Programmazione Concorrente e Distribuita

Assignment 1

Angelo Parrinello, Paolo Penazzi

April 9, 2022

1 Analisi del problema

Si deve creare un programma in grado di simulare il comportamento di N corpi all'interno di un quadrato. Inizalmente i corpi vengono creati fermi in posizione random all'interno del quadrato. A questo punto si calcolano le forze che muovono ciascun corpo, aggiornando ad ogni iterazione le posizioni di ciascuno. Se un corpo sta uscendo dal quadrato, la sua direzione viene invertita in modo che rimanga all'interno. Il programma, per come viene fornito inizialmente, è interamente sequenziale. Per aumentare le performance della simulazione è evidente la necessità di parallelizzare il flusso del programma. Per arrivare a tale risultato si è deciso di decomporre il problema in task:

- **Calcolo della nuova velocità:** la nuova velocità di un corpo B viene calcolata applicando alla velocità attuale due forze: la forza esercitata dagli altri N - 1 corpi (forza repulsiva) e dalla forza di attrito.
- **Aggiornamento delle posizioni:** dato un corpo B la sua nuova posizione viene calcolata sulla base della velocità del corpo.
- **Controllo sui bordi del campo:** Se un corpo B sta uscendo dal campo, si inverte la direzione della sua velocità.

Una volta identificati i task, si è passati all'analisi di eventuali dipendenze tra essi. Dal punto di vista di un singolo corpo, tra i task c'è una dipendenza temporale chiara: devono essere eseguiti nell'ordine in cui sono stati descritti (prima si calcola la velocità, POI si aggiorna la posizione, POI si controllano le collisioni con i bordi). Dal punto di vista della simulazione i primi due task devono essere sincronizzati: per calcolare le nuove velocità dei corpi al tempo T è necessario che le posizioni siano aggiornate al tempo T - 1, e che non vengano modificate durante il calcolo della velocità. Questa dipendenza è data dal fatto che per calcolare la forza repulsiva che agisce su un corpo, devo conoscere le posizioni di tutti gli altri corpi. Nell'immagine seguente è illustrato il comportamento che deve avere il programma:

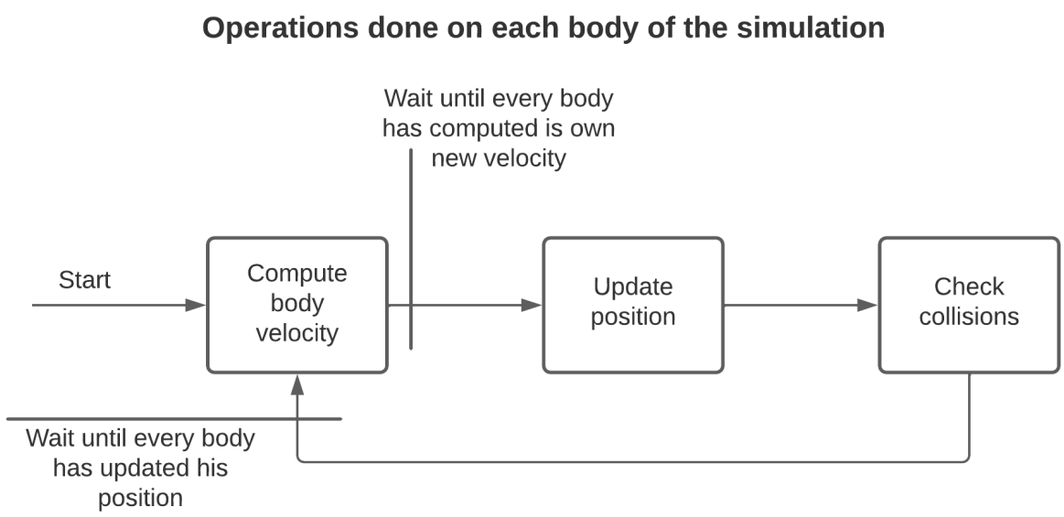

Figure 1: Simulation flow

Analizzate le dipendenze, è necessario capire quali task effettivamente rallentano la simulazione. Il seguente graifoc è stato creato misurando i tempi di esecuzione dei singoli task. Come si può notare il calcolo delle nuove velocità è il task più oneroso (Attenzione: l'asse y del grafico è esponenziale.)

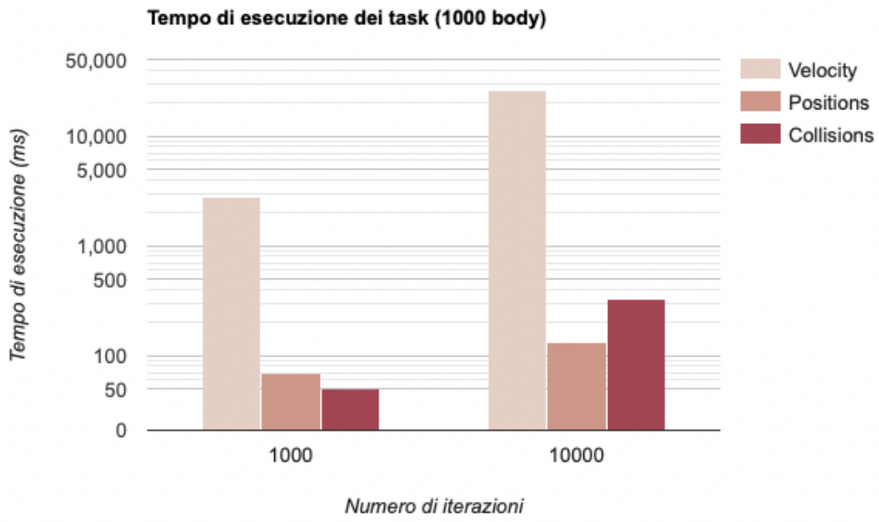

Figure 2: Confronto fra i tempi di esecuzione dei task.

Il tempo di esecuzione del calcolo delle velocità è stato poi confrontato con il tempo totale di esecuzione del programma. Dall'analisi risulta che circa il 60% del tempo della simulazione è utilizzato per calcolare le velocità dei corpi.

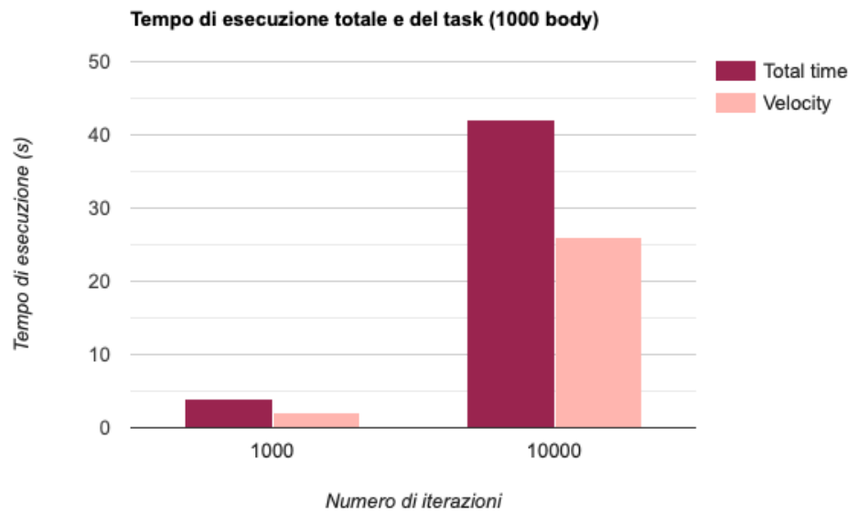

Figure 3: Confronto fra il tempo totale di esecuzione e il calcolo delle velocità.

Si procede quindi a modificare il design del programma in modo da velocizzare la computazione.

2 Design della soluzione

Dato che il nostro risultato è partizionabile, è stato adottato un parallelismo a risultato. Questo tipo di parallelismo infatti prevede che ad ogni step il lavoro venga suddiviso e assegnato a più worker: applicandolo al problema in analisi permette di suddividere il carico di lavoro tra i thread, i quali eseguiranno in maniera concorrente i tre task sopra citati. Essendo quindi un problema di order-of-execution, l'architettura concorrente più adatta risulta essere quella master-worker. L'idea è quella di un thread, detto master, che coordina un insieme di altri thread, detti worker.

Per coordinare il lavoro di master e worker, sono stati usati due costrutti noti in letteratura: la Barriera e il Latch.
Una barriera è un meccanismo di sincronizzazione che permette ad un certo numero di thread, di "aspettarsi" in un determinato punto. I vari thread si mettono quindi in wait sulla barriera fino a che non arriva anche l'ultimo thread che sblocca la barriera e permette a tutti di ripartire. Il numero dei thread da aspettare va dichiarato al momento della creazione della barriera. Un latch è un meccanismo di sincronizzazione che permette ad uno o più thread di attendere, la terminazione di n operazioni. Un thread si mette quindi in wait sul latch e ci rimane finchè non vengono eseguite tutte le n operazioni. La differenza tra barriera e latch è che la barriera richiede n thread per sbloccarsi, mentre per sbloccare il latch basta che venga chiamato n volte il metodo *countDown()* (Queste chiamate possono essere effettuate anche dallo stesso thread).

Dato che il tempo di computazione sarà circa lo stesso per ogni corpo, il carico di lavoro può essere gestito nel seguente modo: il master divide gli N corpi in equal numero tra i worker, in modo che tutti i worker ci mettano circa lo stesso tempo a portare a termine la computazione.

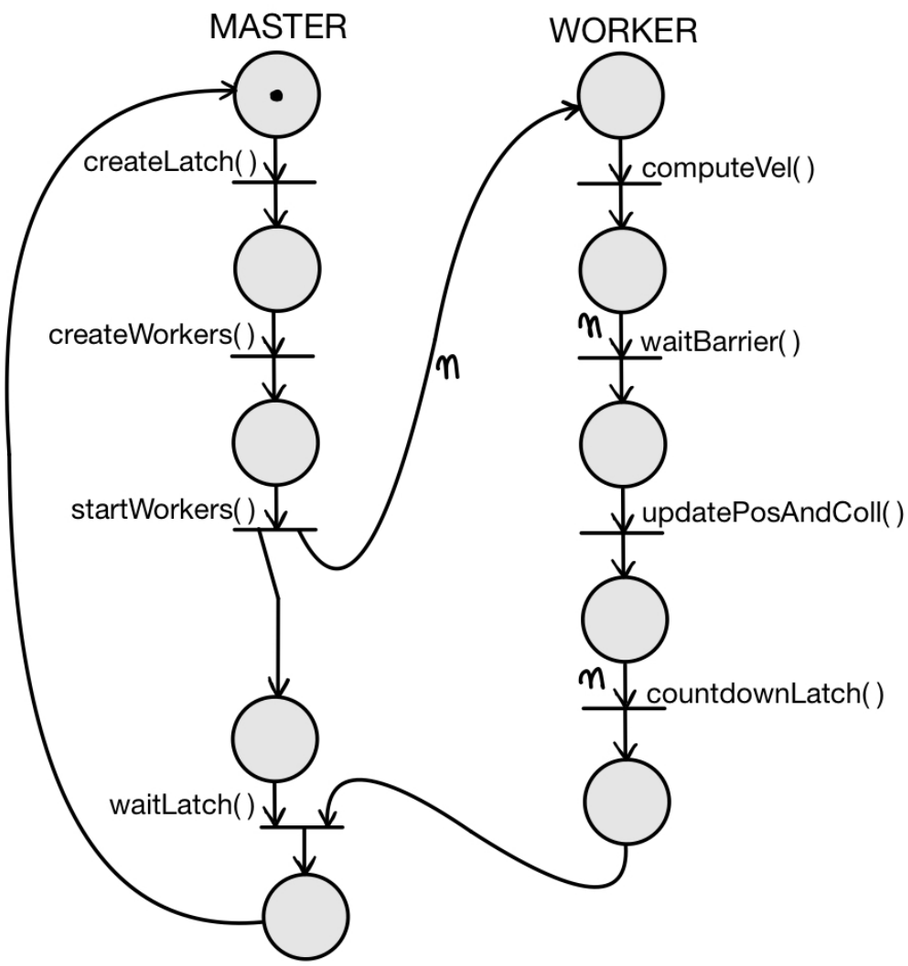

Figure 4: Rete di petri.

Il master si occupa quindi di creare e avviare i worker, assegnandogli un certo numero di corpi tra gli N totali. Fatti partire i worker, si mette in attesa che essi finiscano e una volta che tutti hanno terminato incrementa il numero dell'iterazione. Queste operazioni vengono eseguite un numero di volte specificato al momento dell'avvio della simulazione. Per attendere che i worker abbiamo terminato il loro task, viene utilizzato un latch, in particolare un CountDownLatch (dalla libreria *java.util.Concurrent*)

```
@Override
public void run() {
    long iteration = 0;
    while (iteration < super.getNumSteps()) {
        super.createLatch();
        super.createWorkers();
```

```
        for (Worker worker : super.getWorkers()) {
            worker.start();
        }
        try {
            super.getLatch().await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        iteration++;
    }
}
```

Listing 1: Master class

Il worker, quando viene avviato dal master, calcolerà le nuove velocità dei corpi a lui assegnati. Una volta terminata la computazione, si mette in attesa sulla barriera. Quando questa viene sbloccata, aggiorna le posizioni e controlla le collisioni sempre dei corpi a lui assegnati. Infine richiama il metodo countDown() del latch, per notificare il master che ha terminato i suoi task.

```
@Override
public void run() {
    try {
        this.computeBodiesVelocity();
        this.barrier.waitAndNotifyAll();
        updatePositionAndCheckCollision();
        latch.countDown();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Listing 2: Worker class

3 Test

I seguenti test sono stati effettuati con la seguente configurazione:

- Dispositivo: MacBook Air M1
- GUI: No

Il primo test eseguito ha l'obiettivo di capire qual'è il numero ottimale di thread per eseguire la simulazione. Sono state eseguite diverse prove utilizzando 5000 corpi e 10000 iterazione ma variando il numero di thread. I risultati conseguiti sono i seguenti:

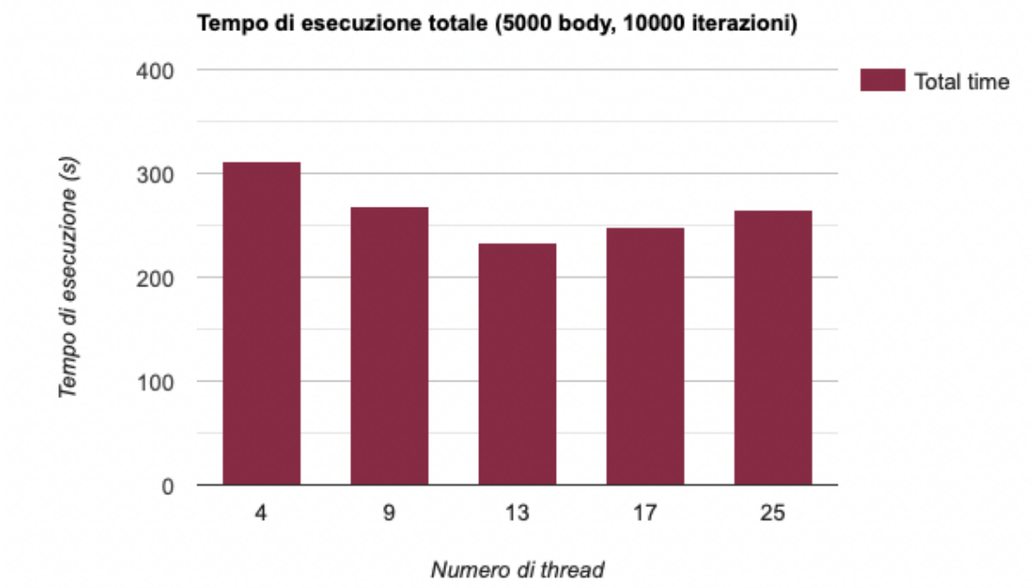

Figure 5: Tempo di esecuzione al variare del numero di iterazioni.

Come si può notare dal grafico il numero ideale di thread per la macchina sulla quale stiamo eseguendo i test risulta essere intorno a 13.

Effettuare un solo test non è però sufficiente per capire effettivamente quale sia il numero ideale di thread. Il test è stato quindi ripetuto variando il numero di corpi ed iterazioni.

Figure 6: Tempo di esecuzione al variare del numero di iterazioni.

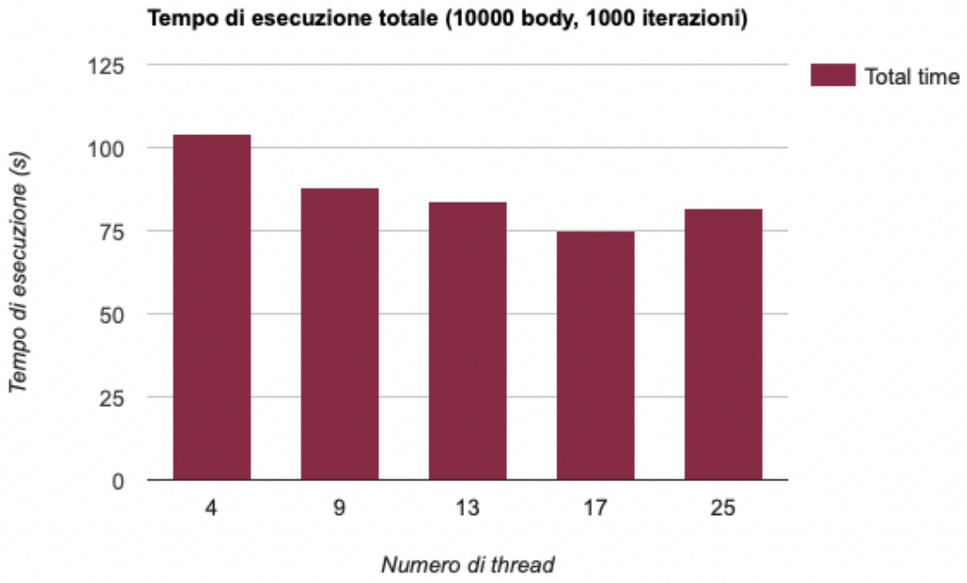

Figure 7: Tempo di esecuzione al variare del numero di iterazioni.

Terminati i test, si è concluso che non esiste un numero ottimale di thread adatto ad ogni simulazione. Con un alto numero di corpi è preferibile avere un alto numero di thread, in modo da dividere tutti i calcoli che devono essere effettuati su più worker. Nel caso di una simulazione con tante iterazioni invece è consigliabile scegliere un numero di thread ridotto, così da ridurre il tempo di creazione dei worker e il tempo che aspettano per sincronizzarsi.

Nella tabella sottostante sono riportati i risultati dei test richiesti dalle specifiche:

Bodies	**Iterations**	**Seq**	**Conc (4)**	**Conc (9)**
100	1000	175ms	381ms	727ms
100	10000	0.6s	2,7s	5.5s
100	50000	2.5s	12.7s	26.6s
1000	1000	4.5s	1.2s	1.6s
1000	10000	42s	10.7s	14.1s
1000	50000	212s	52s	61s
5000	1000	108s	25s	20s
5000	10000	1117s	311s	268s
5000	50000	6247.4s	1669s	1474s

Avendo a disposizione questi dati possiamo ora procedere al calcolo dello speed-up: ovvero un coefficiente che indica l'incremento delle performance di un programma tra la sua versione concorrente e la sua versione sequenziale. Lo speed-up è definito nel seguente modo:

$$SpeedUp = \frac{T_{sequential}}{T_{concurrent}}$$

Nel test più oneroso, ovvero 5000 corpi e 50000 iterazioni, la versione concorrente che utilizza 9 thread del programma ha raggiunto uno speed-up pari a:

$$SpeedUp = \frac{T_{sequential}}{T_{concurrent}} = \frac{6247}{1474} = 4.24$$

4 Verifica della correttezza

L'architettura dell'elaborato è stata verificata con l'ausilio di due tool: JPF e TLA+.

Con JPF abbiamo controllato l'assenza di race condition. Come listener ne abbiamo usato uno presente nelle librerie di JPF ovvero *PreciseRaceCondition*.

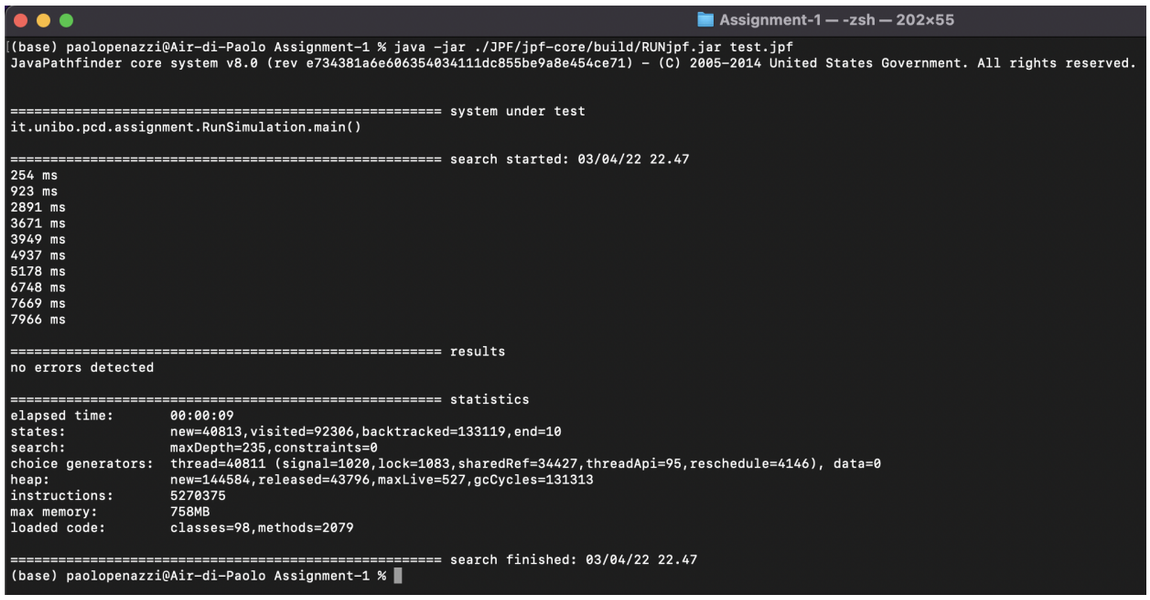

Figure 8: Risultato dell'analisi del programma con JPF.

Tramite il supporto della toolbox *TLA+* abbiamo verificato le principali proprietà del simulatore attraverso alcune specifiche temporali e una invariante. Il linguaggio da noi utilizzato è il PlusCal, il principale linguaggio algoritmico che compila in TLA+, semplificandone l'utilizzo. Sul modello sono state accertate le seguenti proprietà temporali:

```
PositionAfterVelocityComputation == []( (\A n \in 1..NUMBER_OF_WORKERS: positions[n] = 1) =>
    (\A n \in 1..NUMBER_OF_WORKERS: velocities[n] = 1) )
PositionComputation ==  <>(\A n \in 1..NUMBER_OF_WORKERS: positions[n] = 1)
SimTermination == <>(iteration = STEPS)
LatchTermination == <>(latch = 0)
BarrierTermination == <>(barrier = 0)
```

Listing 3: Proprietà temporali verificate sul modello.

E' inoltre stata testata la presente invariante:

```
WorkerIterationGEIterationInvariant == (\A n \in 1..NUMBER_OF_WORKERS: workerIteration[n] >=
    iteration)
```

Listing 4: Invariante verificata sul modello.

Il corretto comportamento del modello è stato testato cercando di tralasciare quelle che sono gli aspetti tecnici del calcolo delle grandezze fisiche dei corpi, focalizzandosi sul giusto coordinamento tra thread.

```
fair+ process master = 0
begin

    evaluateWhile:
        while iteration < STEPS do

            startLatchBarrier:
                startLatch(latch);
                startBarrier(barrier);

            startWorkers:
                creation := 1;

            waitLatchReady:
                waitLatch(latch);
                positions := [position \in 1..NUMBER_OF_WORKERS |-> 0 ];
                velocities :=  [velocity \in 1..NUMBER_OF_WORKERS |-> 0 ];
                creation := 0;

            updateIteration:
                iteration := iteration + 1;
        end while;

end process;
```

Listing 5: Comportamento del processo master.

```
fair+ process worker \in 1..NUMBER_OF_WORKERS
begin

    evaluateWhileWorker:
        while workerIteration[self] < STEPS do

            waitCreationWorkers:
                await creation = 1;

            computeVelocity:
                velocities[self] := 1;
                signalBarrier(barrier);

            waitBarrier:
                waitBarrier(barrier);

            computePositionCollision:
                positions[self] := 1;

            updateMyIteration:
                 workerIteration[self] := workerIteration[self] + 1;

            signalLatch:
                signalLatch(latch);

            awaitWorkFinished:
                await workerIteration[self] = iteration;
        end while;

end process;
```

Listing 6: Comportamento dei processi worker.

5 Conclusioni

L'obiettivo del progetto era quello di realizzare la versione concorrente di un simulatore bidimensionale di N corpi, soggetti a due tipi di forze (repulsiva e di attrito). Riteniamo che il sistema da noi realizzato sia ora concorrente, tenda a massimizzare le performance sull'elaboratore di test e presenti codice che rispetta i principi della buona OOP. Le circostanze però hanno evidenziato come in alcuni casi sia migliore il sistema sequenziale rispetto a quello concorrente. Nel caso della soluzione concorrente il trade-off principale di cui tenere conto risiede sul giusto bilanciamento di thread da impiegare nello svolgimento del lavoro. Sviluppi futuri potrebbero tener conto del sistema su cui viene fatto eseguire il simulatore, in modo tale da affidare maggior carico lavorativo a processori più performanti.

Di seguito uno screenshot della GUI dell'applicazione. La GUI risulta ridimensionabile e reattiva agli input esterni. Rispetto alla GUI iniziale, ora è possibile zoomare attraverso i relativi pulsanti + e -.

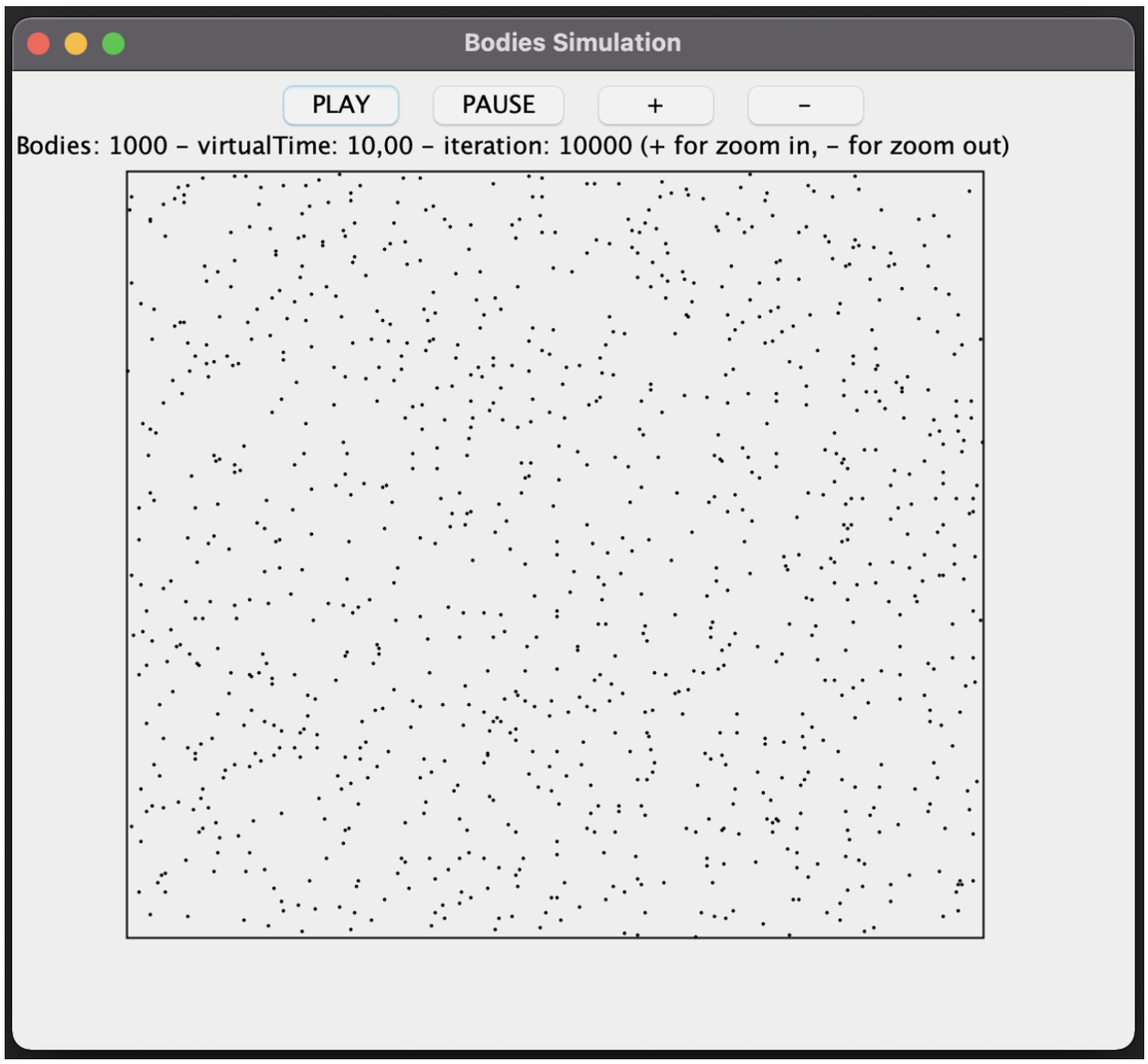

Figure 9: GUI della simulazione.