# HW Programming Assignment 1 - Matrix Multiplication

CS18BTECH11042

I certify that this assignment/report is my own work, based on my personal study and/or research and that I have acknowledged all material and sources used in its preparation, whether they be books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication. I also certify that this assignment/report has not previously been submitted for assessment in any other course, except where specific permission has been granted from all course instructors involved, or at any other time in this course, and that I have not copied in part or whole or otherwise plagiarised the work of other students and/or persons. I pledge to uphold the principles of honesty and responsibility at CSE@IITH. In addition, I understand my responsibility to report honour violations by other students if I become aware of it.

Name: Shreyas Jayant Havaldar
Date: 20-01-2020
Signature: S.J.H

- ## Solution Implementation:

  ### Common to all 3 approaches:

  - If the no of columns in A are not equal to no of rows in B, the program exits as failure printing "Matrix multiplication not possible\n" on *stderr.*
  - Likewise an error message with correct usage printed in case of incorrect argument format.
  - If the argument format is correct, the program proceeds and retrieves the no of available processing units = *npros* using the *get_nprocs()* command.
  - The no of threads and processes produced is equal to *nrt*nct* where each of *nct* and *nrt* is initialized to square root of *npros-1.*
  - Thus maximum processes/threads produced does not exceed *npros - 1.*
  - This minimises the idle time wasted in context switching if any.
  - It's taken care of to ensure the maximum threads/processes across the row of *A* and across the column of *B* does not exceed *arows* and *bcols* respectively.
  - **Note:** All the matrices *A, B, C* are stored and operated upon in row major order.
  - All the memory allocation (in heap for *single_thread_mm()* and *multi_thread_mm()* and in shared memory for *multi_process_mm()*) and checking

whether the --*interactive* argument is true, and computing and calling the output function happens in the individual functions.

- The time taken for computation is returned as an *int.*
- **Note:** I have used a user defined function *getNanos()* to calculate time precisely in nanoseconds and then converted to microseconds.

## Single Threaded:

- The no of iterations of the innermost loop *arows * bcols * acols.*
- The time taken is considered base case for measuring speedup.

## Multi Threaded:

- An array of thread nos is created as each element is passed as an argument to the *matrix_multi_t() function.*
- 2 semaphores are used to ensure all the threads are created before computation is started in any thread.
- *t_count* maintains the no of threads active at the moment and is manipulated only in the critical section locked by the semaphores to prevent synchronization issues.
- *tc* is a counting semaphore and *mutex* is a binary semaphores initialised to 1 used to achieve the same.
- Only then is the start time registered.
- So only the time taken for computation is measured.
- The thread id is passed to the function *matrix_multi_t()* to be able to assign tiles to each of the *nrt * nct* threads to calculate.

## Multi Process:

- An array of shared memory id's is created.
- Each matrix is assigned its own shared memory.
- *stdout* is flushed before forking is done to prevent printing of items in buffer of *stdout* to be carried from the parent process.
- *matrix_multi_p()* is called passing the loop variable in every child process to be able to assign tiles to each of the *nrt * nct* processes to calculate.
- Each child process exits with 0 status.
- The parent process waits for every child process to terminate before it ends measuring the time.

- Speed Up Observations:
    - The matrices A and B were tiled across their rows as well as their columns to distribute them into different threads/processes to achieve fastest computation.
    - This horizontal as well as vertical tiling ensured that the cases in which the input matrices had a huge number of rows and columns were taken care of with equal importance.
    - The most optimal no of threads/processes for every machine is unique and to be able to gauge the processing power of the machine, the approach using *get_nprocs()* is used.
    - If the row/column size was less than the no of threads into which that direction's computation was getting divided, then the rows/columns were divided into maximum the no of rows/columns respectively.
    - So redundancy of threads/processes was overcome.
    - It is ensured that the no of threads/processes created are not excess by checking them against the no of rows in *A* and the no of columns in *B.*
    - For very small sizes of matrices *A* and *B (approx arows < 10 bcols < 10)* the time taken in single threaded computation is negligible (~0) and the time taken for multithreaded (~100 us) due to idle waiting and multiprocess computation (~1000 us) is more due to overheads like idle waiting and time taken to fork() processes in multiprocess.

**Note:** Please refer to the README.txt to check more sample outputs.
**Note:** All the execution was done on a laptop with 8 available processing units, so speed ups are accordingly.

- Speed Up Inferences:
    - The speedup achieved in multithreaded computation was consistently slightly higher than the speedup achieved in multiprocess computation due to forking overhead in multiprocess.
    - ./a.out --ar 100 --ac 100 --br 100 --bc 100
      Time taken for single threaded: 16622 us
      Time taken for multi process: 4588 us
      Time taken for multi threaded: 4126 us
      Speedup for multi process : 3.62 x
      Speedup for multi threaded : 4.03 x
    - For **squarish** matrices *A and B* of sizes around 100*100 a speedup of around 4x was achieved with time taken in single threaded to be around 16000us.
    - ./a.out --ar 1000 --ac 1000 --br 1000 --bc 1000
      Time taken for single threaded: 5219413 us
      Time taken for multi process: 1373959 us

Time taken for multi threaded: 1328743 us
Speedup for multi process : 3.80 x
Speedup for multi threaded : 3.93 x

- For squarish matrices *A and B* of sizes around 1000*1000 a speedup of around 3.9x was achieved with time taken in single threaded to be around 5000000us.
- For square matrices speedup increased steeply as matrix size increased upto 100*100 and then stagnated or even dipped occasionally when I kept on further increasing the matrix dimensions.
- This happened because the number of threads/processes that can be formed on a machine concurrently are limited, thus after a certain computation capacity it just stagnates and no better speedup is achieved.
- For very **rectangular** matrices achieving speedup is tricky as an optimal division of worker threads/processes is essential.
- ./a.out --ar 100 --ac 1001 --br 1001 --bc 1
  Time taken for single threaded: 2880 us
  Time taken for multi process: 1923 us
  Time taken for multi threaded: 1153 us
  Speedup for multi process : 1.50 x
  Speedup for multi threaded : 2.50 x
- Here there is no tiling across the column of *B* and efficient tiling along *A'* rows thus leading to a speedup.
- For ./a.out --ar 100 --ac 1001 --br 1001 --bc 1000
  Time taken for single threaded: 649486 us
  Time taken for multi process: 148810 us
  Time taken for multi threaded: 138682 us
  Speedup for multi process : 4.36 x
  Speedup for multi threaded : 4.68 x
- For the above case the speedup almost identical to the ones obtained for sqaure matrices is observed due to large size.

So it is understood that on a machine with higher computing power better speedup will be obtained as more threads/processes will run concurrently.