable initialized to false, used for mutual exclusion to prevent multiple threads from accessing the critical section simultaneously.
   - const int MAX_RETRIES = 1000: Constant defining the maximum number of retries for acquiring the lock.
2. Matrix Multiplication Function ('multiply'):
   - This function is called by each thread to compute a portion of the resulting matrix.
   - It takes references to matrices A (input matrix) and C (result matrix), the size of the matrix N, and the row increment rowInc.
   - It employs a while loop to repeatedly acquire the lock using CAS operation to enter the critical section, with a bounded number of retries.
   - Inside the critical section, it updates the start row and end row based on the current value of the counter, increments the counter, and releases the lock.
   - It then performs matrix multiplication for the assigned rows.
   - The loop breaks when all rows have been processed or the maximum number of retries is reached.
3. Thread Assignment Function ('BoundedCAS'):
   - This function creates and manages multiple threads to perform matrix multiplication concurrently.
   - It takes references to matrices A and C, the size of the matrix N, the number of threads K, and the row increment rowInc.
   - It creates K threads, each calling the multiply function with appropriate parameters.
   - After creating all threads, it waits for each thread to finish using 'join()'

4. Main Function ('main'):
   - ❒ Reads input parameters (N, K, rowInc) and matrix A from a file (inp.txt).
   - ❒ Calls the BoundedCAS function to perform matrix multiplication.
   - ❒ Measures the execution time using the chrono library.
   - ❒ Writes the result matrix C and the execution time to an output file (out_BoundedCAS.txt).
5. Input and Output:
   - ❒ The input matrix A, size parameters N, K, rowInc are read from a file (inp.txt).
   - ❒ The resulting matrix C and the execution time are written to an output file (out_BoundedCAS.txt).
6. Synchronization:
   - ❒ Bounded CAS (Compare-and-Swap) operation is used for mutual exclusion to ensure that only one thread accesses the critical section at a time, with a maximum number of retries to prevent indefinite waiting.

Overall, the program efficiently utilizes multiple threads to perform matrix multiplication concurrently while ensuring synchronization using bounded CAS operation for locking.

Atomic Design:

This program performs matrix multiplication using atomic increment synchronization. The low-level design for the chunk program is as follows:

1. Atomic Variables:
   - ❐ atomic int counter(0): An atomic integer counter initialized to 0, used to keep track of the current row being processed by threads.
2. Matrix Multiplication Function ('multiply'):
   - ❐ This function is called by each thread to compute a portion of the resulting matrix.
   - ❐ It takes references to matrices A (input matrix) and C (result matrix), the size of the matrix N, and the row increment rowInc.
   - ❐ It fetches the start row atomically by adding rowInc to the counter using atomic_fetch_add_explicit with relaxed memory order.
   - ❐ It employs a while loop to iteratively process rows until all rows have been processed.
   - ❐ Inside the loop, it computes the end row based on the start row and rowInc.
   - ❐ If the start row is less than N, it performs matrix multiplication for the assigned rows and updates the start row atomically.
   - ❐ The loop breaks when all rows have been processed.
3. Thread Assignment Function ('Atomic'):
   - ❐ This function creates and manages multiple threads to perform matrix multiplication concurrently.
   - ❐ It takes references to matrices A and C, the size of the matrix N, the number of threads K, and the row increment rowInc.
   - ❐ It creates K threads, each calling the 'multiply' function with appropriate parameters.
   - ❐ After creating all threads, it waits for each thread to finish using 'join()'
4. Main Function ('main'):
   - ❐ Reads input parameters (N, K, rowInc) and matrix A from a file (inp.txt).
   - ❐ Calls the Atomic function to perform matrix multiplication.
   - ❐ Measures the execution time using the chrono library.
   - ❐ Writes the result matrix C and the execution time to an output file (out_Atomic.txt).

5. Input and Output:

   - ❒ The input matrix A, size parameters N, K, rowInc are read from a file (inp.txt).
   - ❒ The resulting matrix C and the execution time are written to an output file (out_Atomic.txt).

6. Synchronization:

   - ❒ Atomic increment operation (atomic_fetch_add_explicit) is used to fetch and update the start row atomically, ensuring that each thread processes a unique set of rows without interference.

Overall, the program efficiently utilizes multiple threads to perform matrix multiplication concurrently while ensuring synchronization using atomic increment operation for updating the start row.

Experiment 1:

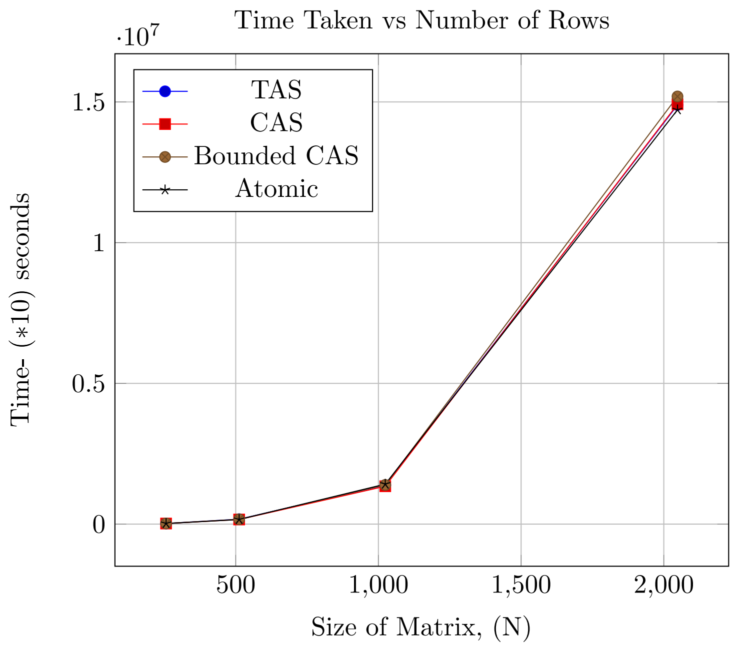


Figure 1: Time vs Size of Matrix (N)

Observations:

- ❒ As the size of the matrix, N increases, the time taken to calculate the square of the matrix also increases.
- ❒ All the algorithms, i.e. TAS, CAS, Bounded CAS and Atomic methods takes almost the same time.
- ❒ Also the pattern in the curve as the N increases is almost same for all the curves.

Experiment 2:

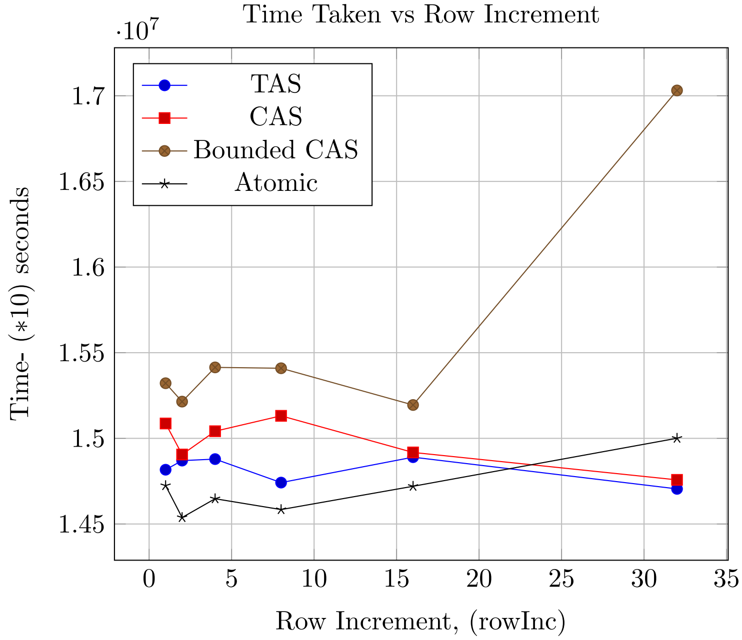


Figure 2: Time vs Row Increment (rowInc)

Observations:

❒ There is no such pattern between the taken taken for different value rowInc.

❒ All the algorithms have almost the same curve shape for initial values of rowInc.

❒ However, the time taken for matrix multiplication roughly follows the order as:

$$\text{Bounded CAS} \geq \text{CAS} \geq \text{TAS} \geq \text{Atomic}$$

Experiment 3:

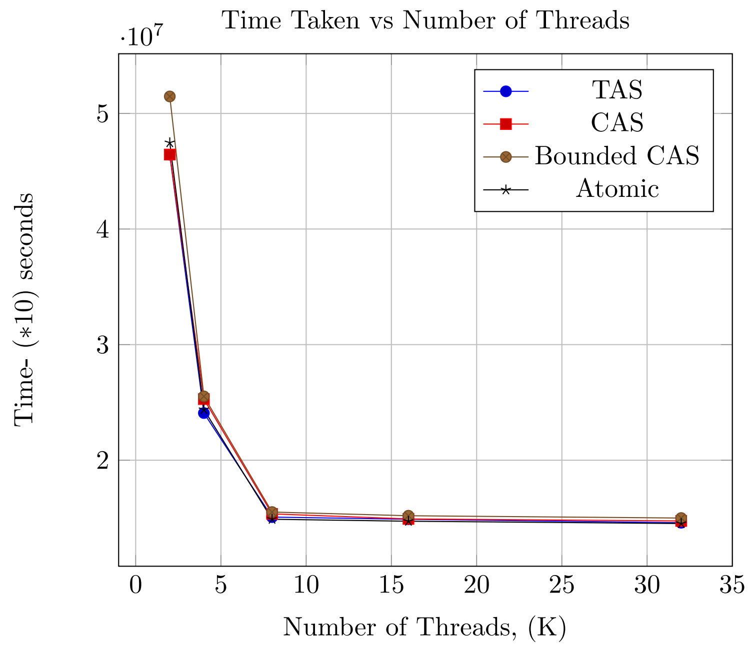
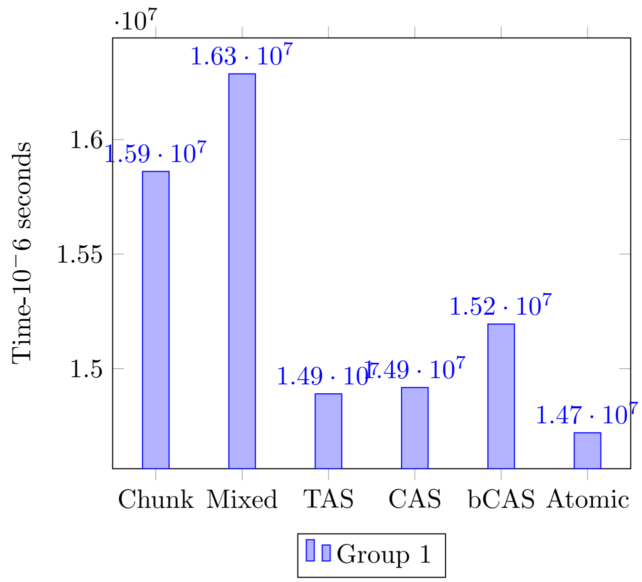


Figure 3: Time vs Number of Threads (K)

Observations:

- ❐ As the size of the matrix, N increases, the time taken to calculate the square of the matrix also increases.
- ❐ All the algorithms, i.e. TAS, CAS, Bounded CAS and Atomic methods takes almost the same time.
- ❐ Also the pattern in the curve as the N increases is almost same for all the curves.

Experiment 4:



Observations:

❒ Algorithms take time in the following order:

$$Mixed \geq Chunk \geq BoundedCAS \geq CAS \geq TAS \geq Atomic$$