PCD-Assignment01

Alessandro Agosta

April 8, 2025

Contents

1 Introduction

In this report, I will analyze the process of applying parallel computing to the simulation of a flock of birds. This flocking simulation abstracts from the concept of birds referring to them as "boids", shortened for "bird-oid-object". These boids follow 3 simple rules:

- **"separation" steering to avoid excessive crowding of local "flockmates".**
- **"alignment" steering to follow the local "flockmates" directions'.**
- **"cohesion" steering to remain near the local "flockmates".**

In my implementation, based on Java, I will use Java's provided functionalities to compute this simulation.
Such functionalities include:

- **Java multi threaded programming using the platform threads.**
- **Task-based approach using the Java Executor Framework.**
- **Java Virtual Threads a more lightweight framework that involves virtual threads.**

2 Analysis

In this section, I will analyze any dependency in the algorithm.
In the base implementation, each iteration consists in computing each boid's new position and velocity through these steps:

1. **Collecting other nearby boids in a certain radius.**
2. **Computing the separation value of the selected boid in relation to the positions of its nearby boids.**
3. **Computing the alignment value of the selected boid in relation to its nearby boids' velocities.**
4. **Computing the cohesion value of the center of the flock in relation to the position of the selected boid.**
5. **Computing the new velocity of the selected boid.**
6. **Computing the new position of the selected boid.**

We can already underline some dependencies in the data necessary for each iteration of the simulation that need to be addressed for a parallel implementation. In the above-mentioned steps, we found that **step 1.** holds a critical role in the correct parallel implementation of this algorithm.

I've found that we must ensure that no boid is collecting its nearby boids' whilst these are being calculated themselves; we will act as such to guarantee that no incorrect value is read or written.
Lastly I've discovered that steps **5**. and 6. must always be done last and follow the order of first updating a boid's velocity and then its position.
As for the remaining steps, only correct order of execution must be maintained.

3 Design

In this section I will explore the choices behind each implementation's design.

3.1 Java Platform Threads

In using platform threads we encounter typical concurrency problems in the form of race conditions and starvation; with the intent of ruling out these possible critical failures in this concurrent implementation, these following design choices were taken:

A. **Initialization of "worker" elements**

B. Workers-System synchronization structures

C. Initial partition of the total boids

D. Structures of synchronization between workers

In this implementation, I chose a Master-Worker architecture; where a "master" is tasked with the synchronization and work distribution between its assigned "workers".
At the start of the simulation, every worker is put on hold until there's pending work to manage and resolve; upon job acquisition each worker is instructed to compute its workload and upon completion each worker is again put on hold for more work.
In this approach, platform thread initialization requires a non insignificant overhead to the performances, with this is in mind I've chose to instantiate the platform threads once and reuse them when necessary. The master creates a finite number of workers and to each distributes an equal amount of work; the number of workers created is given by the number of CPU cores on the machine currently running the simulation; the workload is constituted by a number of assigned boids to compute.
As mentioned before, a correct implementation requires a precise computational order; in order to do so, I've divided each iteration of the simulation in 3 major computing blocks:

1. **Reading nearby boids**

2. Writing each new velocity

3. Updating each new position

These are the main jobs each worker goes through for each partition of boids assigned. Initially each worker goes through each of its assigned boids and memorizes their respective nearby boids; here we encounter our first workers-specific synchronization structure, namely a barrier, to prevent workers from starting the next job on the list before every other worker has completed this "reading" task.
The workers then all start the "writing" and "updating" blocks, making sure to follow this exact order: first writing, then updating.
When these computational blocks are done, each worker puts itself on hold until all workers have finished; then workload allocation and boids computation will start anew when the master gives the signal. Differently from platform threads, I've noticed that tasks creation's overhead does not amount to much performance loss and so I've decided to adopt a marginally less efficient solution for a clearer and easier implementation.

3.2 Task-based Approach

In this approach, I've decided that workload distribution could be more fine-grained that the previous approach with platform threads. As such, instead of allocating partitions of boids, each boid's computation becomes tasks for a main "executor" to compute.
Standard execution's order still applies in this approach, before comes a "reading" phase where each boid "reads" its neighbors, then a "writing" phase updates each boid's velocity and position.
In this approach synchronization is simpler than with platform threads, as Java already provides function to invoke, compute and wait task's completion.

3.3 Virtual-Thread Approach

In this approach, the same fine-grained distribution of workload as the Task-based approach can be done; also similarly workload distribution was divided into 2 major jobs: reading and writing. Since using Virtual Threads causes little to no overhead in thread creation, I've decided to re-instantiate and assign work to new virtual threads every time work is pending. In the same manner for the task-based approach, synchronization is required between the reading phase and writing phase.

4 Behavior

In this section I will present a more in depth behavior description of each approach, while also illustrating the systems using Petri Nets.

4.1 Platform Threads Approach

I've previously mentioned a brief explanation of how synchronization was designed for the platform threads approach by introducing two cases: a system-workers synchronization and an in between-workers one.
During the first setup of the simulation's environment, platform threads are initialized and put on hold, waiting on a synchronization structure called "Barrier". A barrier is a structure in which, given a finite number of parties, unless all subscribed participants arrive, all arrived will wait.
In this application, three barriers were used; two were assigned for system-workers synchronization and one was designed for coordinating work between threads.

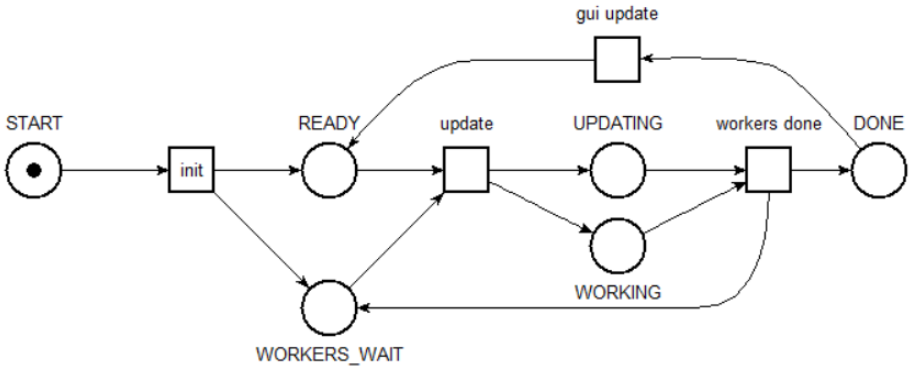

Figure 1: Brief Petri Net of the platform threads approach

In the Figure 1 shown above, we can see the synchronization barriers used, precisely in the Petri Net transitions': "update" and "workers done".
These two transitions translate to barriers that synchronize respectively the workers to the master and the master to its workers; to be precise in the "update" transition we have a barrier that requires all workers with the master activating such barrier to start computation, then in the "workers done" transition we have the master on hold for each worker to be done computing.
We then have the in between workers barrier which has been simplified in the "WORKING" place, this barrier waits for all workers to be done in the already mentioned "reading" phase, where each boids "reads" its nearby other boids.

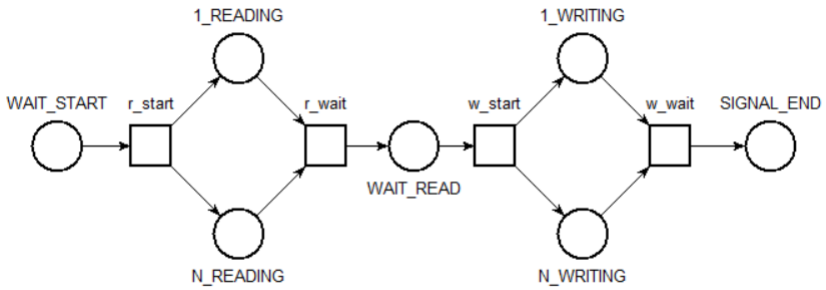

Figure 2: Petri Net of workers

In figure 2, I've highlighted the synchronization between workers, where we have a synchronization in "r_wait" and in "w_wait" where we await each workers job's completion; to then synchronize with the system with the "WAIT_START" and "SIGNAL_END" places which would append respectively to the Figure 1's "update" and "workers done" transitions.

4.2 Task-based Approach

As mentioned during the task-based approach's design, synchronization is simpler when using independent computational loads such as tasks; with synchronization events mainly regarding work start and work done. Although similar to the platform threads' Petri Net, this approach has no explicit "workers" on hold, but rather an "execution service" tasked with receiving one or many independent workloads and executing them; as such the resulting Petri Net doesn't have a "waiting" place for its workers.

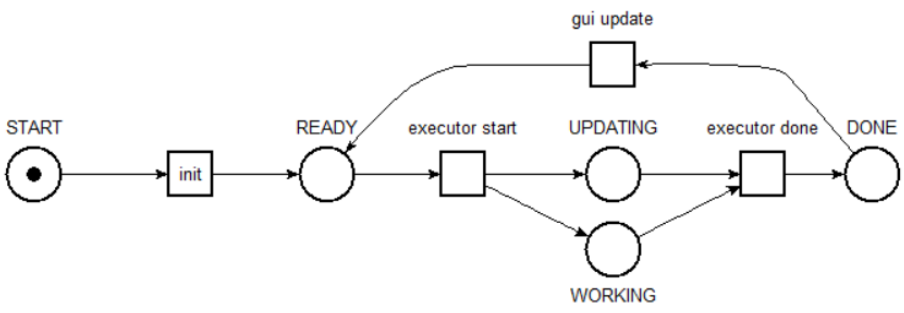

Figure 3: Brief Petri Net of the task-based approach

4.3 Virtual-Thread Approach

Identically to the task-based approach, having a fine-grained subdivision of work has simplified implementation and necessary structures for synchronization; needing just a work start and a work done synchronization structure.
Since virtual threads' lightweight framework allows for thread initialization and deletion with no major performances loss, synchronization was built using all threads completion as a signal of shift in workload computation phases'.

5 Performance

In this section I will present a study on performance of all three approaches.
Performances were calculated upon execution time with utmost care that each reading was exempt from any foreign interference.
This study will mainly focus on the simulation's pure computational aspect, without taking into consideration any improvement to the GUI attached.

5.1 Speedup

Speedup's formulation gives a clear index in performance improvement as more computational units are available for execution.
In particular speedup is used to highlight how as more CPU cores are available for computation, time of execution is shortened; indicating that for increasingly better results more cores are needed or to find a cost-manageable resource requirements for adequate performances.
Speedup is calculated as the time spent during serial execution divided by the amount of time spent during parallel execution with an increasing amount of dedicated cores; I want to note that for "serial execution" I've used parallel implementation using only one thread to get reliable results.
For simplicity reasons, speedup calculation was done only on the platform thread's approach, as precise CPU core's use was easier to manage and document.

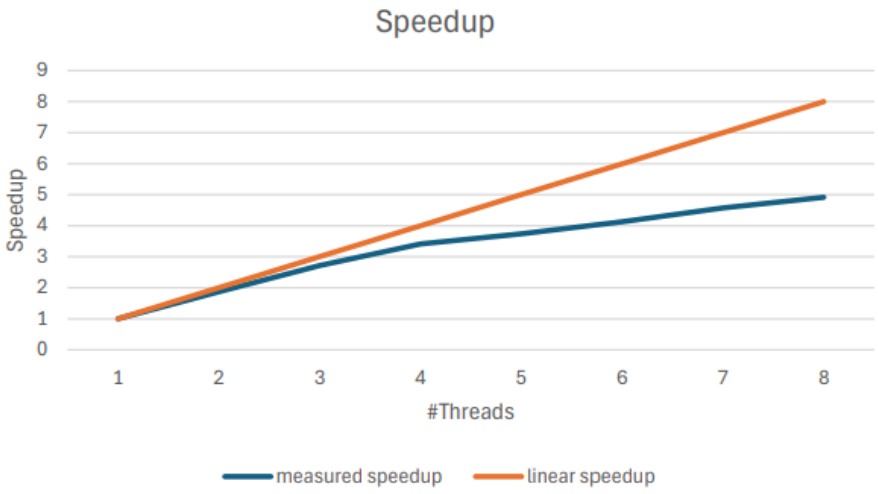

Figure 4: Speedup

In the Figure 4 I've computed and put on display the speedup of the platform's thread implementation. From the graph, we can denote that the implementation is less "core-efficient" that a linear speedup, this loss is caused by the need for threads to be synchronized in common points.
We can also notice a major "efficiency" loss after the fourth thread, this is caused by the physical architecture of the machine where these data were taken, as this was done with a SMT processor.
SMT, Simultaneous multithreading, briefly means that the cores listed by the machine are not all physical, some of them, in this case 4 threads, are simulated by the physical ones.
While I've not shown in the afore-mentioned figure, serial implementation offers better overall performances than the platform thread's using only one thread; this is to be expected as thread initialization, synchronization and deletion causes non-discreet performance losses.

5.2 Throughput

Throughput is the measurement of computational units done in a time frame.

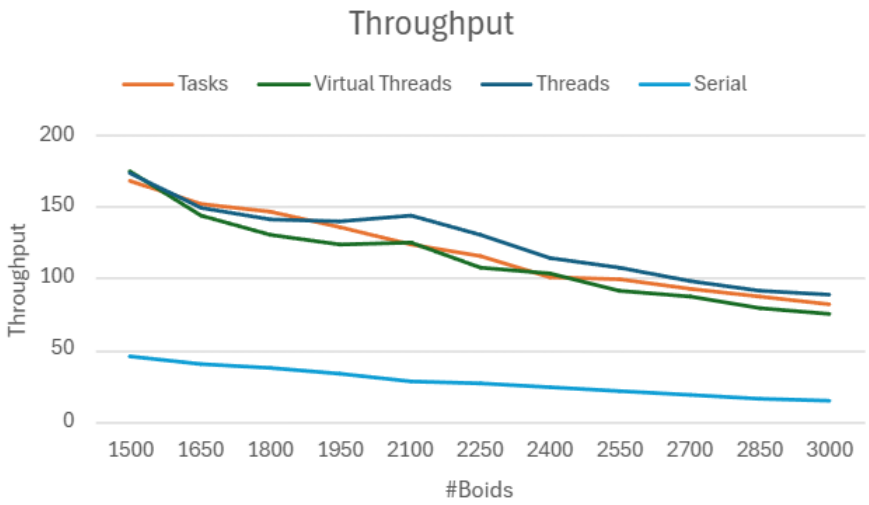

Figure 5: Speedup

In the Figure 5 shown above, I've highlighted the throughput of all implementation; parallelization has clearly impacted positively the number of computations done per time frame of execution.
We can also see a trend of decreasingly throughput as amount of compute work increases, this may be caused by the increasing need for synchronization and non-parallelizable portions of code.

6 Testing

Parallel computation introduces non-determinism, as such appropriate and through testing must be done to ensure no typical errors such as race conditions, deadlocks or starvation occur.
Testing was done using the JPF (Java Path Finder) framework, which removes non-determinism by testing each possible execution state; after a through search JPF found no error withing the code provided, guaranteeing a correct structure of code, without errors.
JPF's in-depth analysis can be easily ran off course when non-trivial computation generates multiple possible branches of execution; for this reason when checking the implementation using JPF I've specifically removed any unnecessary part, reducing the total amount of code to check.
To be more specific, all references to the GUI parts of the program were removed; the implementation was reduced to the core synchronization parts, with focus on correct execution without any race condition or deadlock.

```
============================================ results
no errors detected
============================================ statistics
elapsed time:       00:00:50
states:             new=169341,visited=296750,
                    backtracked=466091,end=0
search:             maxDepth=827,constraints=0
heap:               new=988674,released=987812,
                    maxLive=662,gcCycles=418717
instructions:       53660793
max memory:         222MB
loaded code:        classes=105,methods=2586
============================================ search finished
```

Listing 1: JPF's output

Upon JPF checks' conclusion, the code is assured to be thread safe and without race conditions and deadlocks.

7 Simulation Execution

In addition to this report, a zip file will be shared containing the source code this report refers to. In order to test such code, different folders containing each approach's source code will be provided.