Programmazione Concorrente e Distribuita - Assigment 01

Leonardo Randacio
Filippo Gurioli
Andrea Biagini

Università di Bologna
Scienze e Ingegneria Informatiche

April 8, 2024

Contents

List of Figures

List of Tables

Chapter 1

Analysis

The goal is to create a concurrent agent-based simulation.

An agent-based simulation or model is a computational modeling technique used to simulate complex systems by representing individual entities, known as agents, and their interactions within an environment. The goal of the simulation is to observe the evolution of the states of the environment and the agents in each discrete step.

Agents beheviour for a single step can be described in 3 phases:

- sense phase: the agent acquires data from the environment
- decide phase: the agent determines the next action
- act phase: the action determined is executed on the environment

1.1 Task Decomposition

Each agent's step can compose a single task, which can be divided into 3 subtasks, one for each phase of the step.

This means that for a given step there will be 3 tasks:

- sense
- decide
- act

The total number of tasks for a given step is 3 * nAgents where nAgents is the number of agents.

1.2 Data Decomposition

The environment can be subdevided in agent's states which means the data can be divided in nAgents indipendent states.

1.3 Dependency Analysis

The sense and decide tasks can be joined in a single sense-decide task as the sense task only quearies the environment and the decide task updates the next action to be performed for a given agent. Since the decide phase only sets the next action for a given agent and for a given step every agent's next action will be set only by one decide task, the sense-decide tasks can be executed in a concurrent manner.

The act tasks in a given step can be parallelised between each other, but must be executed after the sense-decide task.

The steps of the simulation must be serialized.

Chapter 2

Design

Using agenda parallelism we design the system around an agenda composed by the various tasks, which were defined earlier. Every step in the simulation imposes a mandatory syncronization.

2.1 Architecture

The master-worker architecture is a natural implementation of agenda parallelism, using a bag of works. The bag of works is implemented as a bounded buffer.

The environment is implemented as a readers-writers monitor, which lets threads read from the environment in a parallel manner, but imposes that no writing can be executed concurrently with other writing or reading.

2 Latches are used for syncronization between master and workers.

2.2 Visual Formalisms

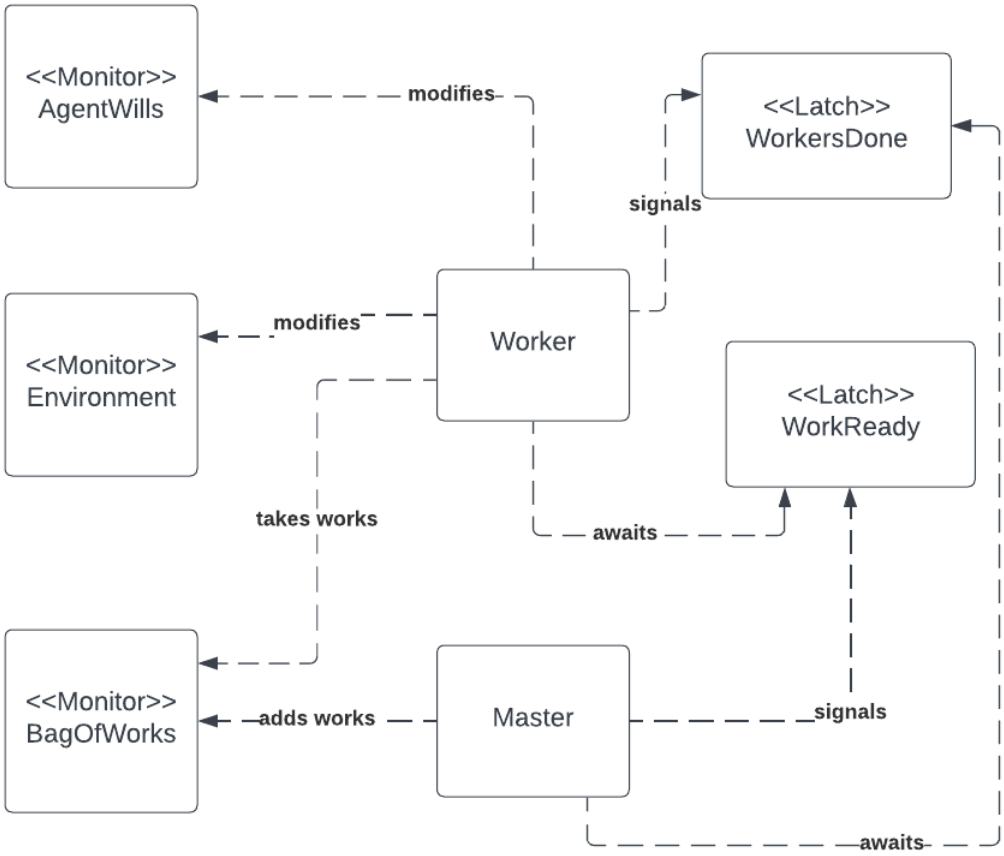

Figure 2.1: UML diagram showing relationships between monitors, active components and syncronization components

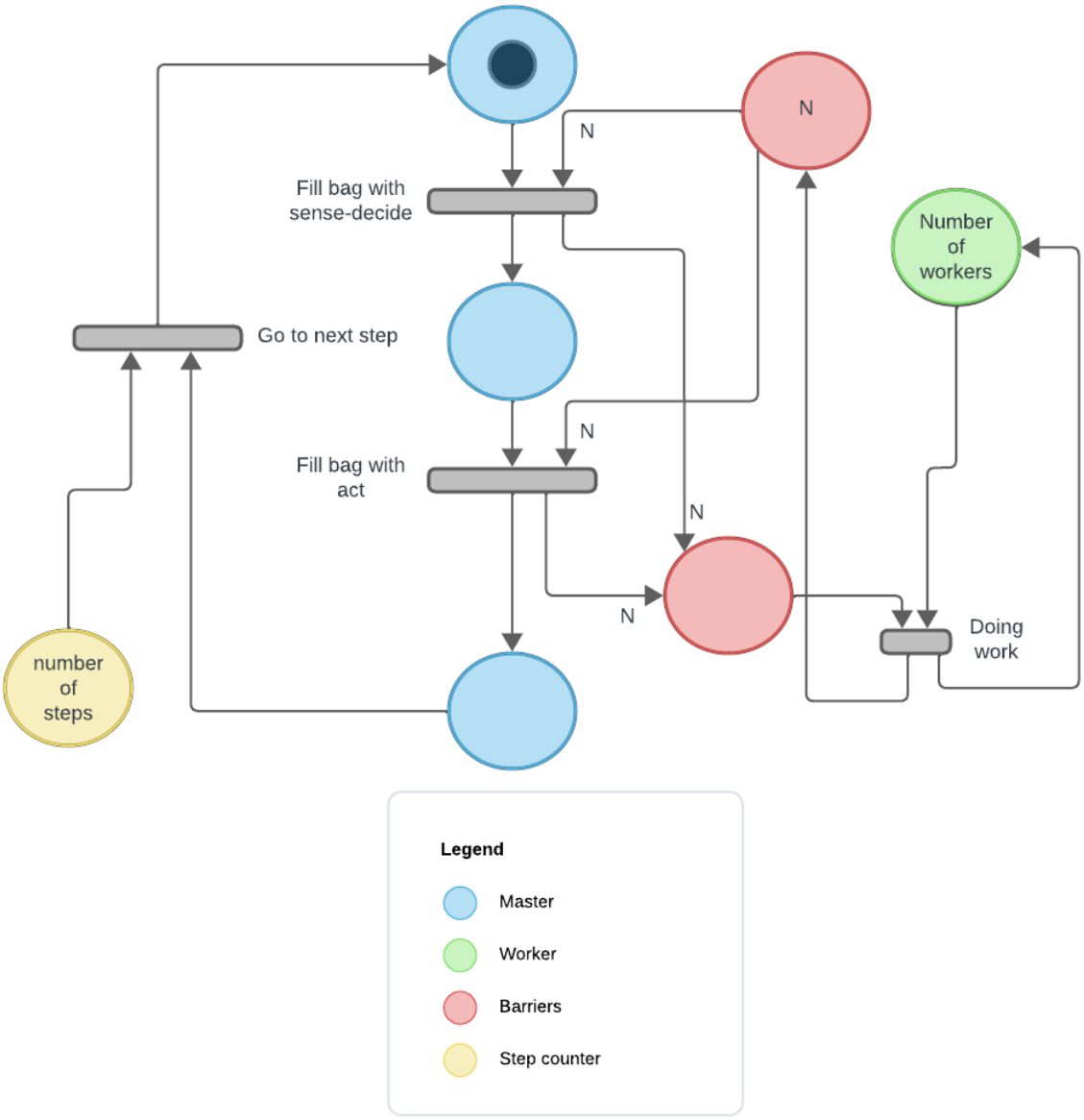

Figure 2.2: Petri net diagram showing the beheviour of the master and workers threads. N is the number of agents in the simulation. The master beheviour is rappresented by 4 places and 4 transitions. The worker is rappresented by one place and one transition. A place containing a token for every step of the simulation regulates the number of steps. Two places rappresent the number of tasks in the bag and the number of tasks finished, regulate the syncronization between master and workers

Chapter 3

Implementation

3.1 Deadlock Problems Encountered

During implementation some deadlock problems arised.

3.1.1 Worker Initialization

A certain state of the system caused some workers to be left behind. After being initialized a worker could start waiting on the `workReady` latch after the master started a step. This caused a deadlock as the worker waited indefinitely for the master to signal that the bag was full and the master waited indefinitely for all the workers to signal they where ready to recieve other work.

This was solved by having the master wait on the `workersReady` latch after initializing the workers. This way, when the first step of the simulation started, all workers where ready to start, making sure no worker was left behind.

3.1.2 Check and Act

A certain state of the system caused two workers to access the bag of tasks even though only one work remained in it. This meant that one of the workers would complete the task while the other one would await on the notEmpty condition of the bag of tasks. Since the master waits for all the workers to notify that they are ready for the next tasks, it would never fill the bag with new tasks, waiting indefenetly for the awaiting worker to notify it was ready to continue. This is a classic check and act problem.

This was solved by making the buffer return an `Optional` istead of a `Runnable`. The worker now receives an empty `Optional` from the bag if it

was empty, instead of awaiting for the bag to be filled again. The notEmpty condition was removed from the bag of works monitor.

3.1.3 True Syncronization

The fact that a worker executed the `countDownLatch()` on the workers ready latch and the `await()` on the work ready latch in a non atomic way, the master could execute the `countDownLatch()` on the work ready latch, signaling to the workers that the bag was full of new works, before all the workers where awaiting on it. This would make it so that the master would wait for the workers to signal that they had executed all the works available but some workers would wait for the master to signal that the bag was full.

This was solved by eliminating the `workReady` latch, exchanging it for a condition of the bag of tasks: notEmpty. This means that workers no longer wait on a latch but wait directly on the notEmpty condition inside the bag of tasks. Before blocking a worker the bag of task executes the `countDownLatch()` on the workers ready latch, telling the master that a worker has paused. The master is then signaled that all workers are paused through the workers ready latch and as soon as a new task is added to the bag by the master a worker is woken to resume it's working loop.

3.2 Components

The master and worker threads are the active components. The bag of tasks, the environment, the agent wills and the workers ready latch are all passive components, implemented as monitors

3.3 Source Code Organization

The basic simulation implementation has been extended with the GUI implementation

The GUI source code is comprised of only one class: simtrafficexamplesconcurrent/RoadSimView.java

The class simtrafficexamplesconcurrent/RoadSimView.java prints information of the simulation on the terminal

The src is organized in the following packages:

- masterworker: containing classes for the master worker pattern implementation

- simengineconcurrent: abstract classes for the simulation framework
- simtrafficbaseconcurrent: concrete classes for the traffic simulation
- simtrafficexamplesconcurrent: concrete simulation and main classes given as examples

3.4 Thread Organization

The master is implemented as a single thread. The Gui will have a dedicated thread managing its events. Depending on the machine architecture, the remaining number of instantiable threads are created and used as workers.

Chapter 4

Execution and Evaluation

The main classes for the part1 and part2 requirements are contained in the `simtrafficexamplesconcurrent/part2` package.

- `RunTrafficSimulation.java`: some simple examples of traffic simulations
- `RunMassiveTrafficSimulation.java`: a simulation with a massive number of cars

The main classes for the part3 requirements are contained in the `simtrafficexamplesconcurrent/part3` package

- `RunRandomTrafficSimulation.java`: simple example that introduces randomness

4.1 Number of Workers

The number of workers is calculated based on the number of cores the executing environment has with the following formula:

$$min(max(cores - 2, 1), nAgents) * coef$$

where *cores* is the number of cores of the executing environment, *nAgents* is the number of agents in the simulation, *coef* is a coefficient euristically determined to be equal to 4.

We believe that if every java thread coencided with an actual core being used (low level thread) the coefficient would not be necessary.

4.2 Sequential and Concurrent Comparison

Testing the execution speed of the concurrent and sequential versions of the simulation with a massive number of cars we obtained the following data:

- concurrent execution time: 25.8 seconds on average
- sequential execution time: 30.4 seconds on average

4.3 JPF

The use of *Java Path Finder* was tried but after converting all the codebase into Java 8, solving uncountable issues with gradle we still detected compilation errors. Thorough testing has been executed to ensure no deadlock/livelock is possible.

Chapter 5

Conclusions

As stated previously, execution time improvements where mesured. The concurrent version is consequently more scalable, as the number of workers is computer based on the number fo cores available. The difference in the execution times detected between the two kind of `RunMassiveTrafficSimulation` seems too little. We came up with the idea that the bottleneck on our application is the large use of data structures. All workers do not have any state inside, leading to the necessity to store the state of every agent on a seperate data structure (AgentWills) to be retrieved at a further time.