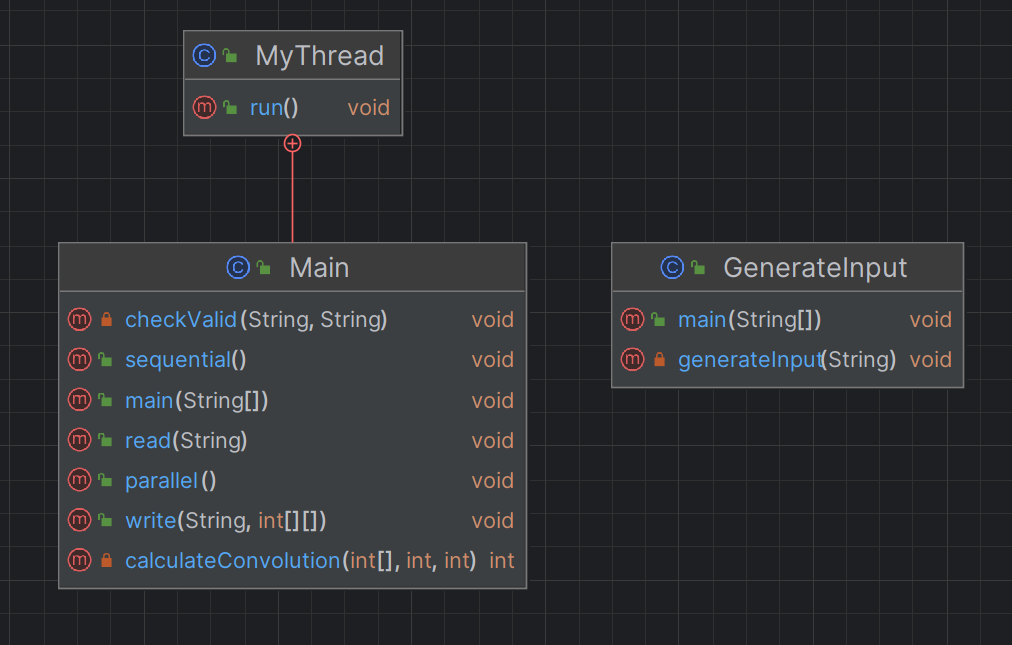
Documentation

**Java:**

|  |  |  |
| --- | --- | --- |
| Tip Matrice | Nr threads | Timp executie |
| N=M=10 si n=m=3 | 0 | 0.07357 |
| 2 | 1.81225 |
| N=M=1000 si n=m=3 | 0 | 26.4592 |
| 2 | 32.98289 |
| 4 | 37.56466 |
| 8 | 46.7661 |
| 16 | 50.83455 |
| N=M=10000 si n=m=3 | 0 | 744.61571 |
| 2 | 321.87792 |
| 4 | 267.29612 |
| 8 | 317.31016 |
| 16 | 255.51115 |

**C++:**

|  |  |  |  |
| --- | --- | --- | --- |
| Tip Matrice | Tip alocare | Nr threads | Timp executie |
| N=M=10 si n=m=3 | dinamica | 0 | 0.29786 |
| dinamica | 2 | 1.62144 |
| N=M=1000 si n=m=3 | dinamica | 0 | 3145.701 |
| dinamica | 2 | 5505.001 |
| dinamica | 4 | 7412.802 |
| dinamica | 8 | 9396.743 |
| dinamica | 16 | 9683.15 |
| N=M=10000 si n=m=3 | dinamica | 0 | 3.09E+06 |
| dinamica | 2 | 1.82E+06 |
| dinamica | 4 | 1.56E+06 |
| dinamica | 8 | 2.20E+06 |
| dinamica | 16 | 2.44E+06 |



**Data distribution:**

The solution employs data distribution row-wise. Data distribution is achieved by dividing the rows of the matrix among different threads. Each thread is responsible for processing a portion of the rows in parallel. This distribution is based on the number of threads specified as a command-line argument (p), and each thread processes a range of rows.

**Solving method:**

The solving method involves reading input data, initializing a synchronization barrier, and then executing a convolution operation on a matrix using multiple threads. If the number of threads specified (p) is zero, the convolution is performed sequentially using the sequential function. In the case of parallel execution (p > 0), the matrix is divided into segments, and each thread is responsible for processing a distinct range of rows. Multiple threads are created to perform parallel convolution, with each thread calculating convolution values while considering edge conditions. Threads synchronize using the barrier, and the execution time is measured and outputted. The resulting matrix is written to an output file, and for p > 0, the code checks the result's validity by comparing it to a reference file (valid.txt).

**Analysis:**

**C++:**

N=M=10 and n=m=3: In this case, the overhead of creating and managing threads outweighs the benefits of parallelism, resulting in a slower execution time when using 2 threads compared to a single-threaded execution.

N=M=1000 and n=m=3: In this case, parallelism becomes advantageous, but it's important to note that adding more threads does not necessarily result in better performance. The best average time is achieved with 2 threads.

N=M=10000 and n=m=3: Similar to the previous case, parallelism is advantageous, but the number of threads that provides the best performance varies with the matrix size. In this case, using 4 threads results in the lowest execution time.

In general, dynamic allocation is used, and the number of threads should be carefully chosen based on the specific matrix and kernel size. Adding more threads does not always lead to better performance, and there is an optimal number of threads that minimizes execution time. It's essential to balance the overhead of thread creation and management with the benefits of parallelism to achieve the best performance for a given problem size.

**Java:**

Matrix Size N=M=10 and n=m=3: The sequential version is more efficient with an execution time of 0.07357. Introducing 2 threads increases the execution time to 1.81225, making it less efficient than the sequential version.

Matrix Size N=M=1000 and n=m=3: The sequential version is still the most efficient with an execution time of 26.4592. Increasing the number of threads from 2 to 16 results in higher execution times, suggesting that parallelism is not advantageous for this configuration.

Matrix Size N=M=10000 and n=m=3: In the case of a 10000x10000 matrix and a 3x3 kernel, the sequential version remains the most efficient with an execution time of 744.61571. Using 2 threads significantly improves efficiency, reducing the execution time to 321.87792. However, increasing the number of threads to 4, 8, and 16 does not provide significant further improvements.

In summary, based on the provided data, it appears that parallelism in Java is not always advantageous. The effectiveness of parallelism depends on factors such as the matrix size and kernel dimensions, with the largest matrix size and a 3x3 kernel showing potential for improvement with 2 threads but limited gains with more threads.

**Java vs C++:**  
In C++, dynamic allocation shows variable thread efficiency, emphasizing the importance of selecting an optimal thread count. In Java, parallelism is less advantageous across different matrix sizes and kernel dimensions, with the sequential version often outperforming multithreaded versions.