Declaration on Plagiarism

This form must be filled in and completed by the student(s) submitting an assignment

Name:	Satyam Ramawat
Student Number:	19210520
Programme:	Masters in Computing (Data Analytics)
Module Code:	CA670
Assignment Title:	Concurrent Programming – Assignment 2 Efficient Large Matrix Multiplication in OpenMP
Submission Date:	15th April 2020
Module Coordinator:	Dr David Sinclair

I declare that this material, which I now submit for assessment, is entirely my own 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 understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I have read and understood the Assignment Regulations. I have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

I have read and understood the referencing guidelines found at http://www.dcu.ie/info/regulations/plagiarism.shtml, https://www4.dcu.ie/students/az/plagiarism and/or recommended in the assignment guidelines

Name: Satyam Ramawat Date: 15th April 2020

Efficient Large Matrix Multiplication in OPENMP

Problem Given: Develop an **efficient** large matrix multiplication algorithm in OpenMP. A prime criterion in the assessment of your assignment will be the efficiency of your implementation and the evidence you present to substantiate your claim that your implementations are efficient. Overall, develop an algorithm which can do Matrix Multiplication, where Large Matrix should be used to test performance via OPEN MP

Solution: The Matrix Multiplication is a very challenging task for humans and as well as for computers machine also. As per the requirement of this assignment, OPENMP has been used in order to write an efficient large matrix multiplication algorithm, where according to the study [1], the most efficient algorithm is ikj-algorithm. In this work, Ikj-algorithm has been compared with the ijk-algorithm along with the traditional way of matrix multiplication, where execution time has been compared between these three.

Malloc() is a function which enable to allocate memory dynamically at run time, whereas it is efficient to occupy NxN matrix memory in order to test NxN Multiplication Matrix.

```
for(int i=0; i<dimension; i++){
    matrix[i] = malloc(dimension * sizeof(TYPE));
}
```

Fig. 1 Malloc() in order to allocate memory dynamically

In this solution, the maximum capacity of large matrix is 46000x46000, thus program will allocate memory for 46000 x 46000 2D array, whereas according to figure below, 47000x47000 overflows the macro.

```
(base) Satyams-MacBook-Pro:19210520_CA670_Assignment_2 satyamramawat$ clang -Xpreprocessor -fopenmp -lomp CA670_19210520_AS2.c -o MatrixMultiply
CA670_19210520_AS2.c:21:13: warning: overflow in expression; result is -2085967296 with type 'int' [-Winteger-overflow]
TYPE OneD_A[MAX_DIM];
            ^
CA670_19210520_AS2.c:16:22: note: expanded from macro 'MAX_DIM'
#define MAX_DIM 47000*47000
                     ^
CA670_19210520_AS2.c:21:13: error: 'OneD_A' declared as an array with a negative size
TYPE OneD_A[MAX_DIM];
            ^~~~~~~
CA670_19210520_AS2.c:16:17: note: expanded from macro 'MAX_DIM'
#define MAX_DIM 47000*47000
                ^~~~~~~~~~~
CA670_19210520_AS2.c:22:13: warning: overflow in expression; result is -2085967296 with type 'int' [-Winteger-overflow]
TYPE OneD_B[MAX_DIM];
            ^
CA670_19210520_AS2.c:16:22: note: expanded from macro 'MAX_DIM'
#define MAX_DIM 47000*47000
                     ^
CA670_19210520_AS2.c:22:13: error: 'OneD_B' declared as an array with a negative size
TYPE OneD_B[MAX_DIM];
            ^~~~~~~
CA670_19210520_AS2.c:16:17: note: expanded from macro 'MAX_DIM'
#define MAX_DIM 47000*47000
                ^~~~~~~~~~~
```

Fig. 2 Failure defining 47000 Dimension matrix

OpenMP has been used in order to develop efficient code, it allows shared-memory parallelism by defining inbuilt parallel region directives, where it has directives are for both non-iterative and iterative program. In this work, below three directives have been used:

1. **#pragma omp parallel for**

Loop Parallelism construct is the very common parallelism technique in OPENMP, which enables to share resources during the iteration of the loop. It allows threads of a team to share and distribute the work among themselves which makes the code run parallel and resultant in the efficient outcome.

- **Non-Parallelism**
  for(int i=0;i<10;i++){
  //Do some work}
- **Parallelism with OpenMp**
  #pragma omp parallel for
  for(int i=0;i<10;i++){
  //Do some work}

2. #pragma omp parallel shared

This directive allows sharing data variable explicitly, where it provides the feature to binding the public variable or which is declared into different scope with the multi parallelism.

#pragma omp parallel for shared(n, a) private(b) has been used in this program, where b is a private variable, so when variable declared private OpenMP creates a duplicate copy of that variable in order to use it for parallelism and rather updating the same variable and avoid unintuitive behavior of a variable.

3. #pragma omp for schedule

Since OpenMP specialty is parallelization of loops, where it also allows us to define scheduling type. The schedule (static) type will divide the number of iteration of loops equally with other threads of the team. For instance, 64 iterations with 4 threads, so schedule(static) will distribute the iteration and work along with other threads equally, where Thread 1 executes 1,2,3,4,5….16 and Thread 2 executes 17,18,19,20,21…32 and likewise other 3rd & 4th Thread will follow up the same sequence. Static scheduling is efficient when there is an equal number of iterations have similar computational cost.

EXECUTION PROCEDURE

Three algorithms have been used, compared and tested in this work. Efficiency has been measured by recording execution time. Input has been provided through **command line argument**, where ***argv[1]*** is Number of iteration or result to be executed by each algorithm, ***argv[2]*** is Minimum NxN Matrix, ***argv[3]*** is Maximum NxN Dimension Matrix to be tested and ***argv[4]*** is the interval number at which NxN matrix should be generated.

*Commands to Execute the Program

```
(base) Satyams-MacBook-Pro:19210520_CA670_Assignment_2 satyamramawat$ clang -Xpreprocessor -fopenmp -lomp CA670_19210520_AS2.c -o MatrixMultiply
(base) Satyams-MacBook-Pro:19210520_CA670_Assignment_2 satyamramawat$ ./MatrixMultiply 1 200 2000 200
```

Fig. 3 Execution of program

- To run OPENMP program in MAC OS system, **Clang** has been used with OpenMP parameters, -fopenmp and –lomp [9].

- Example with 5 iteration(**argv[1]**), matrices tested with minimum 200x200(**argv[2]**) upto maximum 2000x2000(**argv[3]**) with interval of 200(**argv[4]**).

```
(base) Satyams-MacBook-Pro:19210520_CA670_Assignment_2 satyamramawat$ ./MatrixMultiply 5 200 2000 200
---------------------------------
Test : ikj-Algorithm Matrix Multiplication
---------------------------------
...............................
Dimension : 200
1.      0.010728
2.      0.012939
3.      0.012014
4.      0.010732
5.      0.011030
```

Fig. 4 Example of Execution of program with parameters

ALGORITHMS DESIGN

1) Traditional Approach Matrix Multiplication

```
for(int i=0; i<dimension; i++){
        for(int j=0; j<dimension; j++){
            for(int k=0; k<dimension; k++){
                matrixC[i][j] += matrixA[i][k] * matrixB[k][j];
```

Fig. 5 Traditional Way of Matrix Multiplication

This the traditional way how we are doing matrix multiplication, by least bother about the code efficiency and more focused on the result. Results for this algorithm have been saved in TraditionalMultiplyTest.txt file and screenshots are available in the screenshot folder.

2) Ijk-Algorithm for Matrix Multiplication

```
#pragma omp parallel for
    for(int i=0; i<dimension; i++){
        for(int j=0; j<dimension; j++){
            for(int k=0; k<dimension; k++){
                matrixC[i][j] += matrixA[i][k] * matrixB[k][j];
            }
        }
    }
```

Fig. 6 Ijk-Algorithm of Matrix Multiplication

IJK-Algorithm is nevertheless but optimized of the traditional approach where OPENMP directive "*#pragma omp parallel for"* construct has been used to divide the work of loop among the team of thread and enable code to run parallel, which results in faster than the traditional approach of matrix multiplication. Results for this algorithm have been saved in ijk_algorithmTest.txt file and screenshots are available in the screenshot folder.

3) Ikj-Algorithm Efficient Large Matrix Multiplication

```
#pragma omp parallel shared(matrixC) private(i, j, k, iOff, jOff, tot)
num_threads(40)
    {
        #pragma omp for schedule(static)
        for(i=0; i<dimension; i++){
            iOff = i * dimension;
            for(j=0; j<dimension; j++){
                jOff = j * dimension;
                tot = 0;
                for(k=0; k<dimension; k++){
                    tot += flatA[iOff + k] * flatB[jOff + k];
                }
                matrixC[i][j] = tot;
            }
        }
}
```

Fig. 7 Efficient Large Matrix Multiplication Ikj-Algorithm

Ikj algorithm is a more cache efficient algorithm, where it converts 2D matrices into 1D to reduce cache misses so convert (matrixA, matrixB, dimension) function has been created. It is a more optimized and efficient algorithm than the ijk-algorithm, where the ijk-algorithm keeps updating the same variable matrixC[i][j], so the efficiency can be improved by just introducing local variable, as per mentioned in the above figure. Furthermore, to increase more efficiency, OPENMP Directives has been used, where the operation has been performed under #pragma omp parallel shared(matrixC) private(i, j, k, iOff, jOff, tot) num_threads(40){} Block and #pragma omp for schedule(static) used before for loop which performs matrix multiplication. Results for this algorithm have been saved in ikj_algorithmTest.txt file and screenshots are available in the screenshot folder.

Three Additional **functions** has been created:

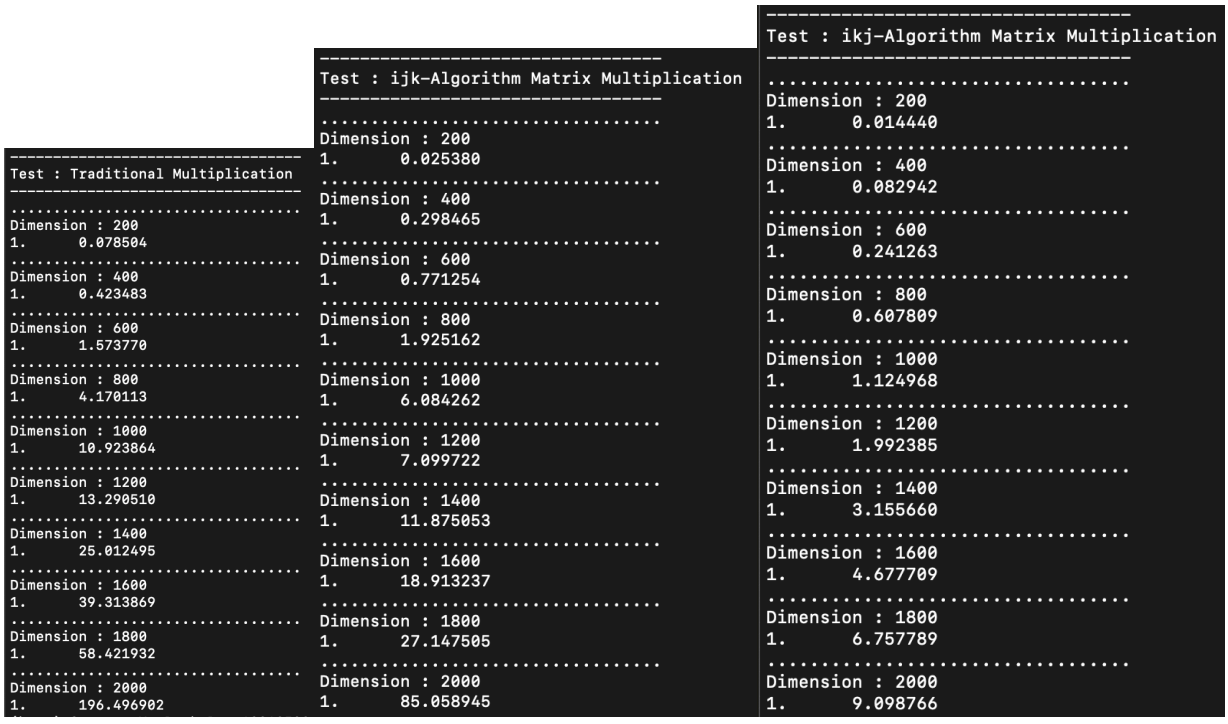
- ***void convert(TYPE** matrixA, TYPE** matrixB, int dimension) -*** Converts 2D array into 1D for cache misses.
- ***TYPE** zeroSquareMatrix(int dimension) -*** Additional Matrix to store the answer.
- ***TYPE** randomSquareMatrix(int dimension) –*** Generate random values into Matrix.

Three Test-Coverage Function has been created, in order to generate the dimension of Matrix from argv[2], upto argv[3], at interval argv[4].

- ***void TraditionalMultiplyTest(int dimension, int iterations);***
- ***void ijk_algorithmTest(int dimension, int iterations);***
- ***void ikj_algorithmTest(int dimension, int iterations);***

RESULT AND EVALUATION

This solution has been run on Configuration, **System:** MacBook Pro, **Processor:** 3.1 GHz Dual-Core Intel Core i5, **Memory:** 8 GB 2133 MHz LPDDR3, **Graphics:** Intel Iris Plus Graphics 650 1536 MB.



```
--------------------------------
Test : Traditional Multiplication
--------------------------------
.................................
Dimension : 200
1.      0.078504
.................................
Dimension : 400
1.      0.423483
.................................
Dimension : 600
1.      1.573770
.................................
Dimension : 800
1.      4.170113
.................................
Dimension : 1000
1.      10.923864
.................................
Dimension : 1200
1.      13.290510
.................................
Dimension : 1400
1.      25.012495
.................................
Dimension : 1600
1.      39.313869
.................................
Dimension : 1800
1.      58.421932
.................................
Dimension : 2000
1.      196.496902
```

```
--------------------------------
Test : ijk-Algorithm Matrix Multiplication
--------------------------------
.................................
Dimension : 200
1.      0.025380
.................................
Dimension : 400
1.      0.298465
.................................
Dimension : 600
1.      0.771254
.................................
Dimension : 800
1.      1.925162
.................................
Dimension : 1000
1.      6.084262
.................................
Dimension : 1200
1.      7.099722
.................................
Dimension : 1400
1.      11.875053
.................................
Dimension : 1600
1.      18.913237
.................................
Dimension : 1800
1.      27.147505
.................................
Dimension : 2000
1.      85.058945
```

```
--------------------------------
Test : ikj-Algorithm Matrix Multiplication
--------------------------------
.................................
Dimension : 200
1.      0.014440
.................................
Dimension : 400
1.      0.082942
.................................
Dimension : 600
1.      0.241263
.................................
Dimension : 800
1.      0.607809
.................................
Dimension : 1000
1.      1.124968
.................................
Dimension : 1200
1.      1.992385
.................................
Dimension : 1400
1.      3.155660
.................................
Dimension : 1600
1.      4.677709
.................................
Dimension : 1800
1.      6.757789
.................................
Dimension : 2000
1.      9.098766
```

Above three images are the results generated by Matrix Multiplication Algorithm for iteration 1, from 200 Dimension till 2000 Dimension with the interval of 200.

Test-case Id	Size of Array(Matrix)	Parallel Execution Time		
		Traditional	Ijk-Algo	Ikj-Algo
1	200 x 200	0.078504	0.025380	0.014440
2	400 x 400	0.423483	0.298465	0.082942
3	600 x 600	1.573770	0.771254	0.241263
4	800 x 800	4.170113	1.925162	0.607809
5	1000 x 1000	10.923864	6.084262	1.124968
6	1200 x 1200	13.290510	7.099722	1.992385
7	1400 x 1400	25.012495	11.875053	3.155660
8	1600 x 1600	39.313869	18.913237	4.677709
9	1800 x 1800	58.421932	27.147505	6.757789
10	2000 x 2000	196.496902	85.058945	9.098766

Table 1. Results Comparison of Algorithms

According to Table 1, we can see that ikj-Algorithm has performed far efficiently well from both ijk-Algorithm which only uses one OPENMP loop construct and a Traditional Approach of calculating matrix multiplication. Following the Figure 8 will provide better insight knowledge on the above results, where it shows how the ikj-algorithm has worked efficiently with parallel time execution in seconds along with the increment of the size of the array.

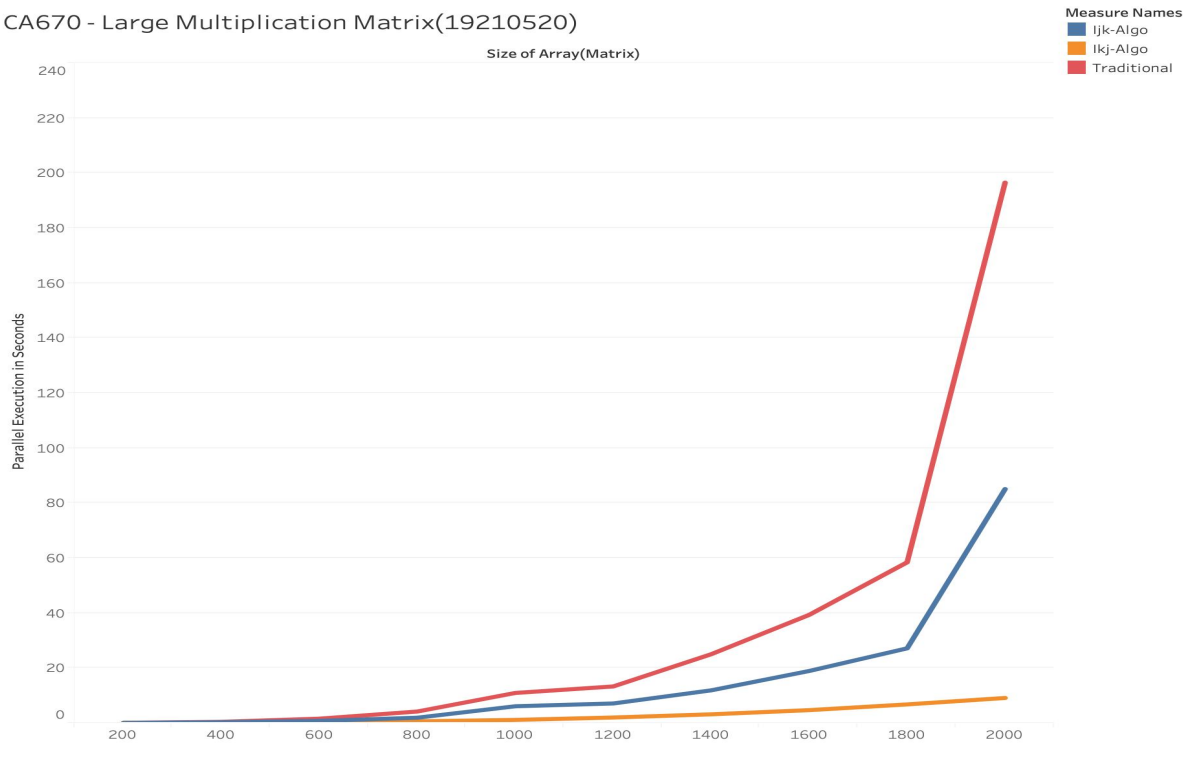


Fig. 8 Performance Comparison Line Graph between Three Algorithms

CONCLUSION / JUSTIFICATION

1. According to test cases and obtained results OPENMP is the efficient parallelism framework that distributes work equally among all other threads so that code execution becomes faster.
2. Ijk-Algorithm is just an addition of one line of code of OPEMMP in the traditional approach, and the result we can see in Figure 8, how efficiently execution time is decreased as compared to traditional.
3. In Figure 8, we can see with just a small change in algorithm, ikj-algorithm performed outstanding from both the algorithms, where OPEMMP parallel shared block used to execute code and for schedule(static) used before multiplication logic.
4. Overall, I'll conclude that OPENMP allows parallelism better in loop parallelism where multiplication matrix logic is developed by using loops and OPENMP allows gradual and distributed parallelization on loop.
5. OPENMP, reduce the execution time by using all available resources from the team of thread as per Figure 8.

REFERENCES

[1]. Mathematics Science College, Computational Science, http://www.math.utep.edu/Faculty/xzeng/2019fall_cps5401/2019fall_cps5401/CPS_5401_Introduction_to_Computational_Sciences_files/homework_3.pdf

[2]. Martin Thoma, Matrix multiplication on multiple cores in Python, Java and C++, 21 october 2013, https://martin-thoma.com/part-iii-matrix-multiplication-on-multiple-cores-in-python-java-and-c/

[3]. Matrix Multiplication, Wikipedia, https://en.wikipedia.org/wiki/Matrix_multiplication#Algorithms_for_efficient_matrix_multiplication

[4]. Singh, Niraj & Chakraborty, Soubhik & Mallick, Dheeresh. (2013). An Adaptable Fast Matrix Multiplication Algorithm, Going Beyond the Myth of Decimal War. https://arxiv.org/pdf/1308.2400.pdf

[5]. OPENMP, OBJECT COMPUTING, Charles Calkins, MAY 2010 https://objectcomputing.com/resources/publications/mnb/openmp

[6]. OPENMP, DCU School of Computing, David Sinclair,https://www.computing.dcu.ie/%7Edavids/courses/CA670/CA670_OpenMP_2p.pdf

[7]. OPENMP topic: Loop Parallelism, Introduction to high performance scientific computing, Victor Eijkhout with Robert van de Geijn and Edmond Chow, lulu.com, 2011, https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html

[8]. OpenMP: For and Scheduling, jakascorner.com, June 13 2016, http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

[9]. OpenMP on High Sierra, Henry Schreiner, August 20, 2018, https://iscinumpy.gitlab.io/post/omp-on-high-sierra/

[10]. https://www.guru99.com/c-dynamic-memory-allocation.html