# CONCURRENT PROGRAMMING-ASSIGNMENT 2

# EFFICIENT LARGE MATRIX MULTIPLICATION

**ALGORITHM IN OPENMP**

|  |  |
| --- | --- |
| ***NAME*** | ***VIVEK ARYA*** |
| ***REGISTRATION NUMBER*** | ***19210337*** |
| ***COURSE*** | ***MSC COMPUTING-SOFTWARE ENGINEERING*** |
| ***YEAR*** | ***2019-20*** |

1. **DECLARATION**

**I declare that the solution source code and this word file, which I now submit for assessment, is entirely my work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references.**

1. **GIVEN PROBLEM:**

To develop an efficient large matric multiplication algorithm in open multi-processing(OpenMP), and present the evidence that substantiates that your implementations are efficient.

**3.PROPOSED SOLUTION:**

**->** I have proposed two solutions in this assignment to solve this problem of large matrix multiplication. The first solution just uses simple threads concept in OpenMP, and the second solution uses both threads and tasks concept in OpenMP to solve this problem.

**(A) SOLUTION 1 DESCRIPTION: USING OpenMP THREADS**

**-> FILENAME: MatrixMultiplicationUsingThreads.C**

**->** Threads are an independent execution stream that queues up in a ready queue. Threads are then picked up by processors for execution. A thread may preempt many times in its lifetime.

**->** Given below is the matrix multiplication logic that is used, when a matrix A of m rows and n columns is multiplied with another matrix B of n rows and p columns, to get a matrix C of m rows and p columns.

**for(i=0 ; i<No\_Of\_matrix\_A\_row ; i++)**

**{**

**for(j=0 ; j<No\_Of\_matrix\_B\_columns ; j++)**

**{**

**matrixC [i] [j] = 0;**

**for(k=0; k<No\_Of\_matrix\_A\_Columns; k++)**

**{**

**matrixC[i][j]=matrix[i][j]+matrixA[i][k]\*matrixB[k][j];**

**}**

**}**

**}**

**->**Basically, to create threads in OpenMP we use the below directive:

**#pragma omp parallel**

**{**

**…………**

**}**

**->**But, as we have here for loops, we can make use of the "for" directive of OpenMP.

**->**Normally, a "for" directive of the OpenMP allows you to distribute the iterations of the for loop to the threads that are created using #pragma omp parallel .

**->**But, as we have a nested for loop here in the logic, I have made use of "collapse" clause along with the "for" directive, such that each thread will now be responsible to generate the a(i,j) element of matrix C i.e the resultant matrix.The syntax to do so is:

**#pragma omp for collapse(2)**

**for(…){**

**for(...){**

**for(…){**

**}**

**}**

**}**

**->**This collapse(2) clause clubs the two immediate loops together, and then divides them among the threads. So each thread will get an (i,j) pair to work on, i.e a row of Matrix A (i here) and a column of Matrix B (j here), which will then carry out the operation of the 3rd loop which is:

**for(k=0 ; k < No\_Of\_MatrixA\_Column ; k++)**

**{**

**MatrixC[i][j]= MatrixC[i][j]+MatrixA[i][k]\* MatrixB[k][j]**

**}**

And we thereby get our MatrixC's [i][j]th element where i is the row and j is the column.

**(a) EVIDENCE OF THE EFFICIENCY OF THIS APPROACH**

**->** A sequential approach is always better for matrices of small size as there is no overhead on the operating system for creating a new thread, preempting and maintaining them throughout the execution of the program.

**->** But, where there is a large number of computations required, we can rather create a separate thread for each computation if one computation does not depend on the other.

**->** As in this method, every thread is given one row of matrix\_A (i) and one column of matrix\_B (j) to evaluate the corresponding element Matrix\_C(i,j). This is just like a multiplication over two vectors, which is independent.

|  |  |  |  |
| --- | --- | --- | --- |
| A[i][j] | .. | .. | .. |
| .. | .. | .. |  |
| .. | .. | .. | .. |
| .. | .. | .. | A[i][j] |

|  |  |  |  |
| --- | --- | --- | --- |
| B[i][j] | .. | .. | .. |
| .. | .. | .. | .. |
| .. | .. | .. | .. |
| .. | .. | .. | B[i][j] |

**J row of B is given to say thread1**

|  |  |  |  |
| --- | --- | --- | --- |
| C[i][j] | .. | .. | .. |
| .. | .. | .. | .. |
| .. | .. | .. | .. |
| .. | ,, | .. | C[i][j] |

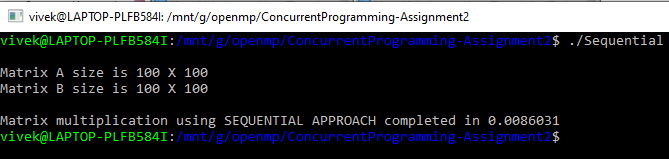
**I row of A is given to say thread1**

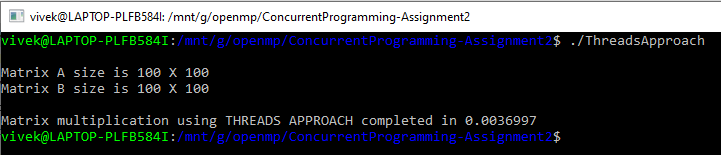
**->**Thread1 now takes I row of A and J column of B and performs the computation to give rise to C[I][J] element of C.

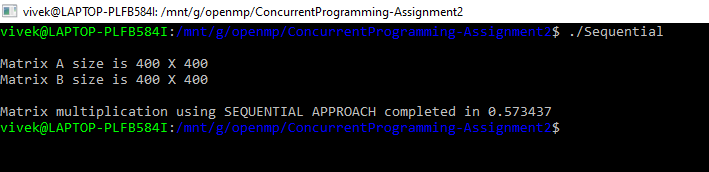
* **RESULTS**

Below are the snapshots of the result achieved from a matrix multiplication using a sequential approach and the threads approach (the proposed solution) , please see that the Sequential approach is highlighted in green, and my solution which is using threads is highlighted in yellow:

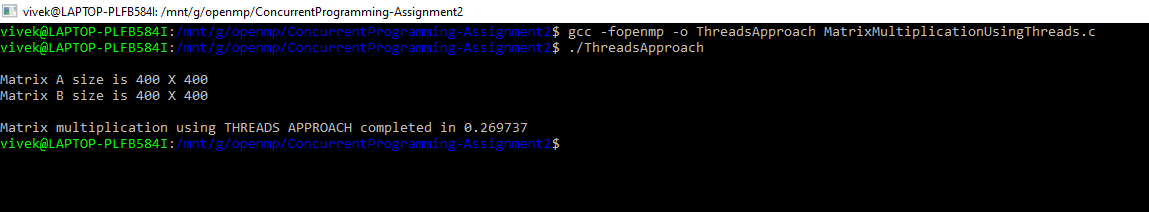
**(i) Multiplication of two 100 X 100 matrices using sequential approach**



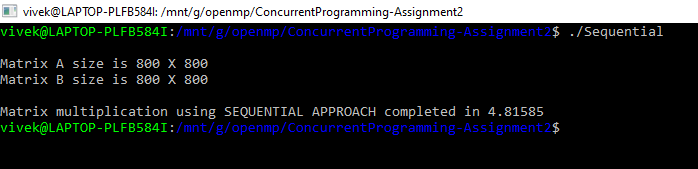
**(ii) Multiplication of two 100 X 100 matrices using threads (Proposed solution 1)** 

**(iii) Multiplication of two 400 X 400 matrices using sequential approach** 

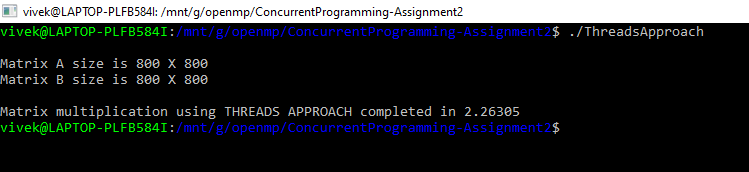
**(iv) Multiplication of two 400 X 400 matrices using threads (Proposed solution 1)**



**(v) Multiplication of two 800 X 800 matrices using sequential approach**



**(vi)Multiplication of two 800 X 800 matrices using threads (Proposed solution 1)**



**OVERALL OBSERVATION:**

|  |  |  |  |
| --- | --- | --- | --- |
| *S.NO* | *MATRIX DIMENSION* | *TIME TAKEN USING SEQUENTIAL APPROACH* | *TIME TAKEN USING THREADS APPROACH* |
| *1.* | *100 X 100* | *0.0086031 sec* | *0.0036997 sec* |
| *2.* | *400 X 400* | *0.573437 sec* | *0.269737 sec* |
| *3.* | *800 X 800* | *4.81585 sec* | *2.26305 sec* |

**(B) SOLUTION 2 DESCRIPTION:USING OpenMP THREADS AND TASKS**

**-> FINLENAME: MatrixMultiplicationUsingThreadsAndTask.c**

**->** In this solution, I have made use of the task concept in OpenMP along with the threads in OpenMP.

**->** Here, the last loop in the logic of matrix multiplication is converted into a task, which can then be completed by any thread that is free to complete this task.

**->** The task is lighter than threads, they are just present in a task queue, and are picked by threads which complete those tasks.

**->** The best thing about this logic is there is no restriction on threads that the task must be completed by only the thread that creates it.

**->** Here comes the concept of depend clause. A depend clause is used to convey that the particular memory block is supposed to be output with a value, hence an implicit lock is applied by the OpenMP to that location whenever a thread has executed a task and now has to write to that location.

**->** Below is the logic that makes every thread to create tasks:

**#pragma omp for collapse(2)**

**for(…){**

**for(...){**

**#pragma omp task depend(out:matrixC[i][j])**

**for(…){**

**}**

**}**

**}**

**->**The #pragma omp for collapse(2) gives every thread an Ith row of matrix a and Jth column of matrix B to produce the corresponding (I,J) element of matrix C. But, the new directive #pragma omp task depend(out:matrixC[i][j]) highlighted in the above box makes every thread to produce a task where the task is the code below the final loop, that contains the actual operation of matrix multiplication logic.

**->** The depend(out:matrixC[i][j]) ensures that the writing to the matrixC[i][j] location is atomic only to avoid the happening of any race conditions.

**->** This method is as effective as using just the threads, but we can see better and consistent performance when the size of the matrices upon which the multiplication has to be carried is large.

**(a) EVIDENCE OF THE EFFICIENCY OF THIS APPROACH**

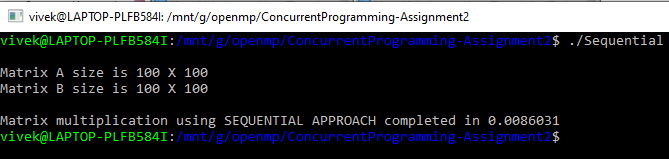
**->** Threads may preempt at any time in their life span. Creating tasks ensures that at least some threads are completing the tasks no matter what.

**->** This ensures that the tasks will be completed in a much faster way.

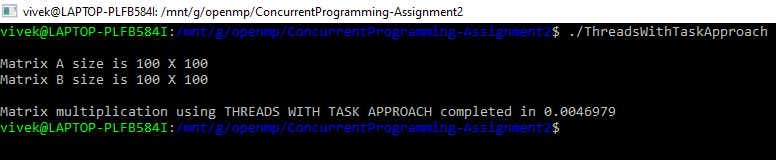
**->** A thread is always allowed to complete a task created by some other thread, which leaves very little scope for completion of the total operations required in matrix multiplication.

**->** To prove the efficiency of this solution, I have mentioned pasted below screenshot from the sequential approach, threads approach and threads with tasks approach.

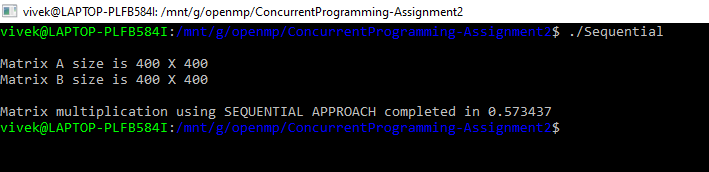
**(i) Multiplication of two 100 X 100 matrices using sequential approach**



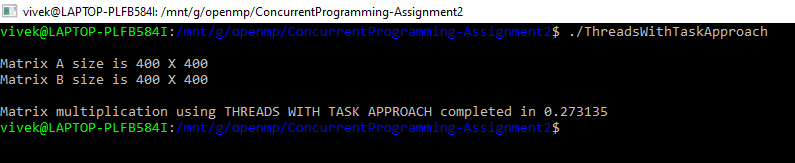
**(ii) Multiplication of two 100 X 100 matrices using threads with task (Proposed solution 2)**



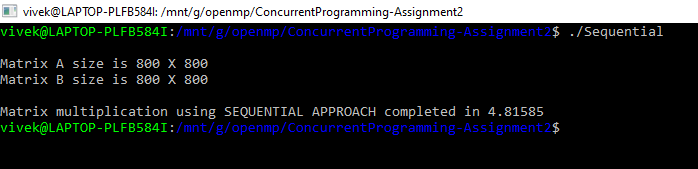
**(iii) Multiplication of two 400 X 400 matrices using sequential approach**



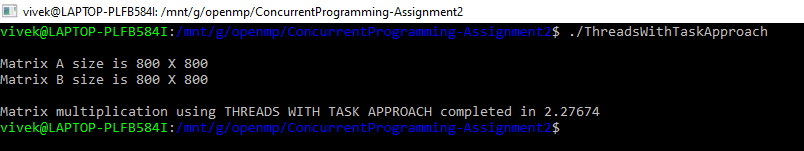
**(iv) Multiplication of two 400 X 400 matrices using threads with task (Proposed solution 2)**



**(v) Multiplication of two 800 X 800 matrices using sequential approach**



**(vi)Multiplication of two 800 X 800 matrices using threads with task(Proposed solution 2)**



**OVERALL OBSERVATION:**

|  |  |  |  |
| --- | --- | --- | --- |
| *S.NO* | *MATRIX DIMENSION* | *TIME TAKEN USING SEQUENTIAL APPROACH* | *TIME TAKEN USING THREADS WITH TASK APPROACH* |
| *1.* | *100 X 100* | *0.0083166 sec* | *0.0046979 sec* |
| *2.* | *400 X 400* | *0.573437 sec* | *0.273135 sec* |
| *3.* | *800 X 800* | *4.81585 sec* | *2.27674 sec* |

1. **REFERENCES:**

**<i>** <https://www.youtube.com/channel/UCNp-uk36t-bnvHr3A_snQtg/featured>

**<ii>**<https://en.wikipedia.org/wiki/Matrix_multiplication>

**<iii>**<https://www.computing.dcu.ie/~davids/courses/CA670/CA670_OpenMP_2p.pdf>