**Operating Systems- 2**

**Assignment 3 Report**

Varun Gopal

CO22BTECH11015

In this assignment we use dynamic thread implementation to improve the performance of the matrix squaring algorithm over the previous static multi-threaded algorithms.

This program is implemented using the creation of multiple threads and assigning some of the rows of resultant matrix to the threads which currently have no work. To maintain synchronisation between the threads we use the variable ‘counter’ to account for the number of rows computed in the output matrix. For synchronisation of memory and eliminating race conditions, we implement mutual exclusion by Test and Set, Compare and Swap, Bounded-Compare and Swap and Atomic increment methods.

**Explanation of Code:**

First ‘N’ (Number of rows), ‘K’(Number of threads), ‘rowinc’(The number of threads assigned to each work-free thread) and the corresponding N\*N matrix (a) are read from the input file(inp.txt). Now we find the square of the matrix using matrix multiplication. We use a function named dot frequently which computes the dot product of the row and the column passed to it. To do this We first create an array to store the ids of the K threads. These are used by the main function to identify each thread and wait till all the threads finish their execution.

Now we create K Pthreads at the beginning of the main program. We use the variable ‘counter’ to store the number of rows which had been computed in the output matrix. Whenever a thread gets free it assigns itself rowinc threads starting from counter. We check the final condition if counter > n then we exit out of all the threads.

The above process is repeated for all the threads

**Analysis of Output:**

The running times of the program were measured by varying K, N,rowinc for each execution using the 4 methods. It is a static multi-threaded program and assigning the rows by the threads themselves. This method helps to reduce the time slightly by improving CPU utilisation and balancing the work much more efficiently than the in built cpu scheduling. Previously the threads were assigned rows by static methods. Now by dynamic allocation we improve the performance of the code by uniformly balancing the code.

Below is the analysis of how the running time of the program changes with variations in N for the 4 methods: -

Observations for each experiment:

Experiment 1: Time vs. Size (N)

- With the increase in matrix size (N), there is a noticeable rise in execution time across all mutual exclusion algorithms.

- As the number of rows increases, the time taken to compute the output matrix increases.

- As matrix size grows, the performance disparity between mutual exclusion algorithms becomes more evident, with Bounded CAS and Atomic Increment often outperforming TAS and CAS.

Experiment 2: Time vs. rowInc

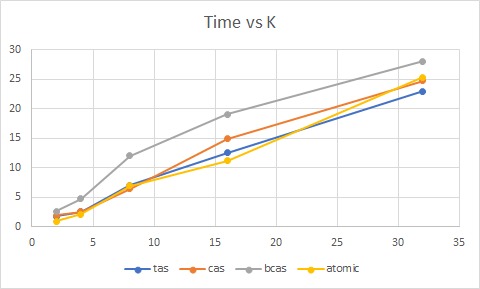
- The variance in row increment (rowInc) from 1 to 32, following powers of 2, highlights the influence of row increment size on execution time.

- Smaller row increments generally lead to shorter execution times due to more parallelism, enabling threads to handle smaller data chunks more efficiently.

Experiment 3: Time vs. Number of threads (K)

- The upward trend in execution time reduction with an increase in the number of threads (K) indicates enhanced parallelism and distribution of computational workload.

- Nevertheless, The performance of the code decreases, as overhead related to thread creation and synchronization begins to outweigh the benefits of parallelism.



Experiment 4: Time vs. Algorithms

- Bounded CAS and Atomic Increment tend to showcase superior performance compared to TAS and CAS in terms of execution time.

**Summary:-**

The dynamic matrix squaring program demonstrates the effectiveness of parallel processing and mutual exclusion algorithms in optimizing computation time. Overall, the program provides valuable insights into parallel computing and optimization techniques for matrix operations.