Dynamic Intel, wants code!

Assignment Details

- Author: `Samuel Jesuthas`

- Module: `MCOMD3HPC – High Performance Computing`
- Deadline: `10th January 2024`

Introduction

- This report will feature detailed analysis of the importance of threading in high performance applications. When a program runs multi-threaded, you are spreading out the workload to multiple workhorses, which increases the overall performance of your program.
- The program we will look at, simply generates 2, 1000x1000 matrixes with random values, and multiplies them together. It then takes the result of this and multiplies it by another 1000x1000 randomly generated matrix, and then it will do this for a 3rd time to get 3 iterations.
- Throughout this report, we will analyze the importance of threading in this application as it significantly increases the performance when the multiplications are performed. We will also look at thread pool implementations for this program and we will test how stable and scalable my solution is for business use.

Testing Information

- Note that all the testing and program execution examples in this report, have been performed on a desktop machine with the following specs:
  - CPU: **AMD Ryzen 5 5600G**
  - RAM: **32 GB DDR4**
  - GPU: **AMD Radeon RX 6700 (10 GB GDDR6)**
- Keep in mind that the actual performance may vary based on the system and the workload.

Task 1 - Simple Non-threaded implementation

matrixEngine.java (Matrix Calculation Handler)

- To perform a 1000x1000 matrix multiplication, firstly, we need to generate the 2 matrixes to get our first iteration. To simplify the process, I made a class called `matrixEngine` that will handle everything for us.

- The first method `GenerateBaseMatrixes()` will generate our 2 1000x1000 matrixes which we will multiply together.

```
public matrixResult GenerateBaseMatrixes()
{
    long[][] matrix1 = new long[1000][1000];
    long[][] matrix2 = new long[1000][1000];

    fillMatrix(matrix1);
    fillMatrix(matrix2);

    return new matrixResult(matrix1, matrix2);
}
```

  - The reason I am using `long` as the data type for the matrix, is so that I can handle overflow of values [1]. `long` can support values from `-9,223,372,036,854,775,808` up to `9,223,372,036,854,775,807`
  - Because Java doesn't allow methods to return more than 1 data type, I made another class called `matrixResult` which stores the 2 matrixes in one object, also known as a tuple.

- The `fillMatrix()` method does a `for` loop through each value of the matrix and fills it with a value that is randomly generated from 1-100.

```
public void fillMatrix(long[][] matrix)
{
    Random random = new Random();

    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix[i].length; j++) {
            matrix[i][j] = random.nextInt(100);
        }
    }
}
```

- The final method in this class is the `multiplyMatrices()` method, which is what we will be using to multiply 2 matrixes together.

```
public long[][] multiplyMatrices(long[][] matrix1, long[][] matrix2)
{
    long[][] resultMatrix = new long[1000][1000];
    for (int i = 0; i < matrix1.length; i++) {
```

```java
            for (int j = 0; j < matrix2[0].length; j++) {
                for (int k = 0; k < matrix2.length; k++) {
                    resultMatrix[i][j] += matrix1[i][k] * matrix2[k][j];
                }
            }
        }

        return resultMatrix;
    }
```

- The triple-nested `for` loop in this method is what performs the multiplication. What it does is iterate through each element of the 2 matrixes and perform dot-product [2] matrix multiplication
    - `i` will iterate over every ROW of `matrix1`
    - `j` will iterate over every COLUMN of `matrix2`
    - `k` will iterate over the COLUMNS of `matrix1` and the ROWS of `matrix2`
- As we are iterating over every element of the matrix, we get the product of the current row of `matrix1` and then multiply it by the product of the current column of `matrix2`. We assign the results of this to the corresponding element of the `resultMatrix` matrix

```java
resultMatrix[i][j] += matrix1[i][k] * matrix2[k][j];
```

App.java (Execution)

- The main application will run based on what option you select. You have the option to run the matrix program in a threaded or non-threaded manner

```
Please choose an option to run this program
1. Run /w No Threading
2. Run /w Threading
3. Run using Thread Pool
4. Verify Threaded vs Non-Threaded
5. Exit Program
```

- The non-threaded method takes in 2 parameters `matrix1` & `matrix2` which are the starting 2 matrixes we need to perform the 1st multiplication. We are also returning a `long[][]` which is the result matrix, in order to compare the result against the threaded versions

```java
private static long[][] performMatrixMultiplicationNoThreading(long[][] matrix1, long[]
{

}
```

- Next, we are displaying the 2 starting matrixes by using the `printMatrixPreview()` method. This method allows us to view a variable length portion of the matrix, in this case the first 10x10.

```
//Initialize MatrixEngine
var MatrixEngine = new matrixEngine();

//Display the 2 starting matrixes which will be multiplied together to give our first i
System.out.println("Matrix 1: \n");
printMatrixPreview(matrix1, 10, 10); //We will only display the first 10x10 portion of
System.out.println("Matrix 2: \n");
printMatrixPreview(matrix2, 10, 10);
```

- Now we will perform our first multiplication by using the `multiplyMatrices()` method.

```
//Executing the multiplication method to get our first iteration
var result1 = MatrixEngine.multiplyMatrices(matrixes.matrix1, matrixes.matrix2);
System.out.println("1st Multiplication /w No Threading: \n");
printMatrixPreview(result1, 10, 10);
```

    - Since the method returns a `long[][]` it would be best to store this in a variable, which is what `result1` is.

- Now we have our first iteration, all we have to do is multiply 2 more times. We take the output of `result1` and multiply it by a brand new 1000x1000 matrix

```
//Now a 2nd randomly generated 1000x1000 matrix will be created
long[][] secondIterationMatrix = new long[1000][1000];
MatrixEngine.fillMatrix(secondIterationMatrix);

//We will multiply this matrix by the result of the 1st multiplication
var result2 = MatrixEngine.multiplyMatrices(result1, secondIterationMatrix);
System.out.println("2nd Multiplication /w No Threading: \n");
printMatrixPreview(result2, 10, 10);

//And we will do this a 3rd time, for our 3rd iteration
long[][] thirdIterationMatrix = new long[1000][1000];
MatrixEngine.fillMatrix(thirdIterationMatrix);

//Multiplying the 3rd matrix by the result of the 2nd multiplication
var result3 = MatrixEngine.multiplyMatrices(result2, thirdIterationMatrix);
System.out.println("3rd Multiplication /w No Threading: \n");
printMatrixPreview(result3, 10, 10);
```

    - `result1`, `result2` & `result3` should show an exponential increase in the output

Output

- When we select the non-threaded option and run our program. This is what 1 run of the program will give us. Note that, each run will be different as we are randomly generating each matrix

```
Please choose an option to run this program
1. Run with No Threading
2. Run with Threading
3. Exit Program
1
Matrix 1:

77 72 77 37 20 70 89 76 97 26
11 20 96 9 34 69 18 19 11 67
46 4 35 75 37 72 18 7 57 46
0 4 57 28 75 32 71 19 68 0
59 21 47 31 72 97 93 94 62 28
86 35 53 69 53 23 65 56 85 71
24 21 55 82 64 1 27 52 34 70
57 1 45 81 4 72 61 52 12 91
67 6 92 99 62 92 56 36 17 36
78 65 80 84 96 47 86 50 21 26

Matrix 2:

97 18 77 20 2 89 0 87 55 96
56 45 6 13 50 81 23 62 48 18
86 47 50 96 10 65 84 84 54 5
2 44 61 3 78 23 99 35 10 13
72 76 88 97 64 82 22 74 52 83
10 69 34 63 55 64 2 88 24 77
55 36 96 35 81 80 97 17 48 99
36 99 22 12 6 30 75 23 43 22
79 23 73 95 48 9 55 78 42 95
18 87 8 91 22 32 72 28 94 48

1st Multiplication /w No Threading:
...

2nd Multiplication /w No Threading:
...

3rd Multiplication /w No Threading:
...

Execution time: 10129 milliseconds
```

The resulting matrixes will not be included to simplify the displayed example

- The non-threaded method takes so long, in this particular run, it took us 10,129 ms to execute the whole method.

Task 2 - Multi-threaded Implementation

- We will now look at a threaded implementation of this program. The multiplication process will now be done by multiple threads, which should significantly increase the performance of the application

matrixEngineThreaded.java (The thread code)

```java
public class matrixEngineThreaded extends Thread {
    private long[][] matrix1;
    private long[][] matrix2;
    private long[][] resultMatrix;
    private int startRow;
    private int endRow;
}
```

- To ensure we are utilizing threads, we can use the `extends` [3] [4] attribute to make this whole class run in a threaded manner
- In this class, we have some basic properties, and a constructor which allows us to pass this information in.
    - `matrix1` & `matrix2` are the matrixes passed in for multiplication
    - `resultMatrix` is used to store the result
    - `startRow` & `endRow` are used for the multiplication, as the workload has been distributed among threads. We are working with portions of a 1000x1000 matrix
- In this class, we have one main method which is the `multiply()` method, which is actually just the same code as the `multiplyMatrices()` method in the `matrixEngine` class. However, we run this method in the `run()` method, which is responsible for starting the thread

```java
@Override
public void run() {
    multiply();
}
```

matrixEngine.java (Running the thread)

- Now we will head back to the `matrixEngine` class to perform our matrix multiplication. Since most of our main logic is in that class, it would be wise to make our threaded implementation in there, to reduce unnecessary code.

- Our threaded multiplication takes place in the `multiplyMatricesThreaded()` method, which returns a `long[][]` which is the resulting matrix after the multiplication
- We start by creating an empty 1000x1000 matrix to use as the return object

```
long[][] resultMatrix = new long[1000][1000];
```

- Now we start creating the threads and we will calculate how many rows each thread will handle by dividing the total number of rows by the specified number of threads, which in this case is `numThreads`

```
// Calculating the distribution of workflow for these threads
for (int i = 0; i < numThreads; i++) {
    int startRow = i * rowsPerThread;
    int endRow = (i == numThreads - 1) ? 1000 : (i + 1) * rowsPerThread;

    threads[i] = new matrixEngineThreaded(matrix1, matrix2, resultMatrix, startRow, end
    threads[i].start();
}
```

  - What we are doing here is `for` looping through each thread in the `threads` array
  - We are then assigning a portion of the matrix for the thread to process, which is why we have the constructor for `matrixEngineThreaded` , because we can assign it to that class, and begin the thread
  - Starting the thread, will execute the `multiply()` method

- Now the threads will begin execution and start multiplying portions of each matrix until the entirety of `matrix1` and `matrix2` are multiplied. We have to make some logic to check if the threads have finished doing their job

```
// Wait for all threads to finish
try {
    for (int i = 0; i < numThreads; i++) {
        threads[i].join();
    }
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

  - A useful tool in java is the `try/catch` block, which ensures the application doesn't crash whenever it encounters an exception

    - We can use the `join()` method to ensure that the main thread waits for each worker thread to complete before moving on.

- Overall, this implementation divides the matrix multiplication task among multiple threads, allowing for concurrent execution and potentially speeding up the process compared to the single-threaded approach.

App.java (Execution)

- A lot of the logic to execute this, is very similar to the `performMatrixMultiplicationNoThreading()` method. However this time, we need to use the `multiplyMatricesThreaded()` method, to run the program in a threaded manner

- This is what happens when we execute the program, with threaded support

```
Please choose an option to run this program
1. Run /w No Threading
2. Run /w Threading
3. Run using Thread Pool
4. Exit Program
2
Matrix 1:

77 69 54 35 84 24 9 63 47 72
94 31 68 74 84 31 62 52 43 70
12 76 58 73 87 25 17 18 51 62
79 43 11 61 28 61 27 89 76 84
30 53 76 39 81 2 58 11 33 98
2 55 46 97 70 71 91 90 47 33
29 23 56 29 85 37 95 45 89 79
13 44 71 94 94 32 47 39 51 82
80 81 48 99 86 41 1 19 17 95
23 9 18 99 66 8 3 60 74 91

Matrix 2:

25 27 13 20 63 29 33 48 79 73
63 70 36 10 99 4 8 16 61 77
72 66 66 69 60 97 3 77 52 77
36 82 32 97 92 23 54 75 3 13
96 62 73 16 62 42 38 94 55 11
82 45 3 2 91 10 43 74 34 81
13 69 83 42 0 15 45 59 59 70
88 76 79 15 16 35 17 19 45 40
73 66 44 20 88 63 59 79 91 27
72 37 71 9 10 89 86 45 74 48

1st Multiplication /w Threading:
...
```

```
2nd Multiplication /w Threading:
...

3rd Multiplication /w Threading:
...


Program Execution time: 1626 milliseconds
```

- This is already significantly faster, only taking 1626 ms in this particular run. However, to ensure we are getting the correct result, we can compare it against the non-threaded method to see if we are getting the same result

Comparing with the Gold Standard

- To ensure that our threaded implementation works, we need to compare the output of this, versus our non-threaded code. Both outputs should be the same, the threaded code should only just run faster
- To check this, we can simply subtract the outputs of the threaded/non-threaded code, and the result should just be 0
    - This is what the `verifyResult()` method does:

```java
private static void verifyResult(long[][] result, long[][] goldStandardResult) {
    boolean verificationPassed = true;
    for (int i = 0; i < 1000; i++) {
        for (int j = 0; j < 1000; j++) {
            if (result[i][j] != goldStandardResult[i][j]) {
                verificationPassed = false;
                break;
            }
        }
        if (!verificationPassed) {
            break;
        }
    }

    if (verificationPassed) {
        System.out.println("Verification successful! Results match the gold standar
    } else {
        System.err.println("Verification failed! Results differ.");
    }
}
```

    - We can `for` loop through both results and perform a simple value check to see if each value of the matrix from both matrices, is equal to each other. If there is one unequal value, we know the matrixes aren't the same and something has gone wrong

Task 3 - Thread Pool Implementation

- We can execute this simple matrix program by using multiple threads, which is the job of a thread pool. By implementing a thread pool in this program, we can effectively manage multiple threads and allocate them tasks when a thread is free. This allows us to achieve concurrency within the program

matrixEngineThreadPool.java

- The logic for performing the matrix multiplication is the same code as we saw in the threaded/non-threaded versions of this program. To keep things simple, I have reused this code in the `performMatrixMultiplicationThreaded()` method
- However, where things start to change is the `performMatrixMultiplication` method. This is the main method responsible for creating the thread pool and allocating tasks to each thread.
  - First of all, the method has 3 parameters, `matrix1` , `matrix2` and `numThreads` which is adjustable. The method also returns a `long[][]` which is the resulting matrix after 3 iterations

    ```
    private static long[][] performMatrixMultiplication(long[][] matrix1, long[][] matr
    {

    }
    ```

- To implement our thread pool, we can take advantage of the `ExecutorService` class [5]. This allows us to execute tasks concurrently. Once the thread has been submitted to the `ExecutorService` then the service will execute the task independently

  ```
  //Using ExecutorService because it is a built-in framework for managing threads
  ExecutorService executorService = Executors.newFixedThreadPool(numThreads);

  //Distributing the workload (in this case the 1000x1000 matrix) among multiple threads
  int chunkSize = MATRIX_SIZE / numThreads;

  //Making an empty 1000x1000 matrix to store the result (MATRIX_SIZE is hardcoded to 100
  long[][] resultMatrix = new long[MATRIX_SIZE][MATRIX_SIZE];
  Runnable[] tasks = new Runnable[numThreads];
  ```

  - We use the `.newFixedThreadPool` method to declare a new thread pool based on the number of threads, which in this case is `numThreads`
  - Then we distribute the workload among each thread, which is what `chunkSize` is. We

divide the overall matrix size (in this case 1000), by the number of threads there are, to get an even spread across all the threads.
    - To manage our different threads, we can create a `Runnable` array of tasks and specify how many threads we have before we assign each thread to that array

- Now its time to assign the multiplication tasks to each thread. Note that each thread will be processing portions of the matrix like in the basic threaded implementation

```
//For every thread that exists, assign a range of rows to process
for (int i = 0; i < numThreads; i++) {
    int startRow = i * chunkSize;
    int endRow = (i + 1) * chunkSize;
    tasks[i] = () -> multiplySubMatrix(matrix1, matrix2, resultMatrix, startRow, endRow
    executorService.submit(tasks[i]);
}
```

    - We are `for` looping through each thread and assigning a portion of the matrix to multiply, which is what `startRow` and `endRow` imply.
    - Now we get to assign the multiplication task to each thread in the `tasks` array. For every thread in this array, we are performing the `multiplySubMatrix()` method, providing our 2 matrixes and the specific rows we want to process.
    - The `multiplySubMatrix()` method is important here, because it allows us to multiply parts of the matrix, rather than just doing it all in one go. The code is the same as the original multiplication method, however we are for looping in-between the `startRow` & `endRow`

- Now the thread pool is processing the matrix and multiplying `matrix1` & `matrix2`. Once the threads are finished doing their job, we need to manage the shutdown process

```
//Initiates an orderly shutdown of the thread pool
executorService.shutdown();

//Wait for all the threads to finish their execution since the shutdown has been called
try {
    executorService.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
} catch (InterruptedException e) {
    e.printStackTrace();
}

//Return the final result
return resultMatrix;
```

    - The `.shutdown()` method of the `ExecutorService` will initiate a thread shutdown process, in the order which the threads were created

- We need to wait for every worker thread to finish doing their execution before we shutdown the whole thread pool, which is why we are using the `.awaitTermination()` method, rather than just directly shutting the thread pool down.
- We can also use a `try/catch` block to further improve the stability of the thread pool. If something goes wrong, the program won't go into a panic state

App.java (Execution)

- Executing the thread pool, we can see it's execution times are very similar to the basic threaded implementation, in some cases it definitely runs much faster
- The following execution took place with the following parameters:

```
private static final int MATRIX_SIZE = 1000;
private static final int NUM_THREADS = 4;

long[][] result1 = performMatrixMultiplication(matrices.matrix1, matrices.matrix2, NUM_THRE
//Will be the same for result2 & result 3
```

```
Please choose an option to run this program
1. Run /w No Threading
2. Run /w Threading
3. Run using Thread Pool
4. Verify Threaded vs Non-Threaded
5. Exit Program
3
Matrix 1:

70 2 38 83 73 9 99 21 56 77
42 92 37 16 30 9 5 99 42 18
9 13 82 57 27 37 69 38 36 65
35 61 53 29 42 51 37 96 45 62
69 0 88 1 4 19 45 89 29 44
53 52 64 76 28 87 45 47 64 31
44 11 81 68 72 88 71 78 38 54
37 22 88 18 37 67 16 19 62 94
32 11 72 73 84 13 86 13 4 77
6 23 47 39 60 81 8 23 49 99

Matrix 2:

27 78 47 83 1 34 67 50 15 13
65 50 30 31 17 55 30 54 57 72
88 85 13 20 56 97 24 77 77 7
90 40 33 16 38 23 30 57 17 45
79 86 48 18 96 34 23 81 6 39
25 33 63 27 88 72 0 77 59 5
15 61 2 40 4 69 89 69 23 88
```

```
92 63 45 18 49 14 67 98 13 28
50 19 7 18 94 24 38 63 63 23
21 70 11 89 48 73 30 13 41 64

1st Multiplication /w Thread Pool:
...

2nd Multiplication /w Thread Pool:
...

3rd Multiplication /w Thread Pool:
...

Thread Pool Execution time: 1305 milliseconds
```

- In conclusion, thread pools are a very useful tool to make a program run concurrently. You have absolute control over the distribution of the workload and each worker thread. In the next section, we will test the stability of this thread pool implementation and see if it is good enough for business use

Task 4 - Testing the Thread Pool

- To implement a deliberate instability in the thread pool, we can add a simple probability calculation to decide if a worker thread is to fail or continue its operation as normal
- This simple probability will create an instability within the thread pool, as if one of the threads were to fail, it woudn't pass its result back to the main thread, and thus the matrix multiplication result will be heavily affected. In some cases, this might not even produce results and return empty matrixes.
- To implement this, I have altered the `for` loop where the threads have been created in the `performMatrixMultiplication()` method

```java
for (int i = 0; i < numThreads; i++) {
    int startRow = i * chunkSize;
    int endRow = (i + 1) * chunkSize;
    tasks[i] = () -> {
        multiplySubMatrix(matrix1, matrix2, resultMatrix, startRow, endRow);
        if (Math.random() < terminationProbability) {
            // Terminate the thread without passing back the result
            System.out.println("A thread was terminated!!!!!");
            return;
        }
    };
    executorService.submit(tasks[i]);

    //Code continues to the shutdown...
```

```
}
```

- We perform the multiplication as normal, which is the implicit expression for every thread in `tasks` , perform the `multiplySubMatrix()` method
- In the main method, I have declared an additional parameter called `terminationProbability` which is a random value between 0-1. We use this value to randomly decide if the current thread in the current iteration of the for loop, is to terminate or continue its operation
- The `if` statement precisely does this. We can use `Math.random()` to generate a value between 0-1 and if that number is lower than the `terminationProbability` , then we terminate the thread by a simple `return` . This simple implementation should produce some drastic changes in the performance and the output of the thread pool

Testing the implementation

- The following output shows the output of the thread pool implementation, with the deliberate instability we programmed into the `for` loop
- The `terminationProbability` was set to 0.5 for all 3 iterations for this particular run

```
Please choose an option to run this program
1. Run /w No Threading
2. Run /w Threading
3. Run using Thread Pool
4. Verify Threaded vs Non-Threaded
5. Exit Program
3
Matrix 1:

69 45 58 82 30 3 83 13 64 61
72 10 96 86 94 68 26 32 85 92
60 58 60 54 1 18 27 13 71 31
95 75 54 98 17 82 79 8 28 66
62 33 6 16 79 46 30 8 22 89
14 76 69 37 37 20 33 47 17 18
98 25 33 26 69 52 38 44 83 64
13 45 86 99 22 74 49 32 28 77
15 83 71 74 92 97 42 85 41 77
9 42 33 68 30 93 58 1 38 44

Matrix 2:

32 77 49 58 61 75 36 84 76 10
12 29 66 59 61 55 51 81 73 84
75 6 95 55 19 22 48 6 84 58
35 60 53 64 74 15 37 23 79 26
```

```
16 60 14 94 66 87 53 1 89 44
83 47 59 36 87 67 34 6 10 48
5 77 22 45 38 86 69 48 60 63
43 71 22 16 28 13 71 22 14 73
9 61 7 16 83 25 81 49 14 84
13 80 24 72 30 4 65 12 20 39

A thread was terminated!!!!!
A thread was terminated!!!!!
1st Multiplication /w Thread Pool:

2504354 2492991 2480875 2369951 2465971 2460229 2491758 2481078 2365107 2364038
2534239 2484828 2502472 2462757 2491914 2500684 2492443 2468545 2373341 2417600
2501095 2477567 2551407 2438405 2491925 2490583 2526490 2487510 2422983 2413047
2503971 2518250 2564521 2442400 2481699 2520353 2484635 2423850 2390734 2360839
2556957 2483322 2481986 2357794 2452782 2466852 2452412 2468129 2355774 2372149
2439289 2394994 2437115 2362490 2393915 2433036 2377105 2438974 2229853 2341729
2438138 2470089 2447989 2351914 2381164 2428863 2415698 2401563 2261655 2350813
2592408 2507213 2570535 2461566 2442763 2565443 2481833 2501218 2390416 2427069
2429361 2422518 2475729 2371159 2432198 2420445 2451972 2418059 2292412 2395076
2421932 2500655 2479489 2366149 2510446 2431350 2443594 2488761 2362632 2375295

A thread was terminated!!!!!
A thread was terminated!!!!!
A thread was terminated!!!!!
2nd Multiplication /w Thread Pool:

122731232653 122174795833 124220504556 121265512551 118556324340 118530055961 121862776
125279664610 124681288136 126799874718 123783328684 120996541223 121056735132 124386878
125001187016 124442369307 126588570455 123572082560 120780662570 120829684811 124150144
124356815278 123804217650 125894991777 122932459603 120163735538 120213965076 123529886
123294108672 122735047286 124835013752 121839711287 119130552022 119190927816 122433921
120872535840 120296650943 122337115640 119433779027 116747963416 116773175025 119990776
120391600755 119872778250 121883843239 119027116997 116307929237 116367345163 119572519
125102599153 124549855675 126621961946 123671241660 120848663233 120895856120 124231663
121944264251 121400899593 123425305546 120502573459 117820764554 117840824704 121085587
122482183900 121942118194 124007706388 121070301447 118352589141 118378429854 121680205

A thread was terminated!!!!!
A thread was terminated!!!!!
3rd Multiplication /w Thread Pool:

5870380164606928 6082507864256446 5863270911655169 6033280285635444 5946469756201848 59
5992034515162628 6208526573204860 5984700655077165 6158282434734355 6069634781884868 60
5982080029206709 6198202816783963 5974815250846544 6148134344108289 6059565957777999 60
5950462567071567 6165464311462664 5943235382830961 6115578520933001 6027540945992842 60
5899181249386171 6112292172202768 5891981716783142 6062892332308181 5975616616066307 59
5781600636627866 5990503668571551 5774557047688629 5942005105238595 5856438702859718 58
5760969847128686 5969118497545753 5753943864111260 5920861671575819 5835581113226800 58
5984851206856313 6201092679685576 5977578908248059 6150943103803511 6062348363255672 60
5832896155031512 6043689080229342 5825790935381870 5994771889126500 5908437412549582 58
5860412123940792 6072163516942910 5853271244959644 6023081352674425 5936337042792467 59
```

```
Thread Pool Execution time: 5687 milliseconds
```

- This particular run was very bizzare. It seems like at certain points, some threads are failing. However, we still managed to get an output for all 3 iterations, which is something I didn't expect to see when testing
- We can conclude that regardless of thread failure, we are getting the correct outputs in each iteration as the gold-standard verifications are also passing. This means we have a good, stable implementation of a thread pool.

Conclusion

The implementation of threading, thread pools, and concurrency provides a means to exploit the full potential of modern hardware, enabling businesses to achieve faster and more scalable solutions. In the context of this assignment, several key insights and advantages have been gained through the exploration of non-threaded, multi-threaded, and thread pool methods.

Threading, especially in scenarios involving complex computations like matrix multiplication, significantly improves performance by parallelizing tasks. This is crucial for reducing execution times, meeting business requirements, and achieving real-time processing capabilities. This promotes your application to be very flexible, as threading reduces the demand on your system resources by effectively managing and reusing existing threads.

The lessons learned from coding this matrix multiplication program shows the importance of aligning threading strategies with specific business requirements, striking a balance between performance gains and the inherent complexities introduced by concurrent execution. By understanding these concepts, businesses can unlock the full potential of their computational resources, allowing for the development of more efficient and responsive applications.

References

1. gongshwgongshw 83211 gold badge66 silver badges1111 bronze badges et al. (1959) A long bigger than long.max_value, Stack Overflow. Available at: https://stackoverflow.com/questions/16546038/a-long-bigger-than-long-max-value (Accessed: 23 December 2023).
2. How to multiply matrices (no date) Math is Fun Advanced. Available at: https://www.mathsisfun.com/algebra/matrix-multiplying.html (Accessed: 23 December 2023).
3. Drake, D.G. (1996) Introduction to java threads, InfoWorld. Available at: https://www.infoworld.com/article/2077138/introduction-to-java-threads.html (Accessed: 28 December 2023).
4. Java Threads (no date) W3 Schools. Available at: https://www.w3schools.com

/java/java_threads.asp (Accessed: 28 December 2023).

5. Jenkov, J. (2023) Java ExecutorService, Jenkov.com Tech & Media Labs - Resources for Developers, IT Architects and Technopreneurs. Available at: https://jenkov.com/tutorials/java-util-concurrent/executorservice.html (Accessed: 30 December 2023).