**Overview:**

Matrix multiplication is one of the most basic operations in computer science. The naïve approach for large matrix multiplication is not optimal and required O(n3) time complexity. OpenMP allows us to compute large matrix multiplication in parallel using multiple threads. In this assignment I have used block based tilling approach and matrix transpose approach for efficient computation.

**Use of OpenMP:**

Using OpenMP in this program we converted sequential matrix multiplication code into multi-threaded code.

* A **Private variable scope** is used in program, which allows each thread to have its own private copy of a variable.
* As we have nested for loops in our program, use of **collapse** directive helps us achieve more efficiency by collapsing nested loops into single loop and parallelize the resulting loop.
* The **parallel** construct creates a team of thread and starts parallel execution. The **parallel for** specifies that the block of code will be iterated in parallel by threads present in team of threads.
* The **critical** construct restricts the execution of code block to single thread at a time. In this program, the actual arithmetic calculation of matrix multiplication is placed inside critical section.

**Efficient Matrix Multiplication approaches:**

Matrix are divided into **blocks** in order to improve temporal locality of inner loops while multiplication. Matrix 1 and Matrix 2 are divided into blocks of size 100. So, while actual multiplication only the block of data will be sent to process rather than whole large matrix. Both below described techniques use blocking approach of matrix multiplication.

**1. Loop tiling:**

Loop tiling is Loop Nest Optimization (LNO) technique which is used for locality optimization or parallelization. It is used to reduce memory access latency for common linear algebra algorithms.

Loop tiling partitions the loop iteration space into smaller blocks in order to store data in cache until it is reused. Large matrix arrays are divided into smaller blocks by partitioning the loop iteration space. Due to this partitioning, size of cache required is also reduced and thus these small blocks could be fitted in smaller cache enhancing cache reuse (Wikipedia, n.d.)

**2. Matrix transpose:**

If the matrix is stored in row in RAM, then it is slow to read the columns because we are not using most of cache lines. To overcome this problem, we transpose one of the matrices, that is we swap row data as column and column data as row. Now while performing the matrix multiplication, we are using the entire cache lines, and hence get better performance gain.

**Program flow:**

* Two matrices: Matrix 1 and Matrix 2 of different sizes (500,1000 and 1500) are created by create\_matrix() function in the program. This function generates random numbers and fills the matrix with these values. Initially memory is allocated for the matrix with specified size.
* get\_clock() function is used to determine the time required by multiplication algorithm to compute the result.
* In **loop tiling** approach i.e. in tiled\_matrix\_multiplication() function we consider only a block of matrix (block size: 100) at a time for computation and iterate over it. After each element is visited in that block, we increment the for loop by specified block size. (chang, 2014)
* In **transpose matrix multiplication with blocking** approach i.e. in block\_matrix\_mul\_transposed() function, we consider multiplication of Matrix 1 and transpose of Matrix 2. We partition the large matrix into submatrices with specified block size (block size: 100) for efficient computation.

**Results:**

Results that is computation time depends on the system configuration, hence below are the specifications of the system on which program was tested

**Processor:** Intel® Core™ i5-8300H

**L3 Cache:** 8MB

**Matrix Size: 500\*500**

|  |  |  |  |
| --- | --- | --- | --- |
| Approach |  | Number of Threads |  |
|  | **2 Threads** | **4 Threads** | **8 Threads** |
| Block Loop Tiling | **0.28 sec** | **0.26 sec** | **0.25 sec** |
| Block Matrix Transpose | **0.30 sec** | **0.29 sec** | **0.28 sec** |
| Sequential | **1.32 sec** | **1.30 sec** | **1.25 sec** |

**Matrix Size: 1000\*1000**

|  |  |  |  |
| --- | --- | --- | --- |
| Approach |  | Number of Threads |  |
|  | **2 Threads** | **4 Threads** | **8 Threads** |
| Block Loop Tiling | **1.90 sec** | **1.85 sec** | **1.7 sec** |
| Block Matrix Transpose | **2.49 sec** | **2.30 sec** | **2.2 sec** |
| Sequential | **17.40 sec** | **17.64 sec** | **17.50 sec** |

**Matrix Size: 1500\*1500**

|  |  |  |  |
| --- | --- | --- | --- |
| Approach |  | Number of Threads |  |
|  | **2 Threads** | **4 Threads** | **8 Threads** |
| Block Loop Tiling | **6.70 sec** | **6.68 sec** | **6.60 sec** |
| Block Matrix Transpose | **6.60 sec** | **6.81 sec** | **6.68sec** |
| Sequential | **69.60 sec** | **70.05 sec** | **69.64 sec** |

# **References**

Wikipedia, n.d. *Wikipedia.* [Online]   
Available at: https://en.wikipedia.org/wiki/Loop\_nest\_optimization  
[Accessed 13 4 2020].

chang, c., 2014. *Github.* [Online]   
Available at: https://github.com/chunkaichang/OpenMP-MatrixOP/blob/master/tiled.c  
[Accessed 13 04 2020].