Operating Systems–2: Spring 2024

Programming Assignment 3: Dynamic Matrix Squaring

Name : Rishitha Surineni
Roll No : CS22BTECH11050

Introduction

This is a C++ program to perform parallel matrix multiplication through a Dynamic mechanism.

Implementation

Processing the Input:

The Input consists of N (number of rows in the matrix),K (number of threads),rowInc and a square matrix with N rows. This input is read from an input file named “inp.txt”.

This input file is read and the matrix is stored in a 2D array.

Dividing data among threads:

A structure is defined which has a pointer to the input matrix and also a pointer to the output matrix where the result of the computation has to be stored.The other attributes in the structure are Number of Rows of the input matrix, the value of row Increment,the number of threads and the thread number of the particular thread to which this structure is passed.

Creating threads:

Threads are created using the **pthread_create()** function.
pthread_create(&Threads[thread_num],NULL,Calculate,(void *)&thread_data[thread_num]);

Threads is an array of data type pthread_t, these serve as identifiers for the threads,
Calculate is a function which is executed in each thread.The computation of the rows of the output matrix happens in this function.
Thread_data is an array of structures which have the attributes required by the threads to compute the matrix.

Each thread would compute the dot product of each of the rows assigned to it with all the columns in the matrix and stores the resultant row in the output matrix which was created in the main and passed to the thread as a pointer.

Dynamical assignment of rows to the threads:

This assignment of rows to the threads is dynamic and it happens based on a counter variable C.
As all the threads are trying to access the same variable C there would be synchronization issues.To prevent this a mutually exclusive algorithm has to be implemented which makes sure that only one of the threads is in its critical section at a point of time and the other threads are waiting.

Mutual Exclusion Techniques:

For this assignment four techniques are implemented for mutual exclusion they are

1) TAS
2) CAS
3) Bounded CAS
4) Atomic

1) TAS

TAS refers to test and set.
To implement this algorithm the function test_and_set() is used.
First an atomic_flag is declared and initialised using

atomic_flag lock_variable = ATOMIC_FLAG_INIT;

In the Calculate function which is executed in each thread an infinite do while loop is run so that a thread which has already calculated some rows can again get a chance to compute few more rows of the output matrix.

In this loop there will be another while loop to make the threads wait so that only one of them can access the critical section at a time

```
while(lock_variable.test_and_set()){
}
```

lock_variable.test_and_set() sets the lock_variable and returns its previous value.If the previous value was set (true), it means the lock was taken by another thread, and the current thread needs to wait.
This loop breaks if the value returned is false and this happens if the lock was successfully acquired.

After this the thread enters the critical section in which the value of C is stored in a temporary variable and C is incremented by rowIncrement.If the value of C is greater than N then the infinite loop breaks.

After this the lock is unlocked using lock_variable.clear(), this allows any one of the other threads to execute its critical section.

In the remainder section, a nested for loop to calculate the rows of output matrix. This contains a for loop on row number from this temporary variable for row Increment number of times.If the row number exceeds N then the loop breaks.Else each row is multiplied with all the columns and the output row is written in the output matrix.

2) CAS

CAS refers to compare and swap.

To implement this the function __sync_val_compare_and_swap() is used.

A global variable named value is declared and initialized with 0.

In the Calculate function an infinite while loop is run

In this while loop there will be an another while loop

```
while (__sync_val_compare_and_swap(&value, 0, 1)!=0){
      }
```

__sync_val_compare_and_swap(&value, 0, 1) checks the value at the address &value. If this is equal to 0 then it is updated to 1,else if it is 1 then it isn't updated. In both the cases the function returns the number which was initially at the address &value.

This while loop breaks when __sync_val_compare_and_swap(&value, 0, 1) returns 0,i.e when the value was 0 . This thread will then execute the critical section and meanwhile other threads will be struct in the while loop.

In the critical section the value of C is stored in a temporary variable ,if C is greater than N then value is updated to 0 and infinite while loop breaks otherwise the value of C is incremented by rowInc and value is set to 0.

The remainder section is the same for all the algorithms so the same as stated for TAS happens here too.

3) Bounded CAS

In this technique a global vector named waiting is declared. This vector tells whether the i th thread is waiting or not. It is initialized with 0 for all the threads.

In the Calculate function an infinite while loop is run.

In this while loop waiting[thread_num] is set to 1 and another variable named key is declared and it is set to 1.

Another while loop is run

```
while (waiting[num] && key == 1)
      {
         key = __sync_val_compare_and_swap(&value, 0, 1);
      }
```

This while loop runs till the waiting[num] becomes 0 or if key becomes 0. Compare and swap is used to calculate key value.Key becomes 0 if the value is 0.This means that this thread will execute the critical section and the others will be waiting.waiting[num] should be updated to 0, as this thread will execute critical section.

In the critical section the value of C is stored in a temporary variable and it is checked if C is greater than N.If so the infinite while loop breaks. Else C is incremented by rowInc.

A variable j is declared and it is given the value of present thread number +1 modulo number of threads. If j th thread is waiting then waiting[j] is made 0 so that the j th thread will break out of while loop and starts executing its critical section.In the similar way all the waiting threads are executed one after the other till there are no more waiting threads.

The remainder section is the same as of the above two algorithms.

4) Atomic

In this method, the Counter C is of atomic int type. This implies only one thread can update the value of C at a time. If multiple threads try to update C then it happens one after the other.

In the Calculate function an infinite loop is run.In this the value of C is checked. If it is greater than N then the loop breaks. Else the value of C is stored in a temporary variable and it is incremented by rowInc.

A nested for loop same as that in the remainder section of other 3 algorithms is run here too to calculate the output matrix's rows.

Calculation of Time :

Using the chrono library the time taken for computation of the square of the matrix is calculated.

The time taken for the whole process is calculated in the main function by starting a clock before threads creation and ending it after joining all the threads.

This time is written into the output file.

Output :

Each of the codes give an output file. They are named as out_TAS.txt, out_CAS.txt,out_BCAS.txt and out_Atomic.txt.

Each output file contains the time taken for computing the square of the given matrix using that particular technique for synchronization and the output matrix.

Results

Experiment-1

The below is the graph between time taken for the computation of square matrix versus the Size of the matrix.

This graph consists of 4 curves

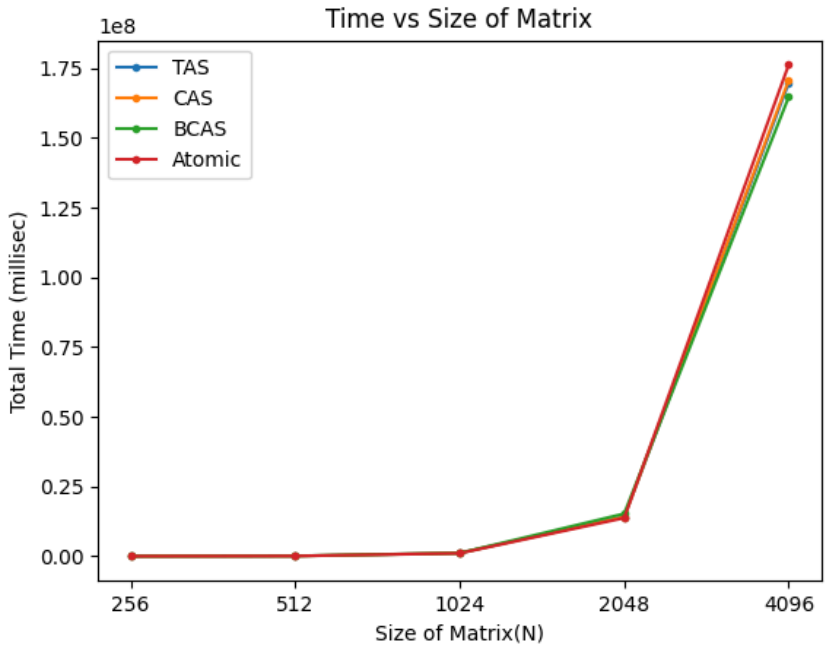
1. Time taken in case of TAS algorithm
2. Time taken in case of CAS algorithm
3. Time taken in case of Bounded CAS algorithm
4. Time taken in case of Atomic algorithm

Here ,

K=16

rowInc=16

And N value changes from 256 to 4096 in powers of 2.



From the graph it can be seen that

As the size of the matrix increases the time taken to compute the square of the matrix also increases. This is because the number of threads is fixed every time and as the size of the matrix is increasing each thread will be responsible for calculating more rows.
The time taken in all the algorithms will be almost the same.

Experiment-2
The below is the graph between time taken for the computation of square matrix versus the Row Increment.
This graph consists of 4 curves

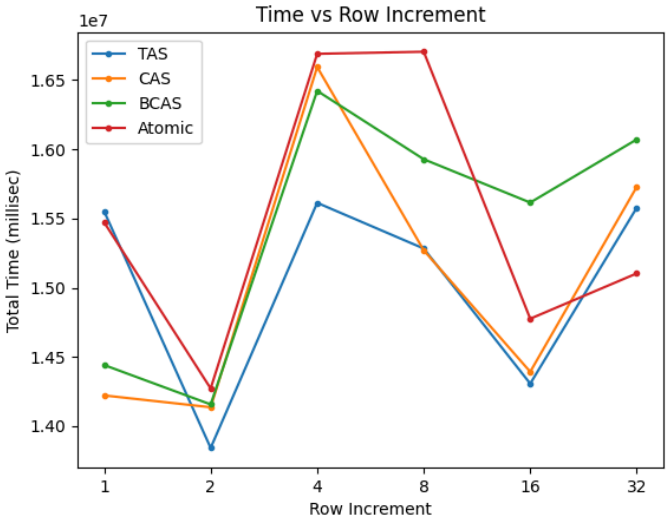
1) Time taken in case of TAS algorithm
2) Time taken in case of CAS algorithm
3) Time taken in case of Bounded CAS algorithm
4) Time taken in case of Atomic algorithm

Here ,

N=2048
K=16

And rowInc value changes from1 to 32 in powers of 2.

From the graph we can observe that there is no particular trend of the time taken for computation with changing row Increment. There is no significant relation between different algorithms too.The time taken depends on CPU condition and the other processes running on the system. In general the performance with all the algorithms will be similar.



Experiment-3

The below is the graph between time taken for the computation of square matrix versus the Number of Threads

This graph consists of 4 curves

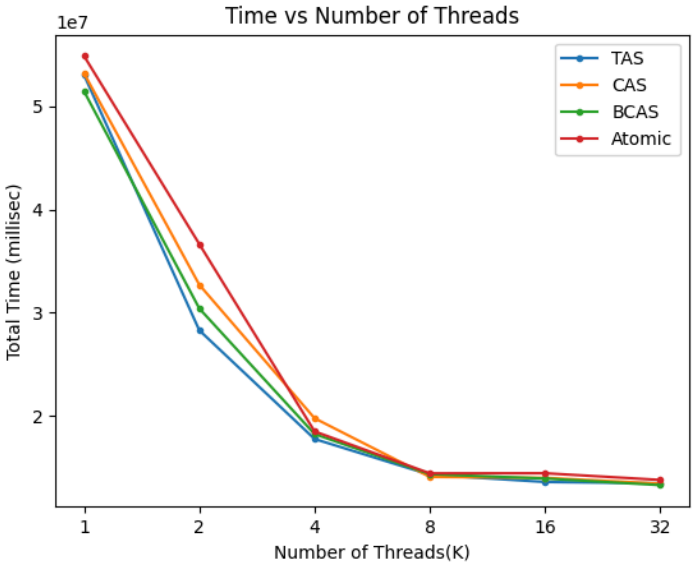
1) Time taken in case of TAS algorithm
2) Time taken in case of CAS algorithm
3) Time taken in case of Bounded CAS algorithm
4) Time taken in case of Atomic algorithm

Here ,

N=2048

rowInc=16

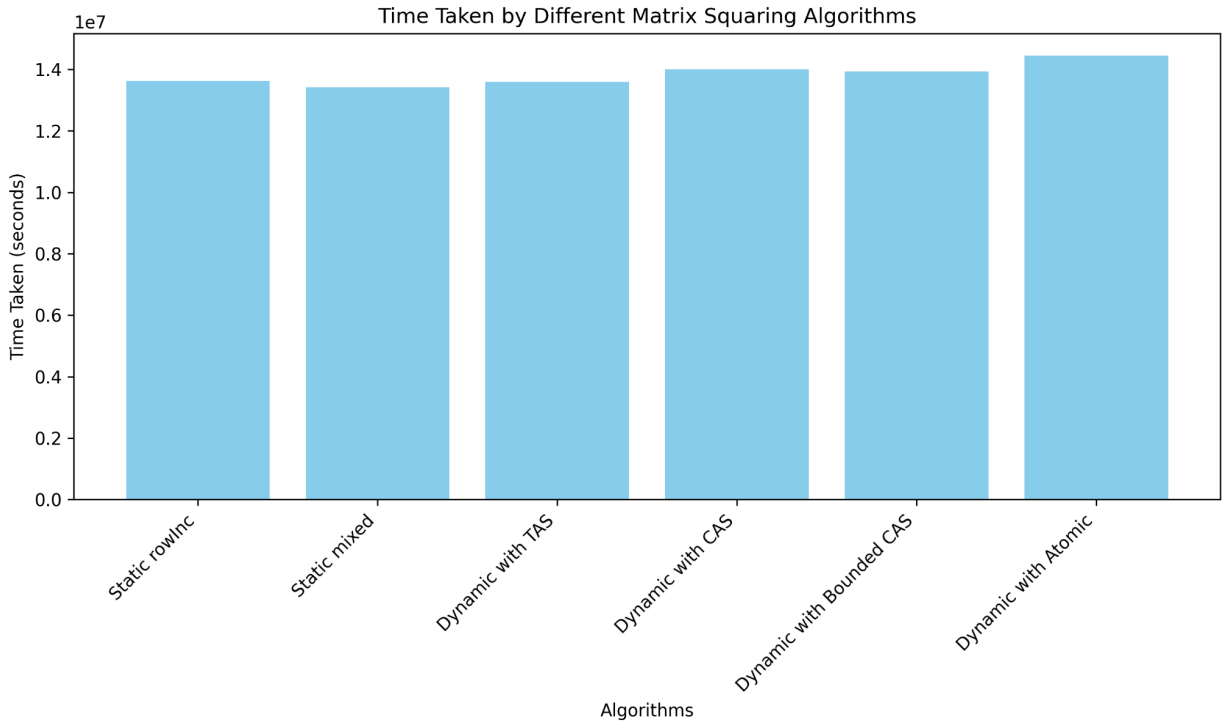
And K value changes from 1 to 32 in powers of 2.



From the graph it can be seen that as the number of threads increases the time taken to calculate the output decreases. This is because the size of matrix is fixed and as the number of threads increases the number of rows allocated to each threads decreases. As these threads run parallely the work is done simultaneous and there will be decrease in time taken.
As the number of cores is fixed , if the number of threads are more than number of cores not all run simultaneous so there wouldn't be much high difference in the time taken as the number of threads increase beyond a number.

Experiment-4

The below is the graph between time taken and the algorithm used.



From the graph it can be inferred that all the algorithms take almost similar time to compute the matrix.
Here
N=2048
K=16
rowInc=16

Complications

The time taken for computation changes from one execution to the other.This is because of the processor availability and time taken for content switch. Hence to find the time taken I have averaged over 5 executions and taken that time.