The Gravitational N-Body Problem - Assignment Report

Roberto Casadei

Alma Mater Studiorum – University of Bologna
via Venezia 52, 47023 Cesena, Italy
roberto.casadei12@studio.unibo.it

1 Introduction

This document represents the report for the assignment issued within the course *Programmazione Avanzata e Paradigmi* (academic year 2012-2013), which is aimed at verifying concurrent programming skills.

The chosen problem is the Gravitational N-Body Problem, a GUI-based simulator for the evolution of a system of bodies according to the Netwon's laws in classical mechanics.

2 Problem analysis

2.1 Workflow

1. First, we develop a sequential solution
2. Then, we develop a concurrent solution
3. Finally, we look for "sustainable" optimizations

This is done in order to assess the application logic – which must be the same in all cases – and to avoid premature optimization.

2.2 Analysis – business logic

Here, we analyze the requirements and build a model for the business domain. The concepts/entities that emerge from the problem are:

- *body*: this term is counterintuitive as it actually refers to a "point particle" with a certain state
- *body state*: is characterized by
  - *position*: identify a point in the bidimensional space
  - *velocity* or instantaneous speed
  - (instantaneous) *acceleration*, which can be changed by a *force*

In the Cartesian coordinate system, these concepts can be represented as vectors. Moreover, their structure and (dynamic) properties derive from physics and, in particular, from the laws of classical mechanics.

Every DT milliseconds, the next state of the system (i.e. the galaxy) must be calculated. The global state of the system is determined by all the bodies' state.

It turns out that there is one main constraint in the system: **in order to be able to (completely) calculate and set the system's (i+1)-th state, the i-th state of all the bodies must available**.

2.3 Synchronization – temporal constraints

A body can be seen as an entity that constantly moves from a state to the next. In this process, two main steps can be considered:

1. Computation of the net force the body is subject to

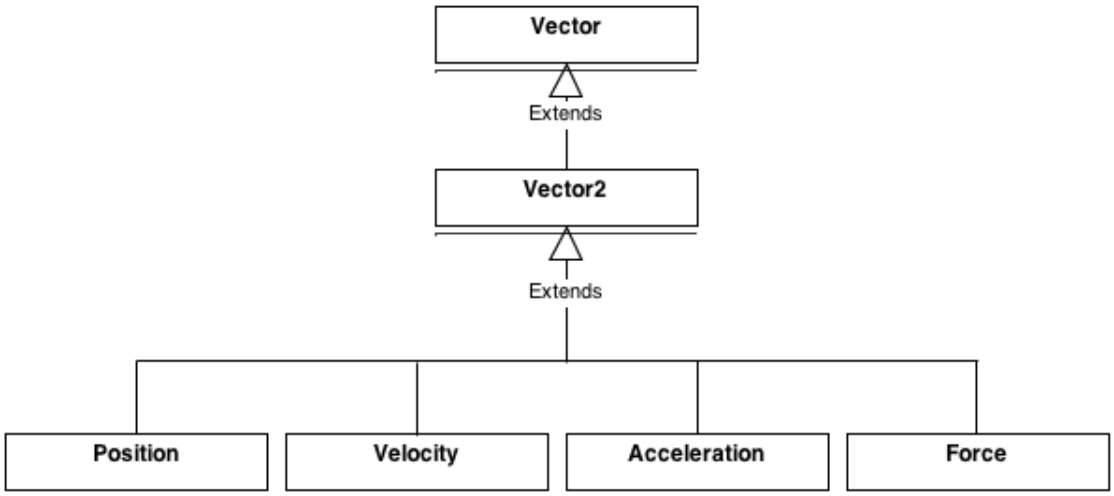

Fig. 1. Domain model

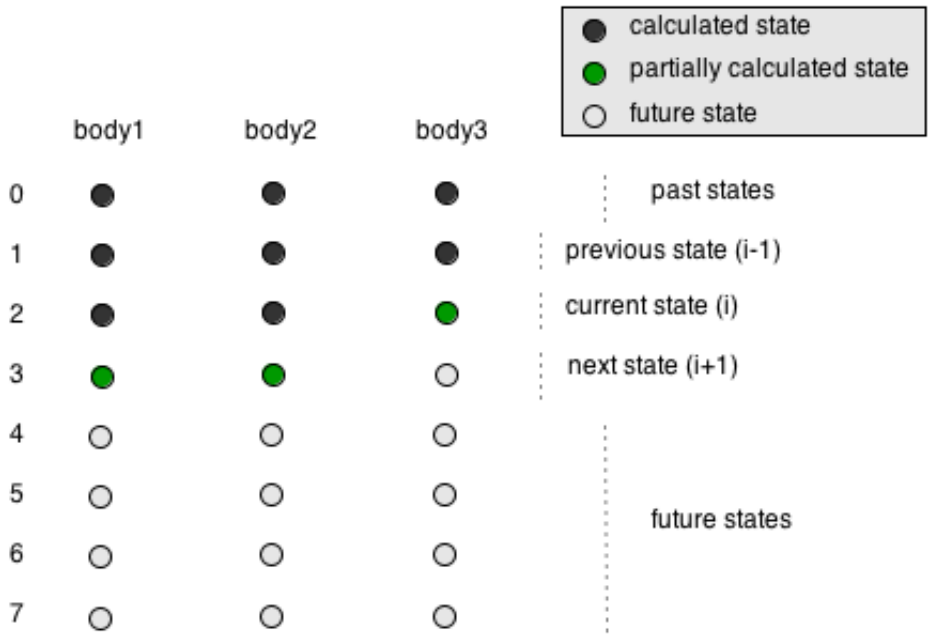

Fig. 2. Bodies and states

2. Update of the body's position, velocity, and acceleration based on the net force and current state (state transition)

In particular, the net force at state i is responsible for the acceleration of the body at state *(i+1)*.

For what concerns **step 1**, there is the case for a body to need to wait for another body. For example, we see in Figure 2 that bodies 1 and 2 need to wait for body 3 to fully calculate its state.

For what concerns **step 2**, a question arise: *when is a body allowed to proceed to update its state?* The risk is that a body moves to state i while other bodies need to know the position of that body at state *(i-1)*. Two options can be considered :

1. a copy of the bodies' current state is created and made available, and the bodies can immediately proceed to update their state
2. a body can proceed to set its next state only after all the other bodies have accessed its current state

A simpler approach (that however introduces some serialization) consists of waiting for all the bodies to complete step 1, then waiting for all the bodies to complete step 2, and doing so recursively. Moreover, this could be easily implemented using barriers.

2.4 Analysis – task decomposition

At this point, we need to identify the tasks (logical work units) and their dependencies.

Multiple decompositions are possible, for example:

Body's behavior as task. A natural decomposition is to have a 1-to-1 mapping between bodies' behavior (lifecycle) and tasks. Thus, bodies may become *active objects* (see Figure 3) or actors.

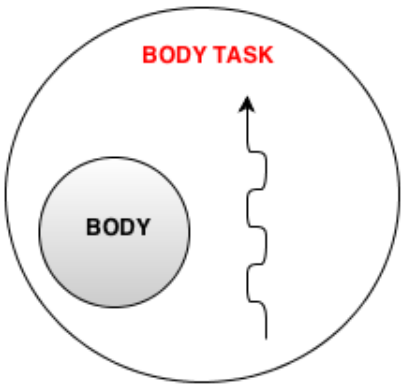

Fig. 3. A body-task can be seen as an augmentation of the body-object (which encapsulates core business logic).

These tasks are subject to the same synchronization constraints outlined in the Section 2.3.

Main computations as tasks (chosen). Two main actions need to be performed by a body in order to move to the next state:

1. calculation of the net force the body is subject to

2. update of the body state (based on the net force and current state)

Moreover, the task 1 might be considered as the summa of the single contributes to the net force (partial calculations), which in turn can be considered as finer-grained tasks.

These tasks must be synchronized with the state of the system.

3 Design & Implementation

3.1 Program architecture

The application is logically (and almost physically) subdivided into 5 modules:

1. interfaces
2. core business logic (bodies and physics)
3. graphical user interface
4. concurrency (tasks, synchronization artifacts, executors)
5. testing and execution

The *Model-View-Controller (MVC)* and *Observer* pattern have been followed to solve the recurring issues around the interaction between the GUI (i.e. the view) and the system of bodies (i.e. the model).

3.2 Concurrent architecture

The concurrent architecture may be seen to recall a sort of *result parallelism*. In fact, at each step, there is one concurrent activity for each body in the system.

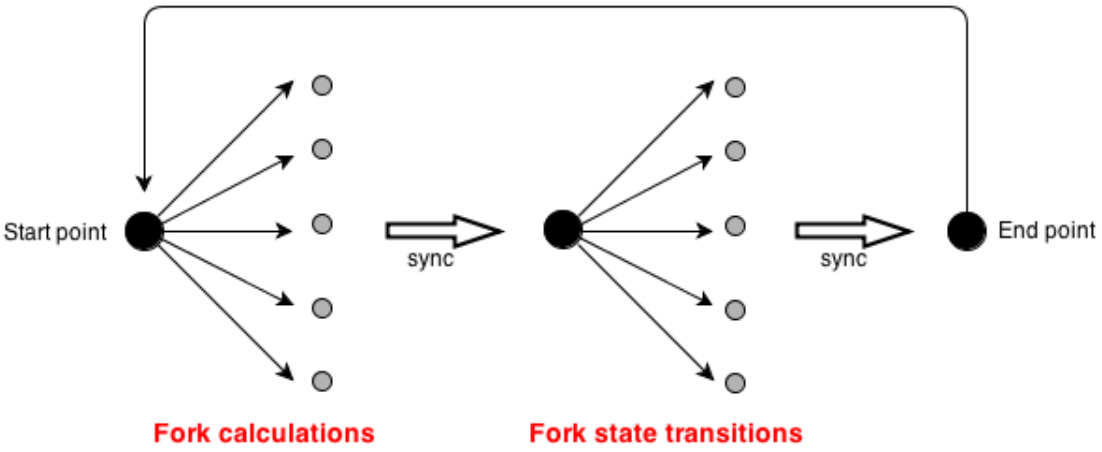

Fig. 4. This is what is recursively done in order to bring the system from a state to the next.

The implementation described in Figure 4 is clean but introduces some serialization. Such serialization is introduced in order to avoid two incorrect behaviors:

1. a body calculating the net force based on a “future” state of another body
2. a body calculating the net force based on a “past” state of another body

3.3 Active and passive components

A high-level overview of the system is represented by Figure 5.

The following components are significant and deserve some description (for more about their behavior and interaction, look at the code).

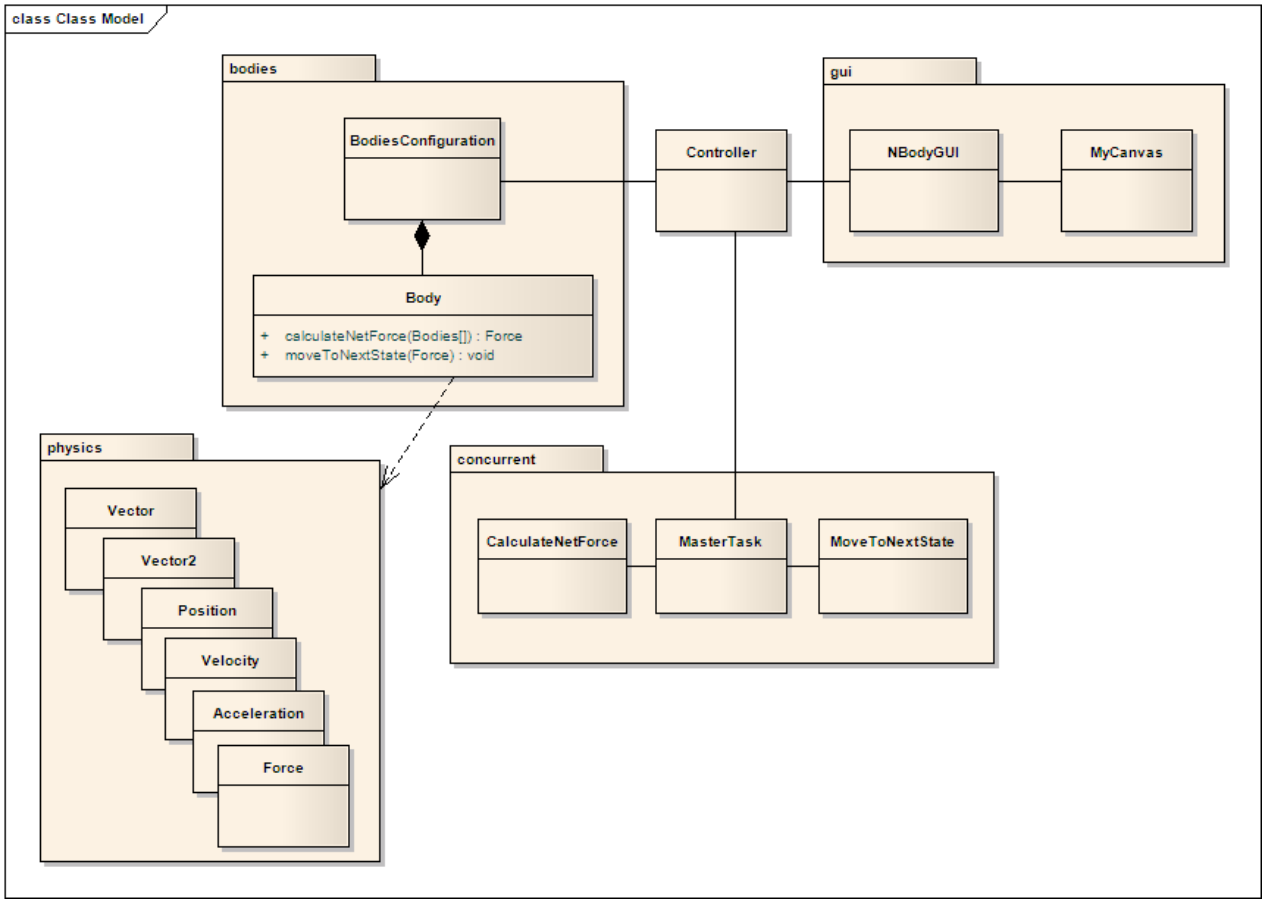

Fig. 5. A high-level overview of the design of the system.

it.unibo.pap.nbodies

- *Body* – represents a body (or, better, a particle), which is defined in terms of its position, velocity, and acceleration. It provides two main operations: one for calculating its associated net force, and one to bring the body to the next state based on a given force.
- *BodiesConfiguration* – represents a system of bodies (a sort of galaxy); it's a sort of container which also keep trace of common properties for the included bodies.
- *KB* – collects static, system-wide properties (such as the delta time value, the factor form value for the canvas, and various constants).
- *Utils* – collects utility methods and, in particular, those for generating systems of bodies (e.g. randomly or from CSV files).

it.unibo.pap.nbodies.gui

- *NBodyGUI* – is the graphical user interface; it contains buttons for commands, a canvas wherein the simulation is painted, and a status bar. Moreover, it provides some keyboard shortcuts for altering the display and the speed of the simulation.
- *SimulationCanvas* – it's where the simulation (i.e. for each state, a representation of the bodies in their relative positions) is painted.
- *Controller* – is the glue component between the GUI and the bodies. It handles the GUI events that are related to the management of the simulation (creation of a system of bodies, start of the galaxy, stop of the galaxy) and notifies its attached listeners (as the GUI itself) which desire to be informed in case of changes at the model (such as when the system of bodies reaches a new state).

- *Flag* and *Flags* – the former represents a boolean flag, while the latter provides a convenient way to pass around a collection of flags. They are useful, for example, for stopping the simulation.

it.unibo.pap.concurrent.tasks

- *MasterTask* – this task is responsible for the evolution of the galaxy by recursively executing (forking), for each body, the *CalculateNetForce* and *MoveToNextState* tasks. It also notifies the *Controller* for the system's state updates.
- *CalculateNetForce* – this CPU-bound task contains the logic for calculating the net force associated to a body (given the configuration of bodies); it can be safely run concurrently with other bodies' respective calculation task.
- *MoveToNextState* – this task is responsible for making a body transit to the next state based on the calculated net force.

4 Testing

4.1 Correctness

At a first step, correctness is verified by executing the simulation – this can provide some hints about a plausible evolution of the galaxy in terms of the effects due to the forces.

At a second step we verify that, at the moment in which the observers of the galaxy are notified for a new state, all the bodies in the galaxy have the *same state count* (see `it.unibo.pap.nbodies.test.CorrectnessTest`).

4.2 Performance

Performance optimization. The following optimizations have been implemented:

- Use of *mutable objects* for vector quantities (such as position, velocity, ...) in order to limit the proliferation of objects.
- Implementations (for algorithms and classes) that avoid superfluous creation of objects or expensive usage of collections.

Other possible optimizations which may be implemented are the following ones:

- Calculating the *center of mass* once in order to reduce the calculation burden for each particle. Thus, each particle can calculate the net force by subtracting its contribute to the center of mass and using that result as an equivalent second body.
- Removing the serialization between the two galaxy steps (calculation of the net force for all the bodies, and state update), allowing the bodies to proceed like represented in Figure 2. It has not been implemented in order to keep the system simple. Moreover, it would introduce some serialization for what concerns the access to the other bodies' state.

Performance results.

Performance is measured by *the number of states that the system is able to compute in a given time window* (throughput). See `it.unibo.pap.nbodies.test.PerformanceTest`.

From the data reported in Figure 6 and Figure 7, a number of observations about the implementation that has been built and about the nature of the solution can be made:

Num bodies	Num of States Computed per second (mean in 10 samples)						
	Sequential	Actors	Concurrent				
			N=1	N=2	N=3	N=5	N=10
5	43884	11286	39495	19076	33491	17039	7606
50	390	263	391	668	555	550	416
200	25	19	25	39	40	38	41
500	4,3	3,2	4,5	7,5	7,3	7,4	7,6
1000	1,2	1,1	1,2	1,9	2,1	1,9	2,1
Tests on:	Intel® Core™2 Duo CPU T6400 @ 2.00GHz×2						
	Memory: 4 GiB						
	OS: Linux Debian 3.2.35						

Fig. 6. Performance results (1).

Num bodies	Num of States Computed per second (mean in 10 samples)						
	Sequential	Actors	Concurrent				
			N=1	N=2	N=3	N=5	N=10
5	109497	31511	100774	43830	61025	9547	8196
50	866	716	861	1686	2162	2472	519
200	52	51	52	467	134	169	153
500	9,5	8	9,5	17,5	22,1	120	26,6
1000	2,5	2,1	2,5	4,7	6,2	7,5	7,5
Tests on:	Intel(R) Core(TM)2 Quad CPU Q8300 @ 2.50GHZ						
	Memory: 4 GiB						
	OS: Windows 7 Home Premium 64-bit						

Fig. 7. Performance results (2).

- Performance degrades quickly with the increase of the number of bodies – this is coherent with the complexity $O(n^2)$ of the problem
- Using many threads with few bodies results in lesser efficiency; in particular, we see that with only 5 bodies, the most efficient solution is the sequential one
- As the tasks are CPU-bound, we see that (except in the cases with few bodies) the best performance are (generally) met when there are $N_{CPU} + 1$ physical threads in use

Profiling results.

The application in execution has been monitored using VisualVM in the following system:

- Processor: Intel Core 2 Duo CPU T6400 @ 2GHzx2
- Memory: 4GiB DDR3
- OS: Linux Debian 3.2.35

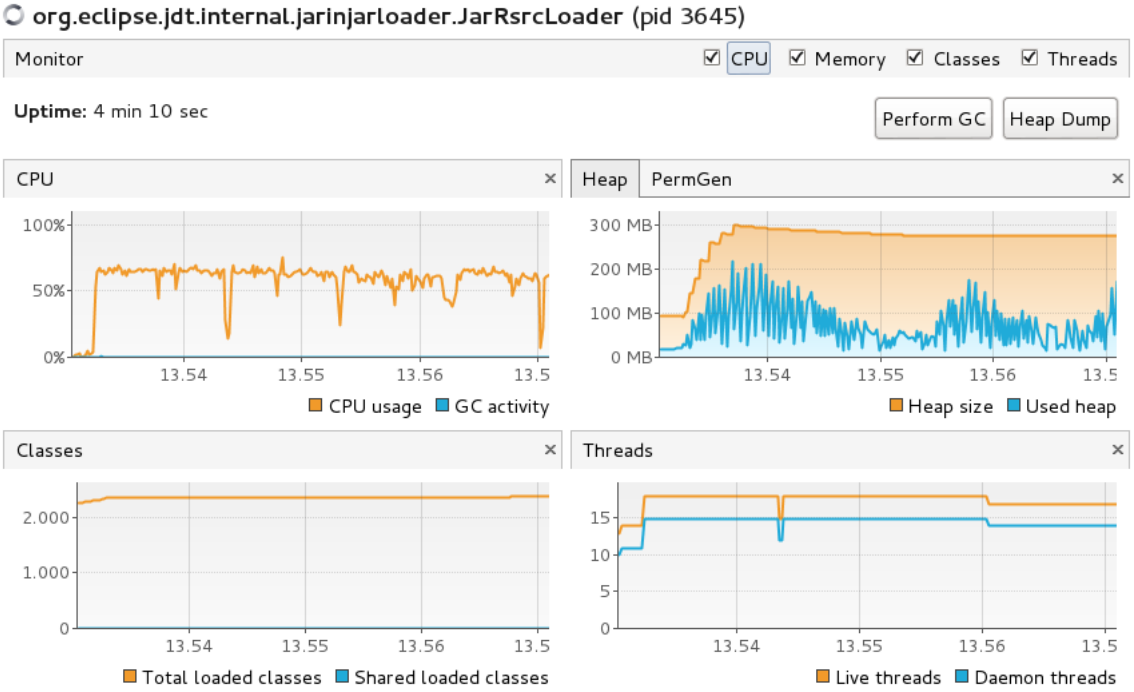

Fig. 8. Monitor.

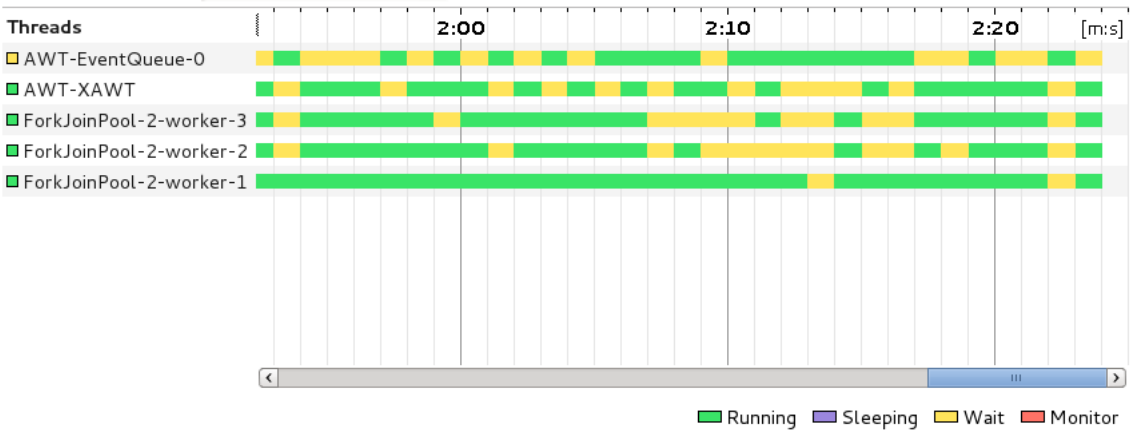

Fig. 9. Thread visualization.

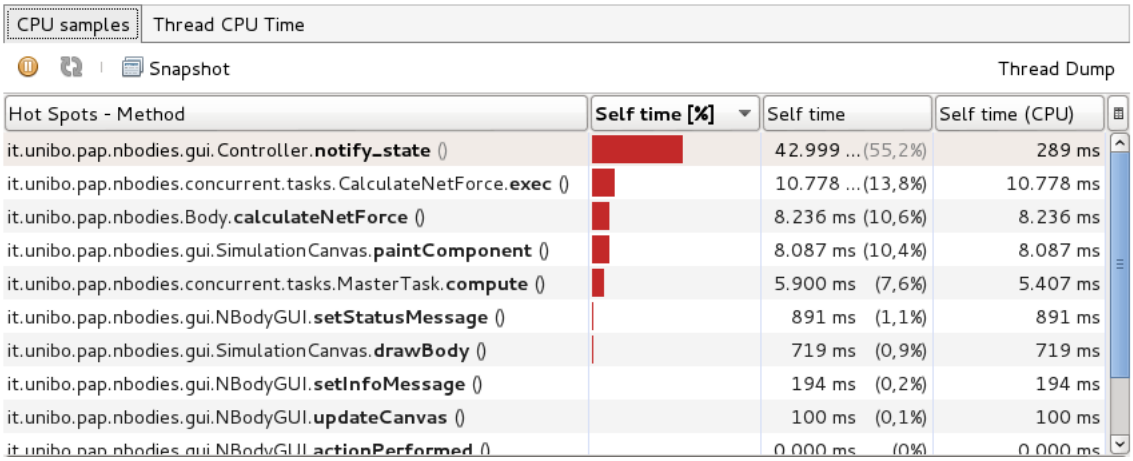

Fig. 10. CPU samples.

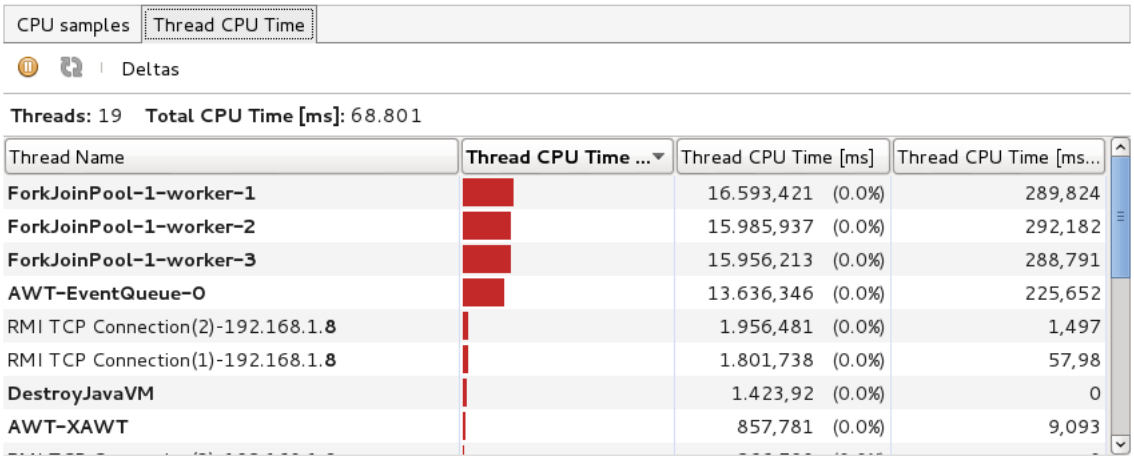

Fig. 11. Thread CPU time.

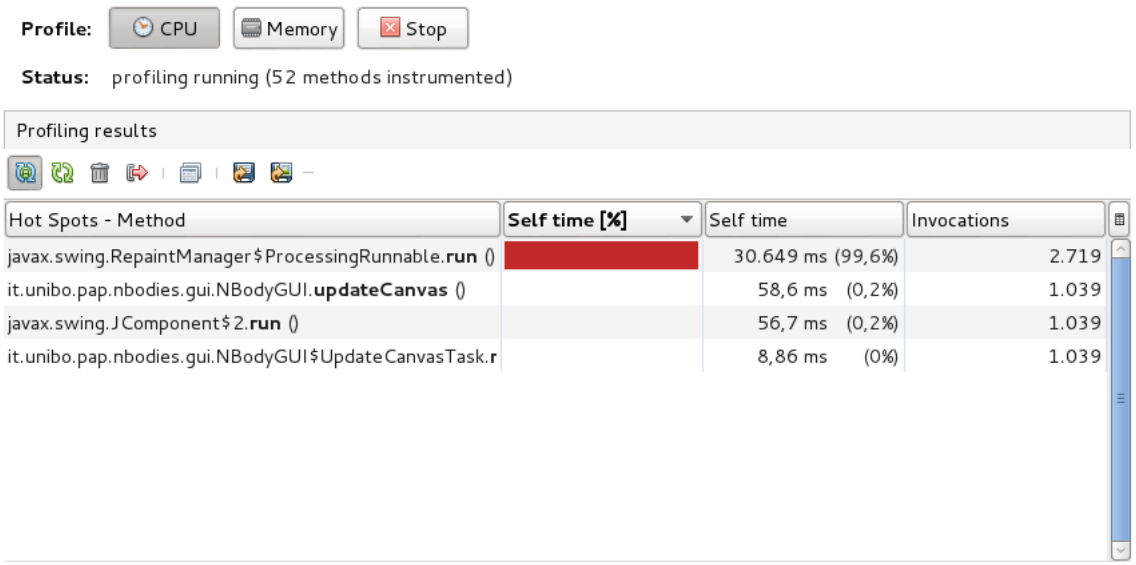

Fig. 12. CPU profile.

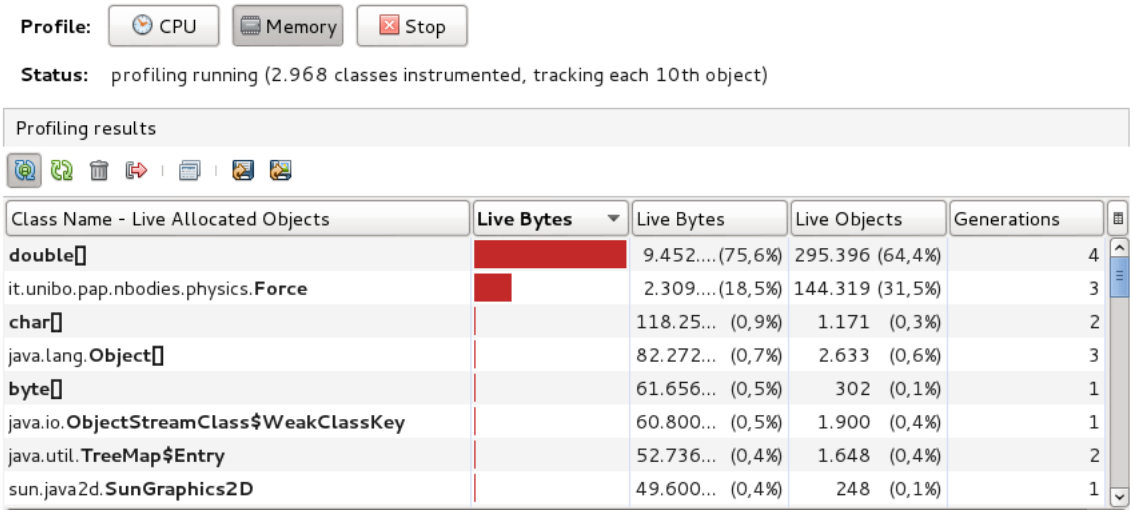

Fig. 13. Memory profile.

Let's make a few comments about these data:

- As Figure 10 (CPU Samples) shows, the *methods in which most time is spent* are `Controller.notify_state()` (because it waits for the canvas to repaint the galaxy) and `Body.calculateNetForce()` (because it is the major business logic-related calculation – and is performed for all the bodies, which number can be significant in many cases).
- As Figure 12 (CPU Profile) shows – but it is not surprising – a great bottleneck is given by the repainting of the canvas.
- Figure 9 (Thread Visualization) and Figure 11 (Thread CPU Time) show the (physical) threads of the application: the GUI event loop and the $N_{CPU}+1$ worker threads which are used to run the tasks.
- Figure 13 (Memory Profile) gives a sense of memory usage in terms of the objects that are created. We see, for example, that the `Force` objects result in a significant burden to the memory (as they are continuously created), while `Position`, `Velocity`, `Acceleration` are not at the top of the list (because they are used in a mutable-manner).