Assignment 01: report

Group member: Farinelli Ettore (ettore.farinelli@studio.unibo.it), Ghignatti Nicolò (nicolo.ghignatti@studio.unibo.it)

Overview of the problem

In a concurrent boids simulation, thousands of independent boids update their positions and velocities based on local informations. Since these updates depend on data shared across many boids, due to the dependencies on shared data, is required to avoid race conditions and, in the meanwhile, guarantee a performance gain from the serial version.

Main problems

- **Race conditions** → Each boid reads the positions of its neighbors and computes new velocities. If multiple threads update shared boid data simultaneously, race conditions may occur.
- **Synchronization and barrier** → Concurrency requires synchronizing threads at key stages. In our case, after all boids have finished calculating their new velocities, and again after they have finished updating their positions and are ready to be displayed
- **Load balancing** → To guarantee good performance a proper balance of work on each thread is the minimum.

Architecture, design and strategies

There are no drastic changes to the program architecture from the version provided from the teacher, the only thing introduced is a `BoidsSimulator` interface, allowing a fast swap between the different versions of the simulators.

The main strategy adopted involves assigning a subset of the boids to each thread, which will perform updates for all boids in its subset.

Our strategy focus on avoiding to recreate the `Threads` every iteration of the main loop but, instead, keep the main loop inside them and recreate them when a change of the number of boids occurs, here are three strategies for each approach, following a brief pseudocode:

- **Platform Threads → F**or this solution the number of processors has been decided using different values and comparing the performance of all of them, in this case the crucial point is balancing the load of each thread to avoid that some of them finishes before the others.

```
divide the number of boids for the number of threads and assign to each thread a boid subpart

for i from 0 to number_of_threads:
    Thread t = new Thread(() ⇒ {
        update boids velocity
        wait for all threads to finish
        update boids position
    });
    t.start();

join all threads
```

- **Virtual Threads** → For this solution we assumed a number of threads equal to the number of boids

```
for each boids:
    t = new VirtualThread(() ⇒ {
        update the velocity of the boid assigned
        wait for all threads to finish
        update the position of the boid assigned
    });
    t.start();
```

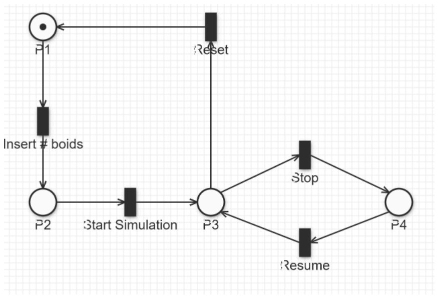
- **Executor** → For this solution, the number of tasks was determined based on previous results. However, the crucial challenge was synchronization, since using barriers within tasks had to be avoided, making synchronization more difficult. There were two possible solutions:
  - by doing busy waiting
  - by using futures

  We choose to use futures to wait all the computations to finish.

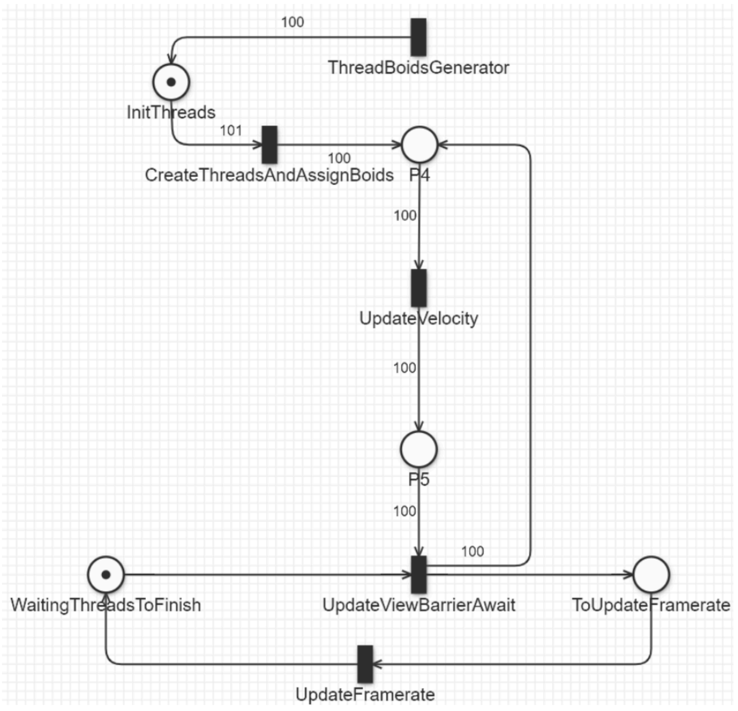
  ```
  for each boid in boids:
      executor.execute(b.updateVelocity())

  wait all boids velocity to be computed by using futures

  for each boid in boids:
      executor.execute(b.updatePos())

  wait all boids position to be computed by using futures
  ```

  In all three solutions the crucial point has been to synchronize all the threads between them and to synchronize all the threads with the view (if present), because the update of the view should happen after all threads finished to compute the positions of all boids. The main strategy consisted in developing an our implementation of a cyclic barrier and using an instance of it to synchronize all threads with each other and another one to synchronize the threads with the view.

Petri nets



General petri net which describe the main flow of the program



On the right there is a more specific petri net representing the simulation running and communicating with the view (represented by the `UpdateViewBarrierAwait` square).

The bottom cycle represent the main thread which decide when to update the view.

The middle/top part, instead, represent the workflow of the threads, from their creation and their interactions with the boids.

Performance

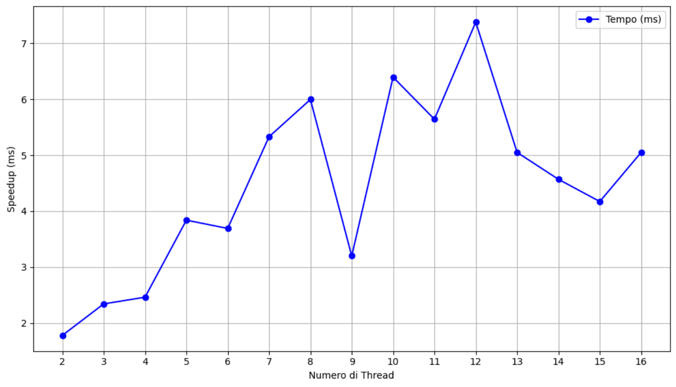
To calculate the speedup we used the following formula:

$$Speedup = \frac{T_s}{T_p}$$

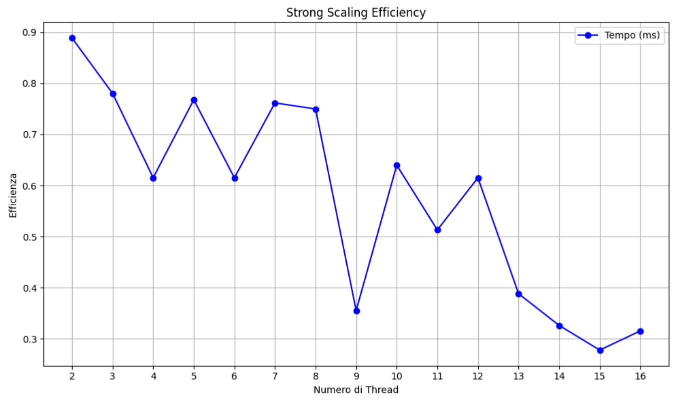
where:

- T_s is the time that our program used to update velocity and position of all boids, in our case is calculated by an average of hundreds of iteration and the time given is $95,96039604$ms
- T_P is the parallel time and has 3 possible values:
  - Using platform thread we have a top score of $13,33529412$ms with $P = 12$

  Follows two plots to visualize the speedup and the efficiency of the program, the second one is calculated as follow: $E_n = \frac{S_n}{n}$



Plot for the speedup of our program according to the core



Strong scaling efficiency plot

  - Using virtual thread we have $32,36298932$ms with $P = NumberOfBoids$
  - Using the task executor we have $40,96103896$ms with $NumberOfFixedThread = 12$

Giving us three different speedup:

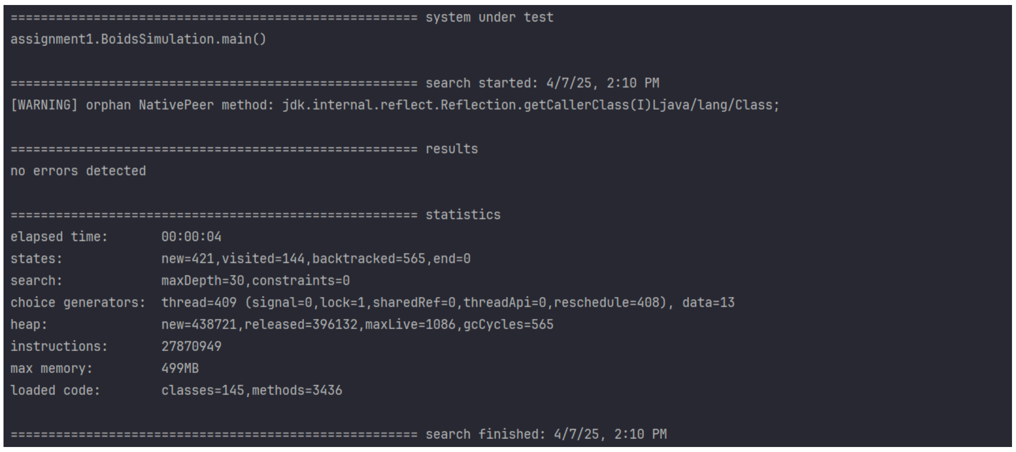
- **Platform Thread** solution has a speedup of 719% or 7,19
- **Virtual Thread** solution has a speedup of 295% or 2,95
- **Task executor** solution has a speedup of 234% or 2,34

Although the proper way to calculate a program's speedup should use

T_1 rather than T_s, since the two may differ, we followed the slides and used the serial execution time. The number of boids used for testing was 5000, with each run consisting of 1500 iterations (60 seconds in total)."

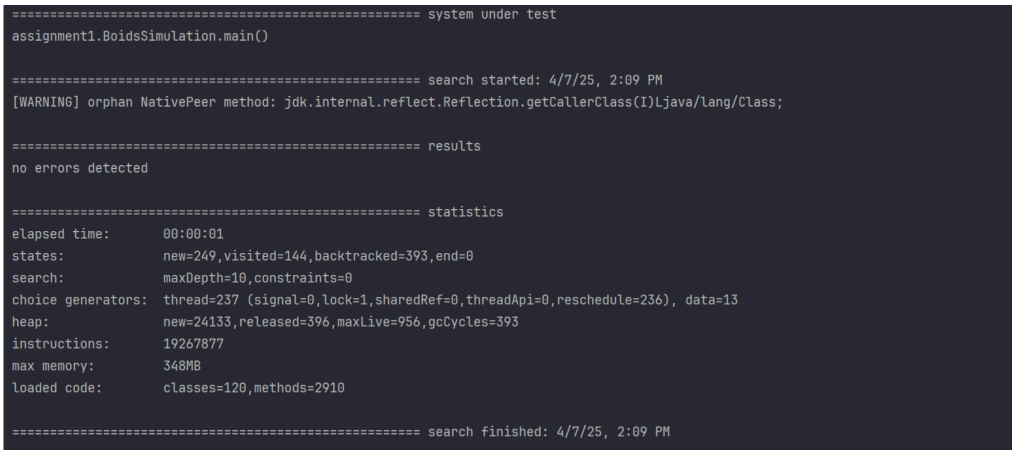
JPF verification

- **Task**



```
====================================================== system under test
assignment1.BoidsSimulation.main()

====================================================== search started: 4/7/25, 2:10 PM
[WARNING] orphan NativePeer method: jdk.internal.reflect.Reflection.getCallerClass(I)Ljava/lang/Class;

====================================================== results
no errors detected

====================================================== statistics
elapsed time:       00:00:04
states:             new=421,visited=144,backtracked=565,end=0
search:             maxDepth=30,constraints=0
choice generators:  thread=409 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=408), data=13
heap:               new=438721,released=396132,maxLive=1086,gcCycles=565
instructions:       27870949
max memory:         499MB
loaded code:        classes=145,methods=3436

====================================================== search finished: 4/7/25, 2:10 PM
```

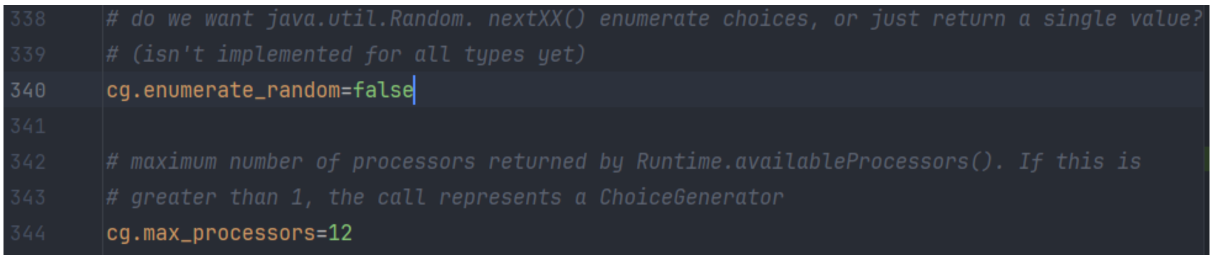
- **Platform**



```
====================================================== system under test
assignment1.BoidsSimulation.main()

====================================================== search started: 4/7/25, 2:09 PM
[WARNING] orphan NativePeer method: jdk.internal.reflect.Reflection.getCallerClass(I)Ljava/lang/Class;

====================================================== results
no errors detected

====================================================== statistics
elapsed time:       00:00:01
states:             new=249,visited=144,backtracked=393,end=0
search:             maxDepth=10,constraints=0
choice generators:  thread=237 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=236), data=13
heap:               new=24133,released=396,maxLive=956,gcCycles=393
instructions:       19267877
max memory:         348MB
loaded code:        classes=120,methods=2910

====================================================== search finished: 4/7/25, 2:09 PM
```

The usage of Java version 21 in Virtual Thread implementation makes impossible to verify the solution with JPF

Both the run has been tested with a little edit on the `jpf.properties` file:



```
# do we want java.util.Random. nextXX() enumerate choices, or just return a single value?
# (isn't implemented for all types yet)
cg.enumerate_random=false

# maximum number of processors returned by Runtime.availableProcessors(). If this is
# greater than 1, the call represents a ChoiceGenerator
cg.max_processors=12
```

The only property changed is the maximum number of processors, because in our program we use the function `Runtime.availableProcessors()`, so we tested it with multiple values, reaching the realistic value of 12.

The "enumerate_random" property is by default `false`, but we tested it also with the `true` value, increasing only the time spent verifying the program.