CS2523

OPERATNG SYSTEMS – II

Programming Assignment – 3

By: Devraj (ES22BTECH11011)

DESIGN OF THE PROGRAMS

Generating the inputs

For generating the inputs required for the programs, I have created a program named ***inputGenerator()***;

It depends on the parameter ***n***, upon which it creates a randomly generated ***n*n*** matrix having numbers from 1 to 1000 (can be changed).

It prints ***n***, ***k***, ***rowInc*** and the matrix in order onto the input file to be used by the main programs.

Common things in the main programs

The main programs read the parameters ***n***, ***k*** and ***rowInc*** from the input file, and then dynamically allocates memory for the two ***n*n*** matrices. Afterwards, they read the values of the matrix from the input file and put it into the 2-D matrix ***A***. Then they create the transpose of ***A*** and store it into the 2-D matrix ***A_trans***.

The critical section consists of : incrementing the global counter ***c*** and setting the value ***rowStart*** equal to ***c***. This critical section is present inside the ***runner()*** function.

In each instance of its execution, i.e, when the thread is not waiting for the lock to be freed, it computes the rows having index: ***c*** through ***c + rowInc-1***.

In each thread, the ***runner()*** function runs till the value of ***c*** becomes equal to ***n***. It exits the while loop and the thread finishes.

The ***dot()*** function gives the dot product of two 1-D arrays. It is used to find the elements of ***A_sq*** by multiplying a row of ***A*** with a column of ***A_trans***.

To implement multithreading, they create an array of threads, with the thread at each index having a unique set of arguments.

The atomic components of the program are implemented using the constituents of the ***<atomic>*** header file.

The time required to perform the row computations is counted using the ***chrono*** header file.

Then the output matrix ***A_sq*** is printed onto the output file.

The files are closed and the memory for the arrays is deallocated and the programs end.

Mutual Exclusion Algorithms

Test and Set (TAS)

A variable "***lock***" of the type ***atomic_flag*** is created.

It is initialized with ATOMIC_FLAG_INIT.

The function ***test_and_set()*** provided by the C++ library in the ***<atomic>*** header file is used to implement this algorithm.

It atomically changes the state of a ***atomic_flag*** variable to set it to ***true(1)*** and returns the value it held before.

So the while loop runs till the value returned by the function is ***false(0)***, i.e., when the lock has been freed by another thread. After acquiring the lock, the thread executes its critical section and calls ***clear()*** to set lock to ***false(0)***, thus freeing the lock.

Compare and Swap (CAS)

A variable "***lock***" of the type ***atomic_int*** is created.

It is initialized to ***0***.

The function ***compare_exchange_strong()*** is used to implement the algorithm.

It takes an ***expected*** value and a ***desired*** value as arguments. If the value of ***lock*** is the same as the ***expected*** value it sets the value of the ***lock*** to the ***desired*** value and returns ***true***. Otherwise, it sets the value of ***expected*** to the actual value and returns ***false***.

Since we always need the ***expected*** value of the ***lock*** to be ***0***, we have to set the value of ***expected*** to ***0*** after each instance of execution of the function.

The thread waits while the value held by ***lock*** is ***1*** (the function returns ***false***) and enters its critical section when the value held by ***lock*** becomes ***0*** (done by another thread).

The thread increments the value of ***c*** by ***rowInc*** and sets ***rowStart*** equal to ***c***. Then the thread sets the value of ***lock*** to ***0*** (frees the lock).

Compare and Swap with Bounded Waiting

The implementation is the same as that of CAS, except for the implementation of bounded waiting.

The use of bounded waiting is to ensure that a thread doesn't starve, i.e., it has a fair chance of being able to execute its critical section.

A Boolean array ***waiting[k]*** is created to store the wait status of each thread.

A Boolean variable ***key*** is initialized to true.

A thread can enter its critical section only either when the value of ***key*** is false, i.e. when the function ***compare_exchange_strong()*** returns ***true*** (value of ***lock*** is ***0***), or when ***waiting[i]*** (***i*** corresponds to the current thread) is ***false***.

In each execution of the critical section of a thread, we check the ***waiting*** array for any threads that are waiting. The first thread that is found to be waiting has its ***waiting*** value set to ***false***, allowing it to enter its critical section. If no such thread is found, the value of ***lock*** is set to ***0***.

Atomic Increment

To implement atomic increment we make the global counter ***c*** of the type ***atomic_int***.

The function ***fetch_add()*** is used to increment ***c*** atomically. It returns the value held by ***c*** prior to the increment and increments ***c*** by ***rowInc***. These two instructions are executed atomically, i.e., all at once.

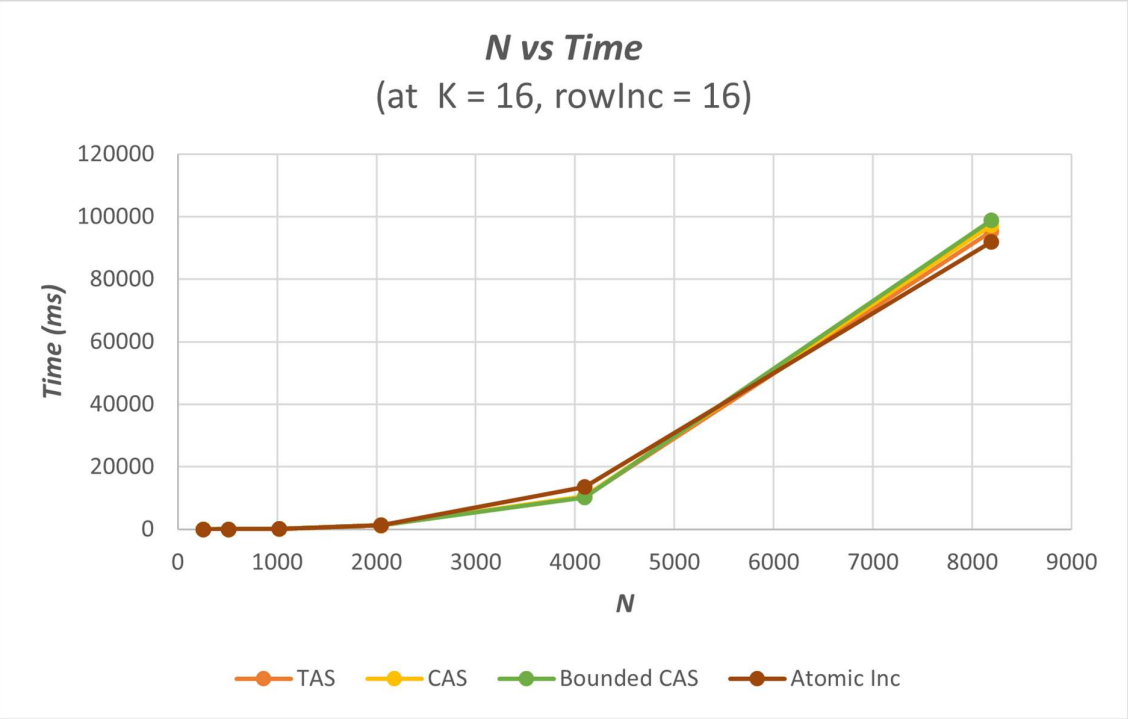
The variable ***rowStart*** has its value set to ***c*** and the value of ***c*** is incremented atomically, without interruption from other threads.

Performance of the programs

Experiment 1: Time vs Size, N

N vs Time(ms)

N	TAS	CAS	Bounded CAS	Atomic Inc
256	4.92	5.21	5.42	5.45
512	25.8225	24.11	25.4368	22.46
1024	161.554	162.54	166.779	163.329
2048	1204.89	1319.9	1324.7	1297.81
4096	10335.6	10527.3	10270	13543.2
8192	95181	97328	98918	91917



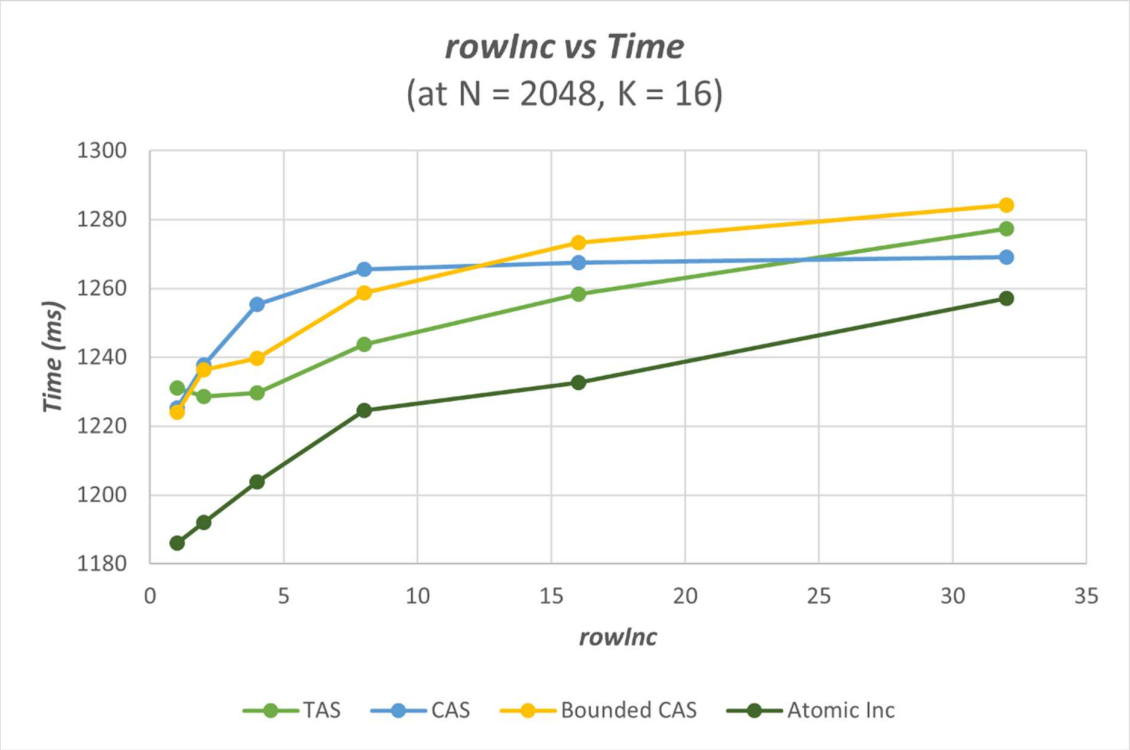
As we can see, the performance of the algorithms is almost the same with ***Atomic Increment*** performing slightly better for larger inputs. This could be because the number of lines of code required to implement it is lower than that of other algorithms, i.e., the number of instructions (like acquiring and releasing the lock) is less leading to less time of execution.

Bounded CAS gives the worst time. This could be due to the large number of instructions in this algorithm to ensure that no thread starves.

For all the algorithms, the time for execution increases exponentially with the size of the input, ***N***.

Experiment 2: Time vs rowInc

rowInc vs Time(ms)				
rowInc	TAS	CAS	Bounded CAS	Atomic Inc
1	1231.11	1225.28	1224.11	1186
2	1228.7	1237.73	1236.42	1192
4	1229.72	1255.37	1239.7	1203.83
8	1243.67	1265.55	1258.77	1224.6
16	1258.37	1267.41	1273.21	1232.6
32	1277.38	1269	1284.27	1257.2

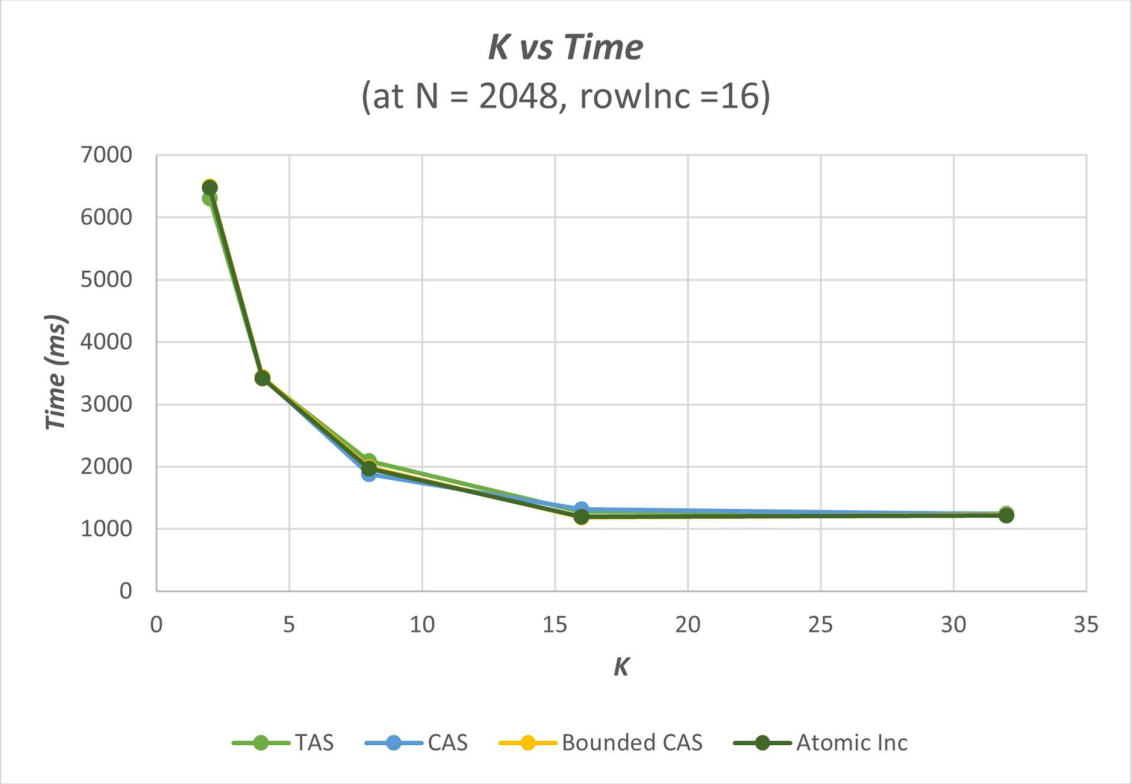


In the case of this graph, the ***Atomic Increment*** algorithm performs considerably well compared to the other algorithms.

For all the algorithms, the time for execution increases by small amounts as we increase the value of ***rowInc***. This could be due to reduced parallelism as the size of the chunks allocated to the threads becomes larger, each instance of execution of a thread also becomes larger, so some parallelism is lost.

Experiment 3: Time vs Number of threads, K

K vs Time(ms)				
K	**TAS**	**CAS**	**Bounded CAS**	**Atomic Inc**
2	6306.21	6479.35	6492.33	6480.36
4	3415	3434.35	3435.67	3416.77
8	2086.89	1882.58	1988.47	1970.79
16	1276.43	1320.88	1185.93	1200.03
32	1243.92	1235.7	1225.28	1212.13

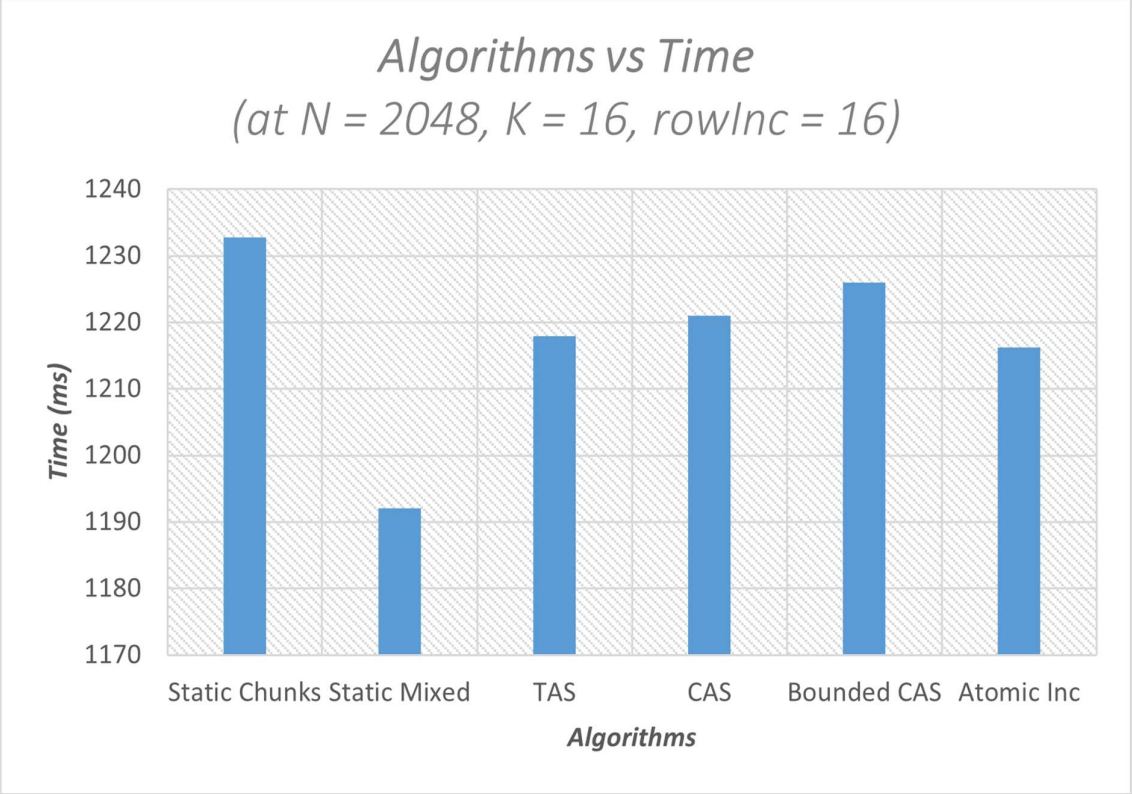


In this graph, the time for execution decreases exponentially with increase in the number of threads, ***k***. This is because an increase in the number of threads allows the processors to perform more tasks in parallel giving better execution times.

All the algorithms give almost the same time for execution, with ***Atomic Increment*** giving slightly better time for higher ***k***.

Experiment 4: Time vs Algorithm

Algorithm vs Time(ms)						
Algorithm	Static Chunks	Static Mixed	TAS	CAS	Bounded CAS	Atomic Inc
Time(ms)	1232.76	1192	1217.87	1221	1226	1216.22



As we can see, the ***Static Mixed*** algorithm gives considerably better time among all the given methods. This could be because in the static algorithms the work to be done by a thread is predetermined, meaning less decisions made in the runtime.

Among the dynamic methods, ***Atomic Increment*** performs slightly better than others. As explained before, this could be due to a smaller number of instructions(like acquiring and releasing the lock) in the ***Atomic Increment*** algorithm compared to others.