CS231P: Homework 4 – Group 14

Name	UCINetID	StudentID
Parth Shah	parths4	54809514
Ritwick Verma	ritwickv	83006696

1. Computation progress for 6 nested loop cases on single processor:

i) **i, j, k-** In this case, i and j remain fixed while the innermost loop of k advances the fastest. With k changing the fastest, each element in ith row of A is multiplied to its corresponding element in jth column of B and added to C(i, j). Elements in matrix C are filled row-wise and once the innermost loop of k ends, it means we have performed all calculations required for that element in C and that value is never updated again. We repeat the procedure by accessing all the different columns (increment j) in B and then update the row (increment i) in A.

ii) **i, k, j-** Here, j is the innermost loop and i is the outermost. This means that we will traverse the entire row in B and correspondingly keep updating the row in C. For the entire innermost loop we will only access a single value in A. Now as the value of k gets updated the above process will be repeated and for each value of k, entire row in C will get updated. Note that in updating the value of k, we will be accessing the different rows in B but the row we update in C remains the same. Once the loop of k ends, we will move to the next row (increment i) in C and the values in previous row won't be changed thereafter.

iii) **j, i, k-** This case is pretty similar to the i, j, k case. Even here the innermost loop of k advances the fastest as j and i remain fixed. The difference here is that instead of filling up matrix C row-wise we fill it column-wise. We perform all calculations for an element in C as the loop of k progresses and once loop of k ends we never update that element again. We repeat the procedure by accessing all the different rows in A (increment i) and then update the column (increment j) in B.

iv) **j, k, i-** Here, i is the innermost loop and j is the outermost. This means that we will traverse the entire column in A and correspondingly keep updating the column in C. For the entire innermost loop we will only access a single value in B. Now as the value of k gets updated the above process will be repeated and for each value of k, entire column in C will get updated. Note that in updating the value of k, we will be accessing the different columns in A but the column we update in C remains the same. Once the loop of k ends, we will move to the next column (increment j) in C and the values in previous column won't be changed thereafter.

v) **k, i, j-** Here k being the outermost loop, we will traverse the entire matrix C in row-wise order. In each iteration of the innermost loop a single value of A will be fixed and the entire row in B will be traversed. Once the innermost loop j ends, we will move to the next row in A and repeat the procedure. In each iteration of k, each element in C will be updated once.

vi) **k, j, i-** Here k being the outermost loop, we will traverse the entire matrix C in column-wise order. In each iteration of the innermost loop a single value of B will be fixed and the entire column in A will be traversed. Once the innermost loop i ends, we will move to the next column in B and repeat the procedure. In each iteration of k, each element in C will be updated once.

2. Computation progress for 6 nested loop cases on n-processors:

We use the 'Mixed Row-Column Algorithm' discussed in class to reduce the space complexity by storing a part of matrix in each of the processor.

(1) i, j, k
Since the outermost loop is i, we parallelize the computation using i. The i^{th} processor is responsible for calculating the i^{th} row in C and contains i^{th} row in A and i^{th} column in B. Once all computations are performed, every processor sends its current column of B to the next processor.

(2) i, k, j
The i^{th} processor is responsible for calculating the i^{th} row in C and contains i^{th} row in A and i^{th} row in B. Once all computations are performed, every processor sends its current row of B to the next processor.

(3) j, i, k
Since the outermost loop is j, we parallelize the computation using j. The j^{th} processor is responsible for calculating the j^{th} column in C and contains j^{th} row in A and j^{th} column in B. Once all computations are performed, every processor sends its current row of A to the next processor.

(4) j, k, i
The j^{th} processor is responsible for calculating the j^{th} column in C and contains j^{th} column in A and j^{th} column in B. Once all computations are performed, every processor sends its current column of A to the next processor.

(5) k, i, j
The k^{th} processor stores the intermediate matrix values in C calculated from the k^{th} columnS in A and k^{th} row in B. Thus, the memory required for each processor is $O(n^2)$ and the whole matrix is updated with new values at each iteration.

(6) k, j, i
The k^{th} processor stores the intermediate matrix values in C calculated from the k^{th} row in A and k^{th} column in B. Thus, the memory required for each processor is $O(n^2)$ and the whole matrix is updated with new values at each iteration.

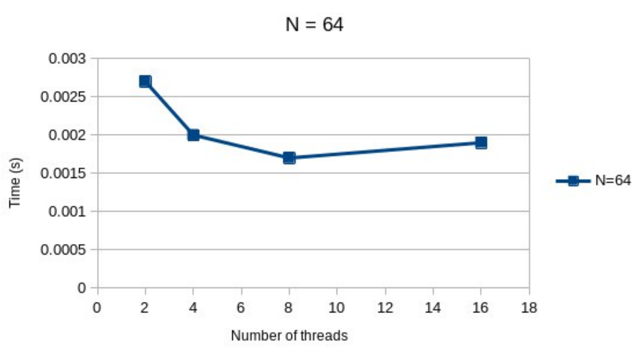
Also, note that when unfolded on i or j:

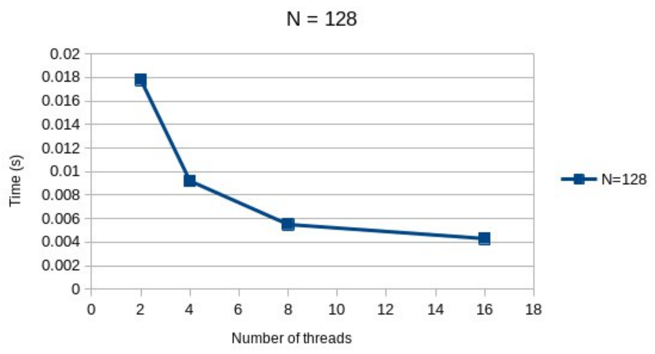
Time = $O(n^2)$
Space = $O(n)$

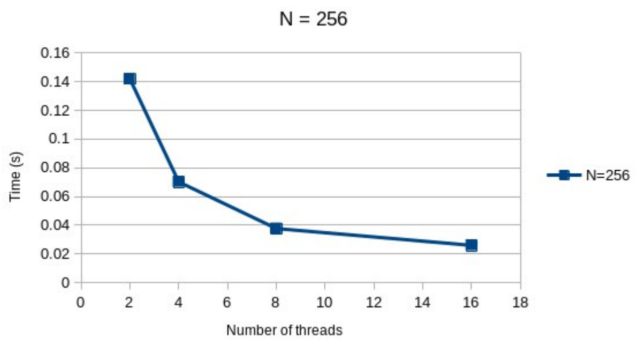
And when unfolded on k:

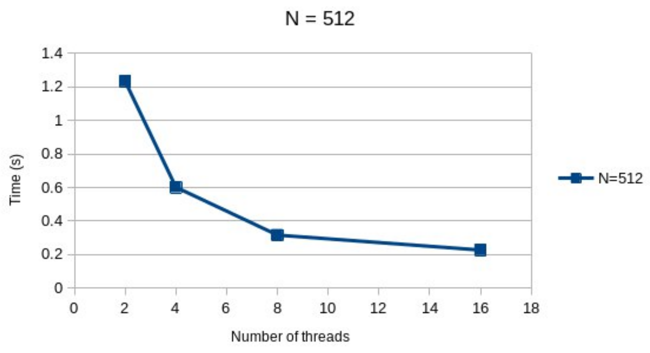
Time = $O(n^2)$
Space = $O(n^2)$

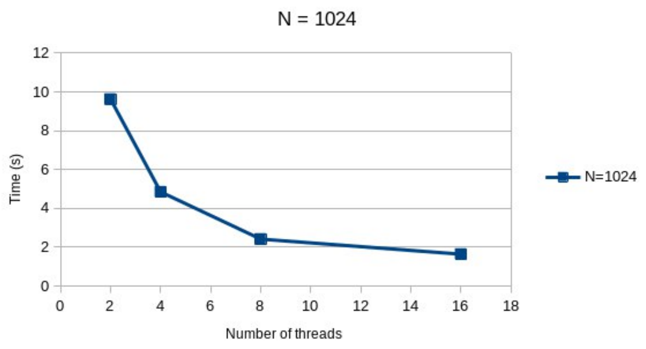
3. Performance comparison for different number of threads and matrix size.











Matrix size (N) = 64

Since for matrix size is small, computations required will be comparatively less. Hence increasing the number of threads beyond a point will cause the time to effectively increase as the cost of various thread operations like creation and destruction of thread, mutex locking unlocking operations etc. will be more than cost of computations. We can also verify this from our graph, until 8 threads the time decreases, however increasing number of threads after this point causes the time to increase.

Matrix size (N) = 128

As we can see from the graph, increasing the number of threads causes the time required to decrease exponentially. Although, we can also see that there is not a substantial decrease moving from 8 threads to 16 threads as cost of thread operations begin to cause unnecessary overhead.

<u>Matrix size (N) = 256</u>

We can observe that as number of threads increase, time decreases substantially. However, it should be noted that the time compared to matrix size =128 has increased almost 10 times, this might be because number of computations also have increased significantly.

<u>Matrix size (N) = 512</u>

Behavior is similar to previous when matrix size = 256. We can see that the time has again increased approximately 10 times.

<u>Matrix size (N) = 1024</u>

Similar to previous.