# 2021/2022 CA670 Concurrent Programming

# Assignment Number 2

**DISCLAIMER**

A picture containing text, newspaper, screenshot, document

Description automatically generated

Submitted By: Praveen Kumar Chauhan

Student ID: 21261912

**Develop 2 programs, capable of executing on multiple cores, that can multiply 2 large dense metrices. The 1st program is a multi-threaded Java program. The 2nd program is an OpenMP program. Both programs should be efficient and not adopt a naive approach.**

**PROPOSED SOLUTION:**In this project, I offered two solutions to the problem of huge matrix multiplication on multiple cores i.e., 2,4,6 and 8 cores respectively. The benchmark of my laptop is as follows.  
**Processor:** Intel(R) Core (TM) i5-8250U CPU @ 1.60GHz (8 CPUs), ~1.8GHz  
**Memory:** 8192MB RAM  
**Java JDK:** 1.8.0\_102 x64 on Windows 10

**NOTE:** Before continuing, you need have a basic understanding of how to adjust the fundamental settings in Windows 10.

### **Core Settings in Windows 10** If you’re using Windows 10, all your processor cores will be fully utilized by default if your BIOS/UEFI is set correctly. The only time you would use this technique is to limit cores, whether for software compatibility reasons or otherwise.

1. Type ‘msconfig’ into the Windows Search Box and hit Enter.
2. Select the Boot tab and then Advanced options.
3. Check the box next to Number of processors and select the number of cores you want to use (probably 1, if you are having compatibility issues) from the menu.
4. Select OK and then Apply.
5. Restart your PC.

Graphical user interface, application

Description automatically generated

* To tackle this problem, the first solution uses the Strassen algorithm in OpenMP, whereas the second solution/approach for Matrix Multiplication leverages the Multiple Threading notion using Strassen algorithm in Java. After the successful execution of both the solutions I have compared the execution time on multiple threads i.e., 1-8 threads working on 500x500 dimensioned matrices for both OpenMp and Java later in the report.
* Following the successful execution of both solutions, I compared the execution time for both OpenMp and Java on multiple threads, i.e., 1-8 threads working on 500x500, 700x700 and 900x900 dimensioned matrices respectively using table and graphs, further in the report.

1. **SOLUTION 1 DESCRIPTION: USING OpenMP THREADS: Strassen Algorithm**

The Strassen Method works with square matrices with dimensions of 2^n\*2^n. The scalability of the system is unrestricted. The approach will not work if the dimensions are not 2^n\*2^n. The matrices A,B, and C are subdivided into four submatrices, avoiding matrix multiplication between large matrices. Because it's best to employ addition and subtraction between matrices, the number of multiplications is limited (and reserved for matrices of dimensions (2^n)/4). The following is a description of the method.

Text

Description automatically generated with medium confidence

Simple operations among matrices of small dimension are computed after the matrices have been divided into submatrices.

Text, letter

Description automatically generated

Now M matrices are used to compute the result Matrix C:

Text

Description automatically generated

The matrix C is now computed, and the submatrices are assigned to it. Because the code runs faster by lowering the number of multiplications (which is the more expensive operation), this method performs well.

As previously stated, I have performed OpenMP matrix multiplication on many cores (2, 4, 6, and 8) correspondingly. With the setups listed below.

**Processor:** Intel(R) Core (TM) i5-8250U CPU @ 1.60GHz (8 CPUs), ~1.8GHz  
**Memory:** 8192MB RAM

* **Matrix Dimension**: 500x500
* **Thread Count**: 1,2,3,4,5,6,7 and 8
* **Threshold Limit**: 120

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Time in secs | 7.348  **(Ref: Pic 1)** | 3.368  **(Ref: Pic 2)** | 1.975  **(Ref: Pic 3)** | 1.687  **(Ref: Pic 4)** | 1.432  **(Ref: Pic 5)** | 1.226  **(Ref: Pic 6)** | 1.143  **(Ref: Pic 7)** | 0.925  **(Ref: Pic 8)** |

**The total time taken for the execution of 500x500 matrices for 1 to 8 threads executing on matrix multiplication using Strassen Algorithm for OpenMP is:  
 19.104 seconds.**

OpenMp Snapshot: Pic 1

Text

Description automatically generated

OpenMp Snapshot: Pic 2

Text

Description automatically generated

OpenMp Snapshot: Pic 3

Text

Description automatically generated

OpenMp Snapshot: Pic 4

Text

Description automatically generated

OpenMp Snapshot: Pic:5

Text

Description automatically generated

OpenMp Snapshot: Pic:6

Text

Description automatically generated

OpenMp Snapshot: Pic:7

Text

Description automatically generated

OpenMp Snapshot: Pic:8

Text

Description automatically generated

In the similar manner for

* **Matrix Dimension**: 700x700
* **Thread Count**: 1,2,3,4,5,6,7 and 8
* **Threshold Limit**: 120

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Time in secs | 17.225 | 9.714 | 7.884 | 7.225 | 6.476 | 4.925 | 4.722 | 3.676 |

**The total time taken for the execution of 500x500 matrices for 1 to 8 threads executing on matrix multiplication using Strassen Algorithm for OpenMP is:  
 61.847 seconds.**

In the similar manner for

* **Matrix Dimension**: 900x900
* **Thread Count**: 1,2,3,4,5,6,7 and 8
* **Threshold Limit**: 120

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Time in secs | 45.356 | 25.050 | 20.221 | 15.126 | 13.089 | 10.254 | 9.705 | 8.721 |

**The total time taken for the execution of 500x500 matrices for 1 to 8 threads executing on matrix multiplication using Strassen Algorithm for OpenMP is:  
 132.396 seconds.**

In the similar manner for

* **Matrix Dimension**: 1000x1000
* **Thread Count**: 1,2,3,4,5,6,7 and 8
* **Threshold Limit**: 120

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Time in secs | 83.071 | 52.529 | 50.205 | 40.481 | 37.936 | 30.243 | 27.011 | 25.327 |

**The total time taken for the execution of 500x500 matrices for 1 to 8 threads executing on matrix multiplication using Strassen Algorithm for OpenMP is:  
346.803 seconds.**

**Conclusion of OpenMP Execution (Strassen Algorithm):**

**Dimensions Vs Time table of OpenMP.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dimensions | 500x500 | 700x700 | 900x900 | 1000x1000 |
| Time in Secs | 19.104 | 61.847 | 132.396 | 346.803 |

According to the overall observation of the matrix multiplication (see table above), there were no concerns with the code running on multiple cores, and the execution time was also very acceptable while utilizing the Strassen algorithm for OpenMP.

**Important Note: Depending on the OpenMP threshold lower limit, the code is performed in two phases. For Strassen's algorithm multiplication, the matrix size must be more than or equal to 128. Otherwise, OpenMP's universal matrix multiplication will be used.**

=====================================================

**SOLUTION 2 DESCRIPTION: Java Multithreaded Matrix Multiplication Execution using: Strassen Algorithm**

**Explanation of the code:**

Implementing Matrix Multiplication using Strassen's Algorithm was a difficult challenge, but it was made easier by using Java Threads. Strassen's approach aids in the efficient multiplication of large matrices, which is substantially faster than conventional matrix multiplication.

* 1. **StrassenThread.java –**This class is the starting point for the whole Java implementation of Strassen's algorithm. It is an Initiator class because it has a main method that works as a template for what needs to be shown to the user in order to get the required input, such as the matrix size. Aside from the main procedure, the two matrices are randomly initialized. After that, the StrassenMatrixMultiplicationAlgorithm.java class is called by constructing an object that contains our algorithm logic. For the sake of our analysis, we've also computed the time it took to compute the result.
  2. **StrassenSplitAddSubJoin.java –**This class is our assistance method, and it has four methods that are commonly utilized throughout our approach. Aside from that, we have the add(), sub(), join(), and split() functions, which are the foundations of our implementation
  3. **StrassenMatrixMultiplication.java –**Our algorithm's main logic is in this class. It contains a recursive method called Strassen() that we may use to create our algorithm using a divide-and-conquer strategy. This entails breaking our matrices into four sections and iteratively working on each one until we reach a conclusion. We have a 128-size threshold below which we utilize the standard Strassen's approach without threads and over which we use our threads. We start eight threads to work on the four sub-parts from each of the two matrices. We have eight threads working on eight sub-parts in total. The join on thread assures that we have a thread dedicated to a single activity. Following that, four more threads calculate the C11, C12, C21, and C22 values, respectively. This is the result. The Mul() class contains all of the code for the threads that will do the various splitting operations according to Strassen's method. It provides a constructor that will set the variables and matrices to their default values. Mul2() contains all of the code for the threads that will do the various join operations according to Strassen's method. It also has a constructor that sets up the variables and matrices.

**Output:**

Matrix Multiplication for Dimension 500x500

Graphical user interface, text, application, email

Description automatically generated

Matrix Multiplication for Dimension 700x700

Text

Description automatically generated

Matrix Multiplication for Dimension 900x900

Text

Description automatically generated

Matrix Multiplication for Dimension 1000x1000

Text

Description automatically generated

**Dimensions Vs Time**

Dimensions Vs Time in seconds table for overall execution of all the 8 threads for multiple dimensions i.e., 500x500, 700x700, 900x900 and 1000x1000 respectively.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dimensions | 500x500 | 700x700 | 900x900 | 1000x1000 |
| Time in Seconds | 5.445 | 40.146 | 63.616 | 64.081 |

**Dimensions Vs Time table of Java.**

=====================================================

**Conclusion:**

**I arrived at the conclusion that Java Matrix multiplication is substantially faster than OpenMp since I utilized the identical algorithm for both implementations of Dense matrix multiplication. I populated the graph by comparing the Dimensions Vs Time tables of OpenMp and Java, respectively. Please find it below.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dimensions | 500x500 | 700x700 | 900x900 | 1000x1000 |
| Time for OpenMp | 19.104 | 61.847 | 132.396 | 346.803 |
| Time for Java | 5.445 | 40.146 | 63.616 | 64.081 |

**REFERENCES:**

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

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

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