Chapter 8 - Exercise 1

In Java (because we want to use threads) implement the serial version for matrix multiply.
In Java create a Thread version that always uses 8 threads.
1. How can we test both give the same results?
    - Because they give the same results does this mean they are both correct?
2. Compare the serial with the thread version.
    - An n of 4096 should work well to get good time values.
3. How many cores do these computers have?
4. Are we making optimal use of the number of cores?

Grading Rubric
_____ (2 Points) Code for serial version of Matrix Multiply
_____ (2 Points) Code for threaded version of Matrix Multiply
_____ (2 Points) Test function in each version checking they work properly
_____ (2 Points) Gather time information for both versions
_____ (1 Points) How many cores does your computer have - screen shot showing this information
_____ (1 Points) Could we have more threads with explanation

Assignment Answers

1. How can we test both give the same results?

We can test that both the serial and threaded versions give the same results by using a verification function that multiplies two small, known matrices and compares the output of both implementations against the expected result. If both methods produce the same output for the same input, we can conclude they are functioning correctly for that case.

- **Does this mean they are both correct?**

Not necessarily. Just because they produce the same output for specific test cases does not guarantee that both implementations are correct in general. It is essential to conduct additional tests with a variety of input sizes and values to ensure correctness across different scenarios.

2. Compare the serial with the thread version.

The serial version of matrix multiplication processes the multiplication using a

single thread, while the threaded version utilizes 8 threads to divide the workload. The expected outcome is that the threaded version should complete the multiplication faster, especially for larger matrices like 4096x4096, due to concurrent execution.

3. **How many cores do these computers have?**

You can determine the number of cores on your computer by running the following code snippet in Java:

```
int cores = Runtime.getRuntime().availableProcessors();
System.out.println("Number of Cores: " + cores);
```

Alternatively, you can check your system settings or use tools like Task Manager on Windows or Activity Monitor on macOS. However, I believe we are being asked to do via code so I did the coding way.

4. **Are we making optimal use of the number of cores?**

If the number of threads in the threaded version matches or is less than the number of available cores, we are making optimal use of the cores. If the number of threads exceeds the number of cores, it may lead to context switching, which can reduce performance. Thus, it is ideal to set the number of threads equal to the number of available cores for maximum efficiency.

TEST METHOD:

1. Matrix Generation:

- Generate two random matrices A and B of size n x n using the generateMatrix(n) method.
- This step prepares the data for multiplication. Random matrices are created to test the performance and correctness of both implementations under typical conditions.

2. Define Known Matrices for Verification:

- Create small, hardcoded matrices A and B for testing the correctness of the implementations. For example:

```
int[][] A = {{1, 2}, {3, 4}};
int[][] B = {{2, 0}, {1, 2}};
```

- Hardcoded matrices are used because their expected multiplication results are

known (like you said in class, we can do small size matrices by hand and test with those). This allows for a straightforward validation of both implementations against a predictable outcome.

3. Expected Result Calculation:

- Manually compute the expected result of multiplying the known matrices A and B. The expected result is:

```
int[][] expected = {{4, 4}, {10, 8}};
```

- By calculating the expected output, we establish a baseline against which we can compare the results from the serial and threaded implementations to ensure accuracy.

4. Run Serial Matrix Multiplication:

- Call the serial multiplication method to compute the product of matrices A and B.
- This step tests the performance of the serial version of the algorithm, allowing us to measure its execution time and correctness.

5. Run Threaded Matrix Multiplication:

- Call the threaded multiplication method to compute the product of matrices A and B.
- This step tests the performance of the multithreaded version of the algorithm, allowing us to compare execution time against the serial version.

6. Verify Correctness:

- Use the areEqual(matrix1, matrix2) method to compare the results of the serial and threaded implementations against the expected output.
- This ensures that both implementations yield the same correct result. If they do not match, it indicates a potential issue with one or both of the implementations.

7. Timing Measurements:

- Measure the execution time for both the serial and threaded methods using System.nanoTime().
- This provides performance metrics, allowing us to evaluate the efficiency of each approach. The results can highlight the benefits of multithreading for larger matrices.

8. Check for Optimal Core Utilization:

- Retrieve the number of available cores using Runtime.getRuntime().availableProcessors().
- This allows us to determine if the number of threads in the threaded version is optimal for the hardware. Ideally, the number of threads should match the number of cores to ensure maximum performance without unnecessary context switching.

Output:

```
[Running] cd "/Users/sajjadalsaffar/Documents/School/
Shippensburg University/Fall 2024/CSC523/Exercises/Chapter 8/"
&& javac MatrixMultiplyTest.java && java MatrixMultiplyTest
Serial and Threaded versions are verified as correct on a known
example.
Serial Time (ms): 266866
Threaded Time (ms): 85906
Both methods give the same result for the large matrix.
Number of Cores: 8
Using optimal or close-to-optimal threads for available cores.

[Done] exited with code=0 in 354.684 seconds
```

Conclusion

In this test, both implementations produced identical results for a small, known example matrix as well as for a larger 4096x4096 matrix, confirming their correctness.

Performance Results:

- The serial version took approximately 266,866 milliseconds (about 4.4 minutes).
- The threaded version, utilizing 8 threads, significantly reduced execution time to around 85,906 milliseconds (about 1.4 minutes), achieving a performance improvement of roughly 3 times.

Optimal Thread Utilization:
With an 8-core system, using 8 threads aligns well with the hardware's capacity(on both my MacBook and Windows PC), enabling efficient parallel processing without overloading the system with unnecessary context switches. This test shows that the threaded version can leverage multi-core processors to

enhance computational speed, especially for large-scale matrix operations such as 4096.

Conclusion Summary:
The multithreaded implementation is notably faster and more efficient for large matrices, while the serial approach is simpler(3 for loops lol) but slower. Using a thread count matching the number of available cores can optimize performance on multi-core systems, highlighting the effectiveness of parallel processing in computational tasks like matrix multiplication.