CompSci 231P HW4 Group22 Report

Student1: **Chung-Yu Chi, 22306400, chungyc7**
Student2: **Amit Somani, 94606960, somania1**

1. Since we are on a single processor computer, all of the variables are stored globally in the memory space and can be accessed by the only processor.
   (i, j, k): In the innermost loop of k, i and j are fixed. The processor reads A in row-major order and reads B in column-major order to calculate the product and updates the corresponding element in C. After one innermost loop finishes, we advance the index of the column we accessed in B (j) and continue to compute the next C element. If all columns of B are accessed, we advance one row in A (the outermost loop of i) and repeat the procedure. In this case, C is updated in row-major order. And each element of C will never be updated again as soon as the corresponding innermost loop finishes.
   (i, k, j): In the innermost loop of j, i and k are fixed. That means only an element of A is accessed during the innermost loop. Since k and j are innermost loops and also the subscript of b, B is traversed in row-major order for total n times (i=0 to n-1). When i is fixed, only the i-th row of C is updated. After all elements of B are accessed, we advance one row in A (i) and the rows before i-th row of C will never be updated again.
   (j, i, k): This case is very similar to (i, j, k) case. After one innermost loop finishes, we advance the index of the row in A (i) and continue to compute the next C element. If all rows of A are accessed, we advance one column in B (the outermost loop, *i.e.* j) and repeat the procedure to compute the next column of C. In this case, C is updated in column-major order. And each element of C will never be touched again as soon as the corresponding innermost loop finishes.
   (j, k, i): In the innermost loop of i, j and k are fixed. That means only an element of B is accessed during the innermost loop. Since k and i are inner loops and also the subscript of a, A is accessed in column-major order for total n times (i=0 to n-1). When j is fixed, only the j-th column of C is updated. After the middle loop finishes, we advance one column in B (j) and the columns before the j-th column of C will never be updated again.
   (k, i, j): In the innermost loop of j, i and k are fixed. That means only an element of A is accessed during the innermost loop. Since i and j are inner loops and also the subscript of c, C is updated in row-major order. After one innermost loop finishes, we advance one row in A (i) and continue to update the i-th row of C. After all elements of the k-th column of A is accessed, we advance k and repeat the procedure. A is accessed in column-major order, and B, C are accessed in row-major order.
   (k, j, i): In the innermost loop of i, j and k are fixed. That means only an element of B is accessed during the innermost loop. Since j and i are inner loops and also the subscript of c, C is updated in column-major order. After one innermost loop finishes, we advance one column in B (j) and continue to update the j-th column of C. After all elements of the k-th row of B is accessed, we advance k and repeat the procedure. A, C are accessed in column-major order, and B is accessed in row-major order.

2. We use n processors and each processor contains parts of the matrices to reduce storage usage like “Mixed Row-Column Algorithm” mentioned in the class.
   (i, j, k): The outermost loop is i. The i-th processor is responsible for the i-th row of C.
   P_i: contains i-th row $[a_{i0}\ a_{i1}\ ..\ a_{i(n-1)}]$, i-th column $[b_{0i}\ b_{1i}\ ..\ b_{(n-1)i}]$, produces i-th row $[c_{i0}\ c_{i1}..\ c_{i(n-1)}]$

```
Do All P_{i=0,n-1}:
  For j = current index of the column of B in P_i, do n times:
    For k = 0, n-1, do:
      c_{i,j} = c_{i,j} + a_{i,k} × b_{k,j}
    endfor
  Every processor i sends its current column of B to processor ( i + 1 ) mod ( n )
  endfor
enddoall
```

   (i, k, j): The outermost loop is i. The i-th processor is responsible for the i-th row of C.
   P_i: contains i-th row $[a_{i0}\ a_{i1}\ ..\ a_{i(n-1)}]$, i-th row $[b_{i0}\ b_{i1}\ ..\ b_{i(n-1)}]$, produces i-th row $[c_{i0}\ c_{i1}..\ c_{i(n-1)}]$

```
Do All P_{i=0,n-1}:
  For k = current index of the row of B in P_i, do n times:
```

```
  For j = 0, n-1, do:
    c_{i,j} = c_{i,j} + a_{i,k} × b_{k,j}
  endfor
  Every processor i sends its current row of B to processor ( i + 1 ) mod ( n )
 endfor
enddoall
```

(j, i, k): The outermost loop is j. The j-th processor is responsible for the j-th column of C.

P_j: contains j-th row $[a_{j0}\ a_{j1}\ ..\ a_{j(n-1)}]$, j-th column $[b_{0j}\ b_{1j}\ ..\ b_{(n-1)j}]$, produces j-th column $[c_{0j}\ c_{1j}..\ c_{(n-1)j}]$

```
Do All P_{j=0,n-1}:
 For i = current index of the row of A in P_j, do n times:
  For k = 0, n-1, do:
    c_{i,j} = c_{i,j} + a_{i,k} × b_{k,j}
  endfor
  Every processor j sends its current row of A to processor ( j + 1 ) mod ( n )
 endfor
enddoall
```

(j, k, i): The outermost loop is j. The j-th processor is responsible for the j-th column of C.

P_j: contains j-th column $[a_{0j}\ a_{1j}\ ..\ a_{(n-1)j}]$, j-th column $[b_{0j}\ b_{1j}\ ..\ b_{(n-1)j}]$, produces j-th column $[c_{0j}\ c_{1j}..\ c_{(n-1)j}]$

```
Do All P_{j=0,n-1}:
 For k = current index of the column of A in P_j, do n times:
  For i = 0, n-1, do:
    c_{i,j} = c_{i,j} + a_{i,k} × b_{k,j}
  endfor
  Every processor j sends its current column of A to processor ( j + 1 ) mod ( n )
 endfor
enddoall
```

(k, i, j): The outermost loop is k. Each processor has to keep a $O(n^2)$ memory space to store the interim result of matrix C. We call the interim result of matrix lives in k-th processor C^k.

P_k: contains k-th column $[a_{0k}\ a_{1k}\ ..\ a_{(n-1)k}]$, k-th row $[b_{k0}\ b_{k1}\ ..\ b_{k(n-1)}]$, produces matrix C^k $[c^k_{00}\ c^k_{01}..\ c^k_{0(n-1)}..c^k_{10}\ c^k_{11}\ ..\ c^k_{1(n-1)}..c^k_{(n-1)0}c^k_{(n-1)1}..c^k_{(n-1)(n-1)}]$

```
Do All P_{k=0,n-1}:
 For i = 0, n-1, do:
  For j = 0, n-1, do:
    c^k_{i,j} = c^k_{i,j} + a_{i,k} × b_{k,j}
  endfor
 endfor
enddoall
```

```
For k = 0, n-1, do: // accumulate all interim results
 For i = 0, n-1, do:
  For j = 0, n-1, do:
    c_{i,j} = c_{i,j}+c^k_{i,j}
  endfor
 endfor
endfor
```

(k, j, i): The outermost loop is k. Each processor has to keep a $O(n^2)$ memory space to store the interim result of matrix C. We call the interim result of matrix lives in k-th processor C^k.

P_k: contains k-th column $[a_{0k}\ a_{1k}\ ..\ a_{(n-1)k}]$, k-th row $[b_{k0}\ b_{k1}\ ..\ b_{k(n-1)}]$, produces matrix C^k $[c^k_{00}\ c^k_{01}..\ c^k_{0(n-1)}..c^k_{10}\ c^k_{11}\ ..\ c^k_{1(n-1)}..c^k_{(n-1)0}c^k_{(n-1)1}..c^k_{(n-1)(n-1)}]$

```
Do All P_{k=0,n-1}:
 For j = 0, n-1, do:
  For i = 0, n-1, do:
    c^k_{i,j} = c^k_{i,j} + a_{i,k} × b_{k,j}
  endfor
 endfor
enddoall
```

```
For k = 0, n-1, do: // accumulate all interim results
 For i = 0, n-1, do:
  For j = 0, n-1, do:
    c_{i,j} = c_{i,j}+c^k_{i,j}
  endfor
 endfor
endfor
```

The complexity of each case (assume only one data item can be transmitted in a single network cycle):

Unfolding by i or j:

Time = $O(n^2)$ multiplications, Memory = $O(n)$, Communications = $O(n^2)$ network cycles

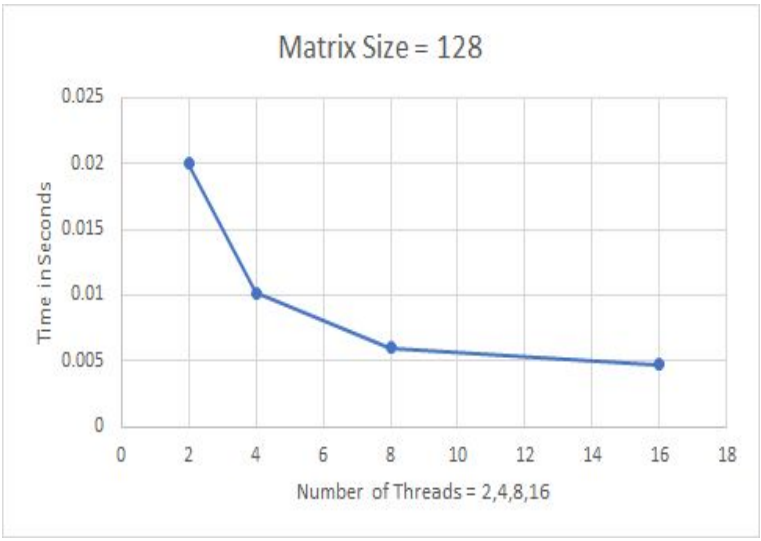
Unfolding by k: (need to accumulate the final result)

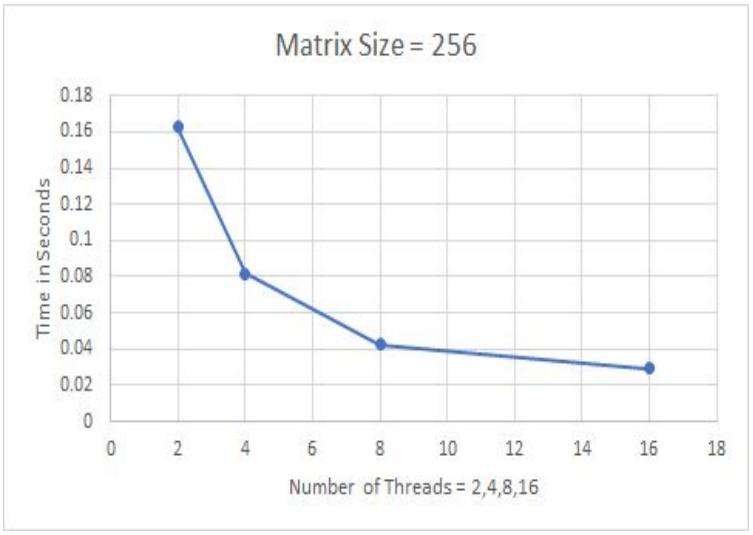
Time = $O(n^2)$ multiplications, Memory = $O(n^2)$, Communications = $O(n^3)$ network cycles

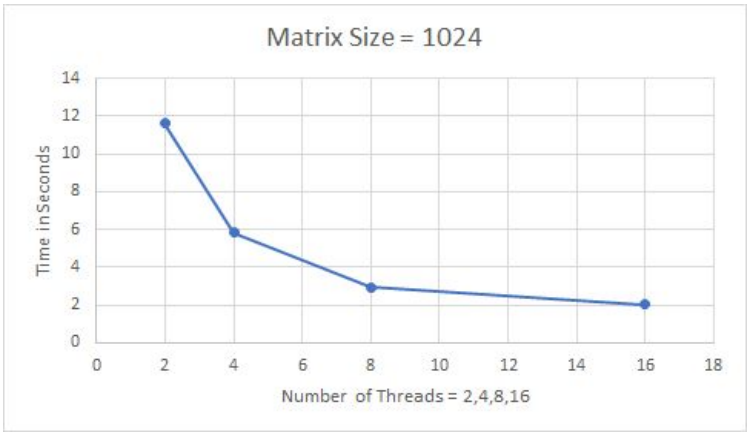
3. HW Specifications : UCI openlab server

CPU Model	Intel Xeon @ 3.33 Ghz
# of cores	24
# threads per cores	2

In the mat-mult.c, each thread performs getRow() operation which gets rows of resulting product matrix One by one. The matrices are stored in row-major form.







Comparison of matrix multiplication for different number of threads and matrix size.

Analysis :

Matrix Size	Comments
64	The highlighted graph in the matrix shows that the number of computations are very less and the effect of using multiple threads is not prominent. Until 4 threads we do see a reduced time, but after that the overhead of spawning more threads comes into effect and causes run time to be more.
128	As per the graph as the number of threads increases the time reduces exponentially. But between 8 and 16 threads the improvement is almost similar, as the number of spawned threads again starts to be more prominent.
256	As per the graph as the number of threads increases the time reduces exponentially. Also, if we notice, compared to M = 128, the run times are almost 8x. This could be due to the fact that matrix sizes are getting bigger.
512	Behavior is similar to M = 256
1024	Behavior is similar to M = 256.

One thing we haven't considered so far is that, as the matrix sizes are getting bigger, the time taken for a fixed number of threads compared to smaller M's is growing exponentially. For example, it takes 16 seconds with 2 threads for M = 1024, while only 1.4 seconds for M = 512. This has to do with the size of matrices and how the cpu to memory access patterns are changing with this increased size. Because a matrix with M = 1024 has a size of about 4000Kb (considering 4 byte integers) and we have 2 such matrices for multiplication. For M = 1024, the program would definitely try to access L3 cache to Main Memory more number of times which increases the access time for each matrix element. For smaller matrices, they can be stored in lower levels of caches and hence reduce the access time for them. This will become a major overhead compared to thread spawning whose impact is significantly overshadowed for M >= 256. We might introduce various formats introduced in the class to take more advantage of temporal locality using cache.