****

**REAL-TIME EMBEDDED SYSTEMS**

**ECTE331**

**PROJECT REPORT**

**Student:**

**Ammar Al Hamadi**

**7809402**

**Teachers:**

**Dr. Abdsamad Benkrid**

**Mr. Kiyan Afsari**

**Part 1**

***Implementation Overview:***

**a) Single-Thread:**

This implementation reads an image and a template, then performs template matching using a single thread. It includes these methods:

* “main” Method:

This method reads the source and the template image, calls “templateMatchingSingleThread” method to perform template matching, and measures and prints the execution time in milli seconds.

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** IOException { |

Figure 1 - Code Snippet for "main" Method Signature

* “templateMatchingSingleThread” Method:

This method calculates the mean absolute differences (MAD) between source image patches and the template, finds the minimum MAD and sets a threshold, and calls “drawRectangles” method to draw a rectangle around the matching regions.

|  |
| --- |
| **public** **static** **void** templateMatchingSingleThread(**short**[][] source\_img, **short**[][] template\_img, BufferedImage image\_source) **throws** IOException { |

Figure 2 - Code Snippet for "templateMatchingSingleThread" Method Signature

* “calculateMeanAbsoluteDifference” Method:

This method calculates the Mean Absolute Difference (MAD) between the source and the template.

|  |
| --- |
| **public** **static** **double** calculateMeanAbsoluteDifference(**short**[][] source\_img, **short**[][] template\_img, **int** x, **int** y) { |

Figure 3 - Code Snippet for "calculateMeanAbsoluteDifference" Method Signature

* “drawRectangles” Method:

The method draws rectangles around the matching regions.

|  |
| --- |
| **public** **static** **void** drawRectangles(BufferedImage sourceImage, **double**[][] absDiffMat, **double** threshold, **int** r2, **int** c2, String outputFileName) **throws** IOException { |

Figure 4 - Code Snippet for "drawRectangles" Method Signature

* “readColourImage” Method:

Reads an image file and converts it to grayscale.

|  |
| --- |
| **public** **static** **short**[][] readColourImage(String fileName) { |

Figure 5 - Code Snippet for "readColourImage" Method Signature

**b) Multi-Thread:**

This implementation divides the template matching task across multiple threads. It includes these methods:

* “main” Method:

This method reads the source and the template image, calls “templateMatchingMultiThread” method, gets the number of available processors, and measures and prints the execution time in milli seconds.

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** IOException { |

Figure 6 - Code Snippet for "main" Method Signature

* “templateMatchingMultiThread” Method:

This method divides the task into chunks and assigns each of them to a thread. Then, it combines the results from the threads. Next, it finds the minimum MAD and sets a threshold. And finally, it calls “drawRectangles” method to mark the matching regions.

|  |
| --- |
| **public** **static** **void** templateMatchingMultiThread(**short**[][] source\_img, **short**[][] template\_img, BufferedImage image\_source, **int** numOfThreads) **throws** IOException, InterruptedException, ExecutionException { |

Figure 7 - Code Snippet for "templateMatchingMultiThread" Method Signature

* “calculateMeanAbsoluteDifference”, “drawRectangles”, and “readColourImage” Methods:

These methods are the same as in the single-thread implementation.

|  |
| --- |
| **public** **static** **double** calculateMeanAbsoluteDifference(**short**[][] source\_img, **short**[][] template\_img, **int** x, **int** y) { |

Figure 8 - Code Snippet for "calculateMeanAbsoluteDifference" Method Signature

|  |
| --- |
| **public** **static** **void** drawRectangles(BufferedImage sourceImage, **double**[][] absDiffMat, **double** threshold, **int** r2, **int** c2, String outputFileName) **throws** IOException { |

Figure 9 - Code Snippet for "drawRectangles" Method Signature

|  |
| --- |
| **public** **static** **short**[][] readColourImage(String fileName) { |

Figure 10 - Code Snippet for "readColourImage" Signature

***Implementation Summary:***

Single thread processes the entire image sequentially, while multi-thread splits the task into chunks that are processed concurrently.

Both implementations read images, convert them to grayscale, perform template matching, and draw rectangles around matched regions. The main difference in multi-threading is the ability to use multiple processors for faster computations.

***Output Results:***

|  |  |  |
| --- | --- | --- |
| Value | Single-Thread (ms) | Multi-Thread (ms) |
| Test Case 1 | 16504 | 2618 |
| Test Case 2 | 15757 | 2530 |
| Test Case 3 | 16119 | 2541 |
| Average Execution Time | **16127** | **2563** |

Table 1 - Average Execution Time in 3 Test Cases

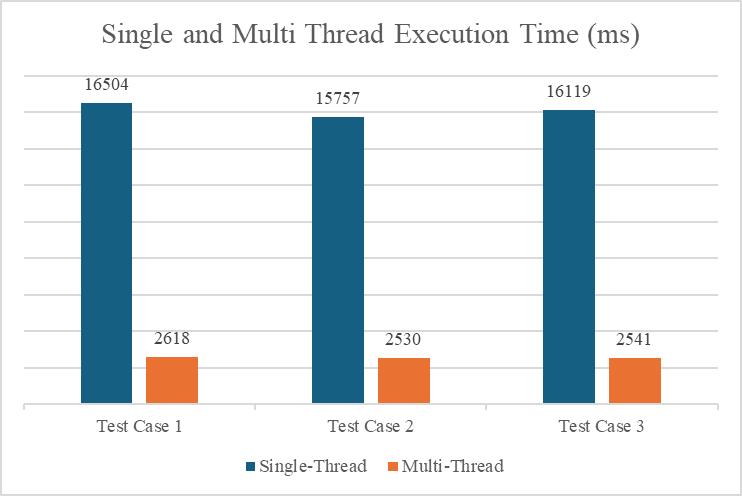
****

Figure 11 - Execution time in Milliseconds

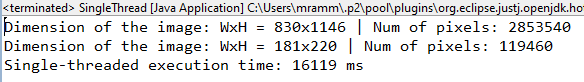
****

Figure 12 - Test Case 3 Single-Thread

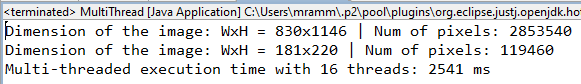
****

Figure 13 - Test Case 3 Multi-Thread

The multi-threaded implementation works much faster than the single-threaded one, as shown by the quicker execution times. This speedup comes from running tasks at the same time and making better use of the CPU, especially on computers with multiple cores.

|  |  |
| --- | --- |
| A card with a card of spades  Description automatically generated with medium confidence  Figure 14 - Single-Thread Output Image | A card with a card of spades  Description automatically generated with medium confidence  Figure 15 - Multi-Thread Output Image |

**Part 2**

***Requirements:***

**a) Final Correct Values of Shared Variables**

|  |
| --- |
| **public** **static** **int** calculateSum(**int** n) {  **int** sum = 0;  **for** (**int** i = 1; i <= n; i++) {  sum += i;  }  **return** sum;  } |

Figure 16 - Code Snippet for Calculating the Sum

|  |
| --- |
| *A1*.set(SumUtil.*calculateSum*(500));  *A2*.set(*B2*.get() + SumUtil.*calculateSum*(300));  *A3*.set(*B3*.get() + SumUtil.*calculateSum*(400));  *B1*.set(SumUtil.*calculateSum*(250));  *B2*.set(*A1*.get() + SumUtil.*calculateSum*(200));  *B3*.set(*A2*.get() + SumUtil.*calculateSum*(400)); |

Figure 17 - Code Snippet for Calling calculateSum

**b) Synchronization**

To ensure the thread functions are executed in the specified order irrespective of the operating system’s thread scheduling, CountDownLatch was used. This class makes sure that one or more threads wait until a set of operations being performed in other threads are completed.

* latchA1 is used to signal ThreadB that FuncA1 has finished.
* latchB1 is used to signal ThreadA that FuncB1 has finished.
* latchB2 is used to signal ThreadA that FuncB2 has finished.
* latchA2 is used to signal ThreadB that FuncA2 has finished.
* latchB3 is used to signal ThreadA that FuncB3 has finished.
* latchA3 is used to signal ThreadB that FuncA3 has finished.

|  |
| --- |
| **private** **static** CountDownLatch *latchA1* = **new** CountDownLatch(1);  **private** **static** CountDownLatch *latchB1* = **new** CountDownLatch(1);  **private** **static** CountDownLatch *latchB2* = **new** CountDownLatch(1);  **private** **static** CountDownLatch *latchA2* = **new** CountDownLatch(1);  **private** **static** CountDownLatch *latchB3* = **new** CountDownLatch(1);  **private** **static** CountDownLatch *latchA3* = **new** CountDownLatch(1);  *latchA1* = **new** CountDownLatch(1);  *latchB1* = **new** CountDownLatch(1);  *latchB2* = **new** CountDownLatch(1);  *latchA2* = **new** CountDownLatch(1);  *latchB3* = **new** CountDownLatch(1);  *latchA3* = **new** CountDownLatch(1);  *latchA1*.countDown();  *latchB2*.await();  *latchA2*.countDown();  *latchB3*.await();  *latchA3*.countDown();  *latchB1*.countDown();  *latchA1*.await();  *latchB2*.countDown();  *latchA2*.await();  *latchB3*.countDown(); |

Figure 18 - Code Snippet for Related Latch Codes

**c) SumUtil Class**

A new utility class was created and called SumUtil with a static method calculateSum that computes the sum using a loop. Then the new utility method is called in ThreadSync class.

|  |
| --- |
| **package** project1;  **public** **class** SumUtil {  **public** **static** **int** calculateSum(**int** n) {  **int** sum = 0;  **for** (**int** i = 1; i <= n; i++) {  sum += i;  }  **return** sum;  }  } |

Figure 19 - Code Snippet for SumUtil Class

**d) Extending The Main for A High Number of Iterations**

The main function was extended to run the threads for 1000 iterations to ensure the synchronization and execution is correct. The final values were checked in each iteration.

|  |
| --- |
| **private** **static** **final** **int** ***MAX\_ITERATIONS*** = 1000;  **for** (**int** i = 0; i < ***MAX\_ITERATIONS***; i++) |

Figure 20 – Code Snippet of Related Codes for Extending Main for High Iterations

***Output Results:***

The output of this program is:

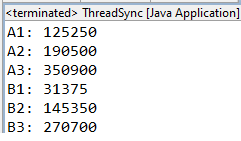
****

Figure 21 - Output of the Java Code

Which is the same as what have been calculated manually in part a.