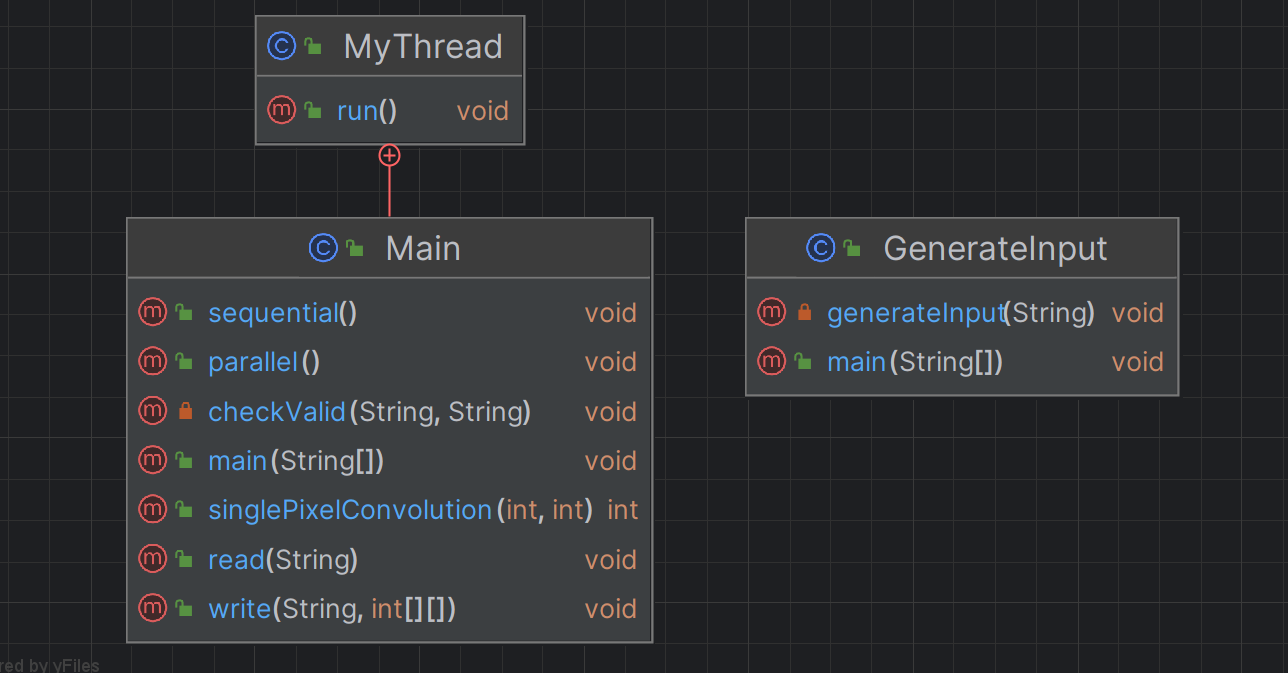
Documentation

**Java:**

|  |  |  |
| --- | --- | --- |
| Tip Matrice | Nr threads | Timp executie |
| N=M=10, n=m=3 | 0 | 0.28533 |
| 4 | 1.90037 |
| N=M=1000, n=m=5 | 0 | 118.86044 |
| 2 | 122.61104 |
| 4 | 83.19647 |
| 8 | 143.77416 |
| 16 | 230.86484 |
| N=10 M=10000, n=m=5 | 0 | 73.3508 |
| 2 | 95.36487 |
| 4 | 79.47552 |
| 8 | 99.81324 |
| 16 | 92.62437 |
| N=10000 M=10, n=m=5 | 0 | 24.03412 |
| 2 | 39.53741 |
| 4 | 30.90139 |
| 8 | 38.34527 |
| 16 | 32.06596 |

**C++:**

|  |  |  |  |
| --- | --- | --- | --- |
| Tip Matrice | Tip alocare | Nr threads | Timp executie |
| N=M=10, n=m=3 | static | 0 | 0.02519 |
| 4 | 1.40473 |
| dinamic | 0 | 0.02838 |
| 4 | 1.31185 |
| N=M=1000, n=m=5 | static | 0 | 610.8876 |
| 2 | 318.9841 |
| 4 | 235.4733 |
| 8 | 195.3584 |
| 16 | 198.5158 |
| dinamic | 0 | 819.7082 |
| 2 | 423.7344 |
| 4 | 288.2624 |
| 8 | 197.3782 |
| 16 | 212.1616 |
| N=10 M=10000, n=m=5 | static | 0 | 98.23997 |
| 2 | 33.66939 |
| 4 | 26.51166 |
| 8 | 17.48938 |
| 16 | 21.68759 |
| dinamic | 0 | 64.88434 |
| 2 | 40.38211 |
| 4 | 33.21173 |
| 8 | 18.85278 |
| 16 | 22.69063 |
| N=10000 M=10, n=m=5 | static | 0 | 86.40045 |
| 2 | 45.64083 |
| 4 | 30.34497 |
| 8 | 21.60111 |
| 16 | 20.37873 |
| dinamic | 0 | 83.74107 |
| 2 | 44.37765 |
| 4 | 29.4205 |
| 8 | 19.68272 |
| 16 | 20.62719 |



**Data distribution:**

The solution employs data distribution based on the size of the input. It chooses between distributing data row-wise or column-wise, depending on which dimension has fewer elements. This decision aims to balance the workload effectively.

**Solving method:**

The code uses an algorithm that divides the workload among a specified number of threads (read from the command line). It distributes the workload evenly by dividing the number of rows/columns by the number of threads, taking any remainings into account. The code calculates two starting and ending positions, which are updated at each thread initialization.

**Analysis:**

**C++:**

Static vs. Dynamic Allocation: Static memory allocation is more time-efficient compared to dynamic allocation. This is because static allocation occurs at compile time in a known space on the heap, while dynamic allocation happens at runtime, which consumes RAM.

Matrix Size 10x10, Kernel 3x3: The sequential version is better due to the overhead of starting three threads, which outweighs the benefits of parallelism.

Matrix Size 1000x1000, Kernel 5x5: The sequential version is inefficient, and parallelism becomes advantageous. The best average time is achieved with 8 threads, while using 16 threads is slightly less efficient.

Matrix Size 10x10000, Kernel 5x5: Variants with 2-4 threads are the most efficient because a larger number of threads doesn't provide significantly better parallelization.

Matrix Size 10000x10, Kernel 5x5: Parallel versions are more efficient, with the maximum number of threads providing the best performance.

**Java:**

In Java, the sequential version is often faster due to compiler and virtual machine-level optimizations, with one exception (10x10000 matrix) where parallelism improves efficiency.

Matrix Size 10x10, Kernel 3x3: For small matrices (10x10) with a 3x3 kernel, the sequential version is more efficient in Java due to the overhead of starting multiple threads.

Matrix Size 1000x1000, Kernel 5x5: With larger matrices (1000x1000) and a 5x5 kernel, the sequential version becomes inefficient, and parallelism is advantageous. The best performance is often achieved with 8 threads.

Matrix Size 10x10000, Kernel 5x5: For a matrix of 10x10000 and a 5x5 kernel, Java's efficiency peaks with 2-4 threads, as additional threads may not significantly improve parallelization.

Matrix Size 10000x10, Kernel 5x5: In cases with a matrix of 10000x10 and a 5x5 kernel, Java's parallel versions are more efficient, with a higher number of threads providing the best performance.

**Java vs C++:**

In general, C++ threads are faster than Java threads, but there are cases where Java's sequential execution is comparable to C++ parallel execution in terms of performance.