OpenMP project - Design Document
Middleware Technologies
Prof. Mottola
A.Y. 2017 - 2018

DEBS 2013 - Soccer Monitoring
Arcari Leonardo
Cilloni Stefano
Gregori Giacomo

1 Introduction

1.1 Scope

This document provides a description of the design of our solution to the DEBS 2013 Grand Challenge [1] in its simplified version [2].

1.2 Definitions, abbreviations

T Number of seconds every which to output partial statistics.

K Maximum distance (in meters) a player is eligible for ball possession.

Critical section A section of a program that can be executed only by one thread at a time. In some languages this is called a `synchronized` section.

1.3 Assumptions

Together with the assumptions already set in [2], we set others taking them from the FAQ section of [1]. (a) The playing field is approximated as a perfect rectangle whose vertices are at the following coordinates: $(0, 33965)$, $(0, -33960)$, $(52483, 33965)$, $(52483, -33960)$. (b) Each player has at least 2 sensors. His position is computed as the average of the positions of his sensors. (c) Towards the end of the first half they had a problem with ball transmitters, so computing the ball position is meaningless within that time interval. Therefore we set the end of the first half of the match to the time instant this problem arises, cutting out the last 2.5 seconds.

2 Technologies

Our solution to the problem adopts C++ as programming language (standard 2017). The choice was justified by the language *zero-cost abstraction* philosophy that allowed us to structure the code according to good OO design principles without sacrificing performance. Moreover we used OpenMP [3] as parallelization paradigm because it made it possible to design the application workflow sequentially and easily introduce parallelism in those sections that could benefit.

3 Architectural Design

In this section we describe our application architecture along with a rationale of the design choices. We start from a few numerical considerations. The application is required to be a *real-time stream processor* of data arriving at a frequency of $2000Hz$ for ball position events and $200Hz$ for players position events, summing up to roughly 15000 events per second. For the purpose of computing the ball possession, given a single upcoming player position event only one *ball-player* distance computation is required to decide which player is the closest, because the distance of the ball from all the other players is left unchanged. Although, for a ball position event we must compute the distance for all the players with the new position of the ball to elect the ball possessor, keeping in mind that the ball position update frequency is 10 times the players' one. Therefore we opted for a solution that could exploit as much as possible the available parallelism in the process.

The application architecture is composed of the following main components: (a) `EventFetcher` (Figure 2) reads the incoming events sequentially from the dataset, parses them and adds them to a *mini-batch* of variable size. (b) `GameStatistics` (Figure 3) receives a mini-batch and it elects the ball possessor *per time-instant*. It accumulates the number of times a player was in possession of the ball and outputs partial and whole-game statistics. (c) `Visualizer` (Figure 4) gracefully displays the computed statistics on the desired output stream.

A UML sequence diagram of the components interaction is reported in Figure 5.

4 Algorithms Design

4.1 Events parsing

The process of events parsing hides two choices that are worth mentioning. As the position updates arrive as a stream we must keep a stateful `context` (Figure 1) registering current position of each sensor on the field. It is `EventFetcher` responsibility to update it as long as new events arrive. Moreover, events meaningful for ball possession statistics are grouped in `mini-batches`. The decision of employing mini-batches was taken to maximize the parallelization impact on the application performance. In fact, moving a large number of events to the processors caches considerably reduces the application time spent in I/O operations, since the number of *reads* from secondary memory becomes less than the number of events. When a batch is either full, the game is paused or **T** seconds of game are elapsed, the batch is sent to `GameStatistics`.

4.2 Accumulating ball possessions

This is the parallel section of our application. We want to compute the ball possessor at each time instant. How do we discretize the time? We know that player sensors events arrive at a frequency of $200Hz$ while ball sensors transmit at a frequency of $2000Hz$. This order of magnitude of difference convinced us to use ball position updates as *time-instants*. The process to compute ball possessions is reported in the sequence diagram in Figure 6 and hereby described.

First, we instantiate a `shared` structure $\mathcal{B}$ to store the ball possessor for each time instant. Then for each player `p` on the field we perform the following **in parallel**: (a) Store a local structure $\mathcal{D}$ to hold a list of *ball-player* distances. (b) Store a local copy of player `p` and ball positions at the time instant the first event was added to the mini-batch. (c) For each event in the batch do the following: if it is an update on the position of player `p` then update the local copy of `p` position. If instead it is a ball position event, first update the local ball position, then compute the distance d between the local position of `p` and the new position of the ball. If $d \leq \mathbf{K}$ then append d to $\mathcal{D}$. Otherwise append ∞. (d) After going through all the events, *reduce* $\mathcal{D}$ to $\mathcal{B}$ in a `critical` section. That is, if $\mathcal{B}$ is empty, just copy $\mathcal{D}$ to $\mathcal{B}$. Otherwise, for each time instant t check if $\mathcal{D}_t < \mathcal{B}_t$. If so, store in $\mathcal{B}_t$ that `p` is the current closest player, together with its distance $\mathcal{D}_t$. Because the number of ball position events is seen equal by all the threads, we are sure that $\mathcal{B}$ and $\mathcal{D}$ dimensions are the same across all the threads. (e) When all the threads are done, go through all the elements $(min\_distance, p')$ of $\mathcal{B}$ and increment the ball possession count for player p' by 1 if $min\_distance \neq \infty$.

4.3 Computing partial statistics

In section 4.2 we described how we compute the number of times a player is in ball possession. Computing partial statistics from that is straightforward. When **T** seconds of game are elapsed, a *weighted-average* of the players possessions is performed. Let $\mathcal{A}$ be the accumulator, such that $\mathcal{A}_p$ is the number of instants the player p was in possession of the ball. Let $total = \sum_p \mathcal{A}_p$. Then the partial ball possession statistics for player p is $\mathcal{A}_p/total$. After computing partials, the accumulator is summed to the *whole-game-accumulator* for final game statistics and then it is erased to accumulate the next **T** seconds of events.

5 Implementation details

5.1 Data preparation

As it was provided, the dataset contained different files for the sensors position events and the game interruption events. To make the stream processing possible we merged, through a preprocessing phase, the two files of game interrupts and game events into a single one.

5.2 Managing multiple balls

As reported in the metadata, there are 4 balls each one with its sensor that transmit their position. In order to decide the ball we should compute the distance from we abstracted this away by means of a single `BallPosition` class. This class stores all the 4 sensors, but exposes only the position of the *in-game* ball, that is the ball currently within the field rectangle.

Whenever a sensor is registered as inside the field rectangle, that becomes the position of the *in-game* ball. When the *in-game* ball goes out of the field, `BallPosition` returns an infinite-distant position for the ball. This way, every distance computed with respect to the ball results in an infinite distance and does not participate in the ball possession statistics.

6 Timing analysis

In this section we report the timing analysis of our application to show that the *real-time* requirement is satisfied. We ran the experiments on a Macbook Pro Mid-2012 shipping a Intel(R) Core(TM) i7-3615QM CPU @ 2.30GHz (4 cores, 2 threads per core). All the experiments are run with $\mathbf{K} = 5m$ which maximizes the number of players eligible for ball possession. Results are reported in Table 1. From our results we can conclude that the *real-time* constraint is satisfied as we are able to process 1 second of events in less than 1 clock second. Specifically, we are able to satisfy this constraint in half of the time if we set the batch size to 1 (i.e. every event is processed right away) and in $1/30th$ of the time if we set the batch size to 10^7 and we are asked to output partials every 60 seconds.

T (s)	Batch size	Resp. time (s)	Resp. time per sec
1	1	0.28	0.28
1	10^7	0.05	0.05
60	1	16	0.27
60	10^7	1.8	**0.03**

Table 1: Application response time $[n_{threads} = 4]$

7 Parallelization factor

In this section we report the effectiveness of our parallelization strategy. We ran an experiment to jointly explore the average response time w.r.t. the number of threads and the batch size [5]. In Figure 7 we plot the response time and the speedup w.r.t. the single thread execution. In this experiments we discovered that the best combination for our machine was $n_{threads} = 4$ and $BatchSize = 10^7$. Projecting the analysis on the $BatchSize = 10^7$ axis we illustrate the impact of the number of threads on the execution time, as shown in Figure 8. We achieved a speedup of 27% w.r.t. the single-thread execution running on 4 threads. The performance decrease with 8 threads was predictable. By enabling `-O3` optimizations the compiler will most likely introduce vectorial instructions in the application binary and in general there is a single vectorial ALU on each processor, making the *hyper-threading* penalizing.

8 Code analysis

The application code was analyzed with Valgrind [6] to check for any memory access error which it resulted free of. Moreover the code was constantly profiled with Linux Perf [7] during the whole development process to identify the *hot-spots* and improve the application performance. As of this final version, the application hot-spot is the stream events parsing section, which is expected given the volume of the processed data.

References

[1] ACM *DEBS 2013 Grand Challenge: Soccer monitoring.* `http://debs.org/debs-2013-grand-challenge-soccer-monitoring`

[2] Luca Mottola *MPI/OpenMP Project.* Middleware Technologies A.Y. 2017-2018. `https://groups.google.com/forum/#!topic/middleware1718/tONorXHhOFk`

[3] OpenMP *Enabling HPC since 1997.* The OpenMP API specification for parallel programming. `https://www.openmp.org`

[4] Arcari, Cilloni, Gregori *Speedup Analysis.* `https://github.com/polimi-mt-acg/DEBS-2013-soccer-monitoring/tree/master/docs`

[5] Wikipedia *Hyper-threading.* Intel's proprietary simultaneous multithreading implementation. `https://en.wikipedia.org/wiki/Hyper-threading`

[6] Valgrind. `http://valgrind.org`

[7] Linux Perf. `https://perf.wiki.kernel.org/index.php/Main_Page`

Appendices

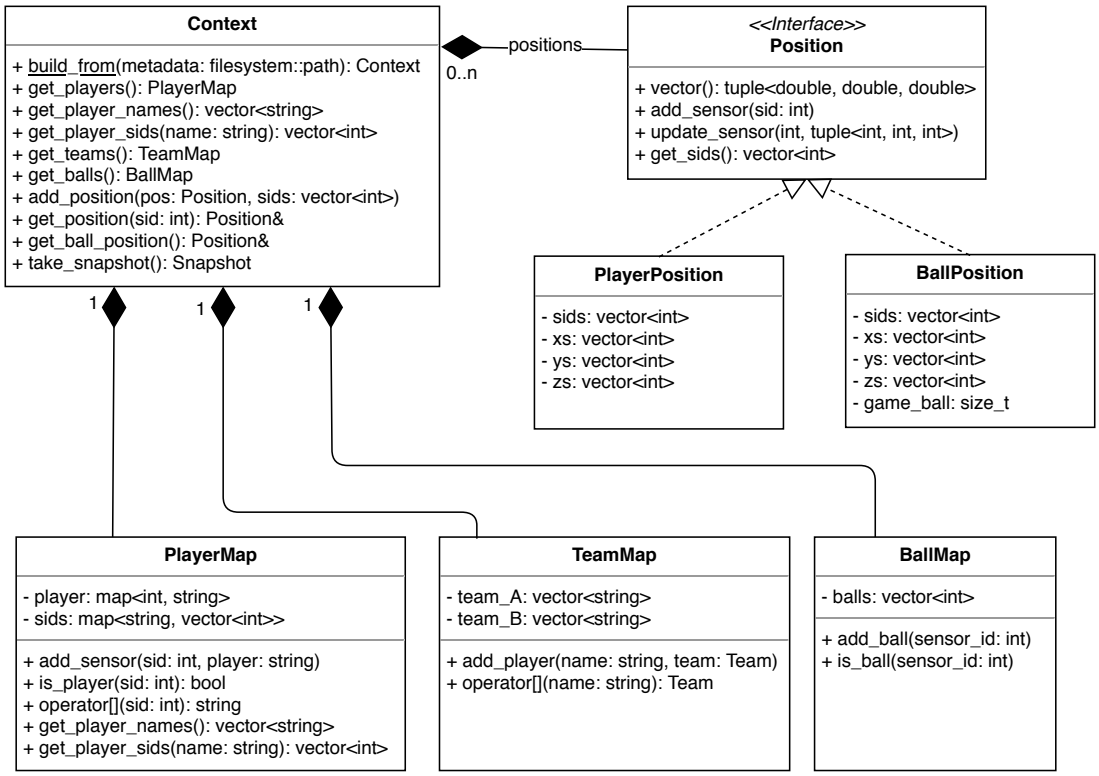

Figure 1: Context component class diagram.

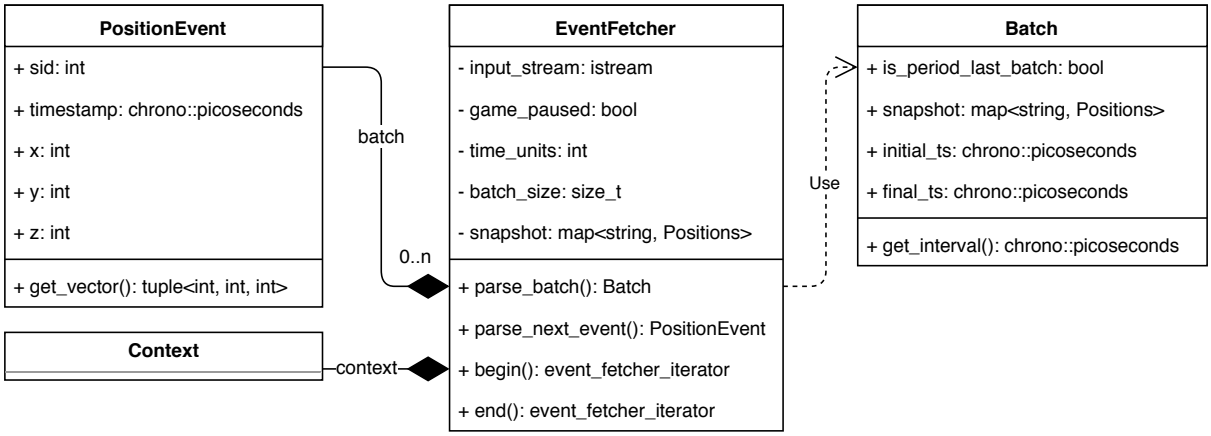

Figure 2: EventFetcher component class diagram. Refer to Figure 1 for details on Context class.

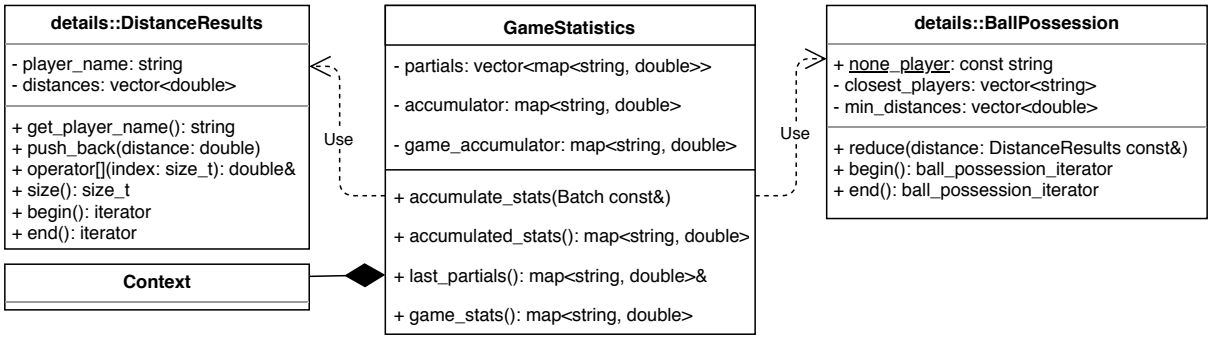

Figure 3: GameStatistics component class diagram. Refer to Figure 1 for details on Context class.

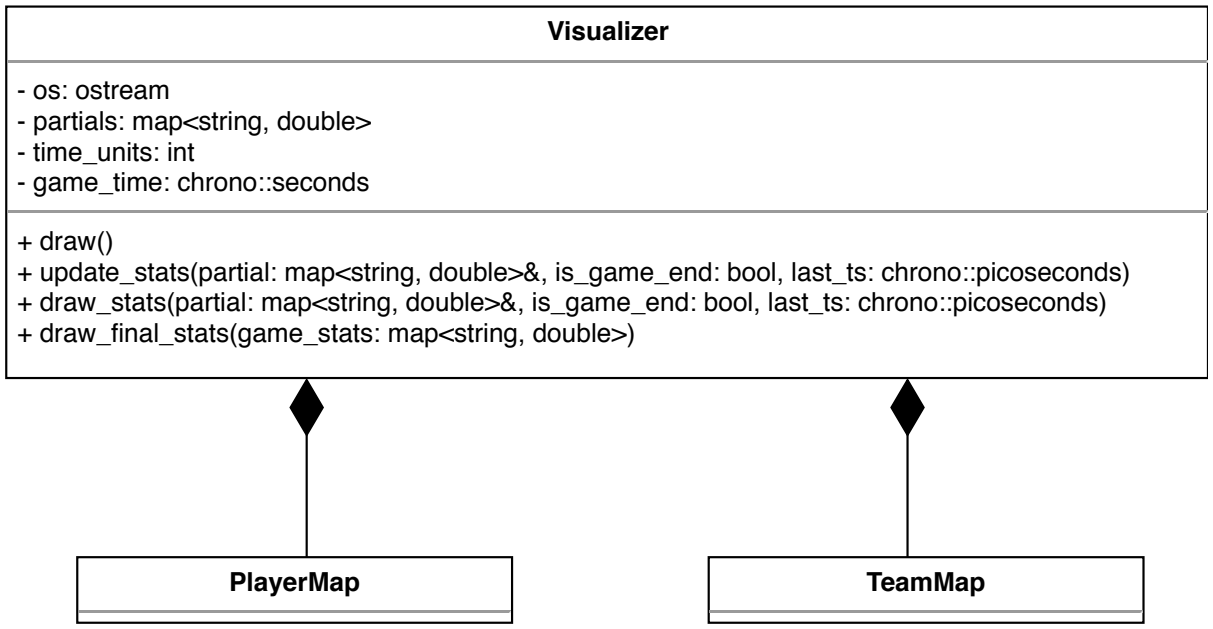

Figure 4: Visualizer component class diagram. Refer to Figure 1 for details on PlayerMap and TeamMap classes.

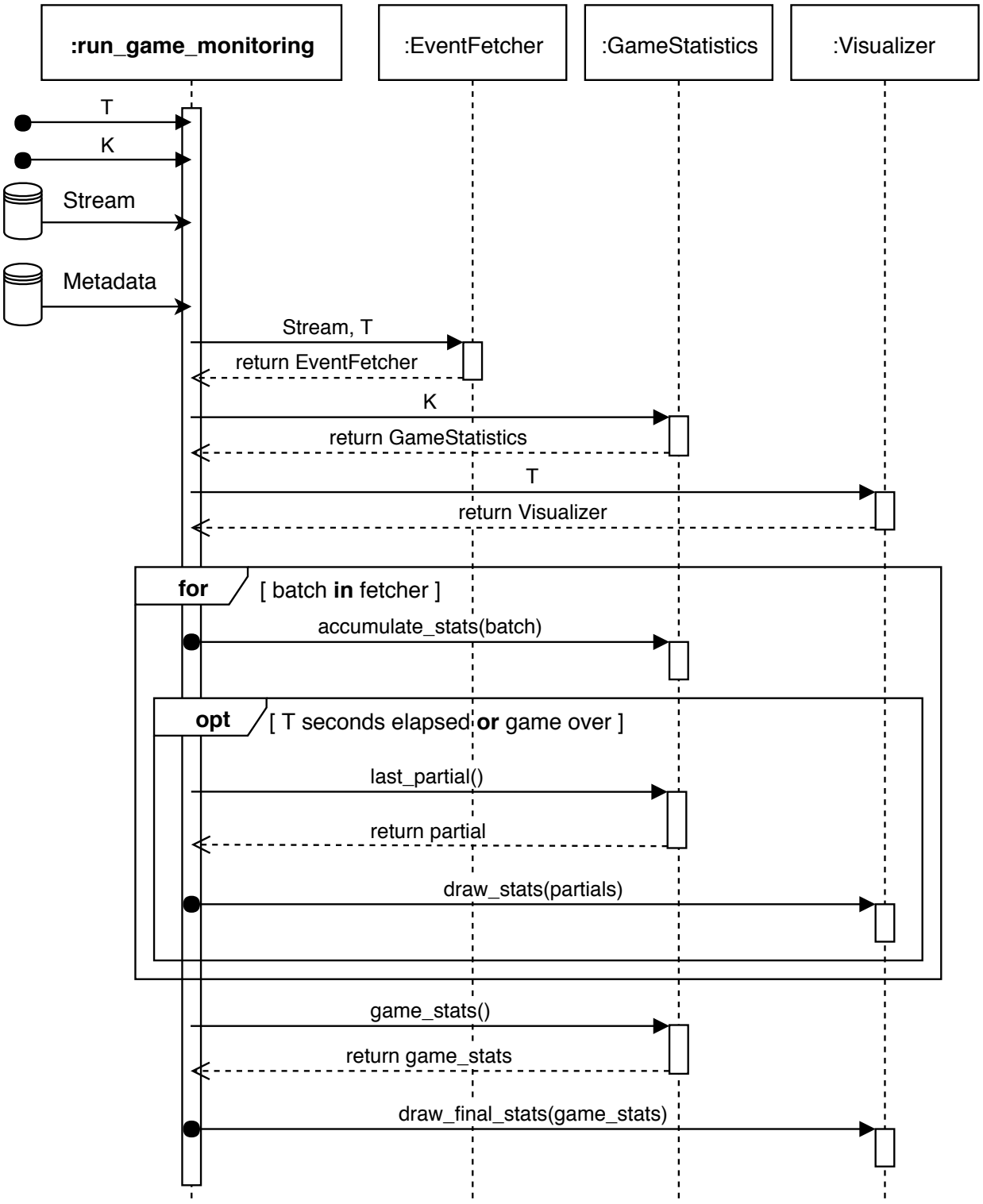

Figure 5: Sequence diagram of the top-level interaction of main components.

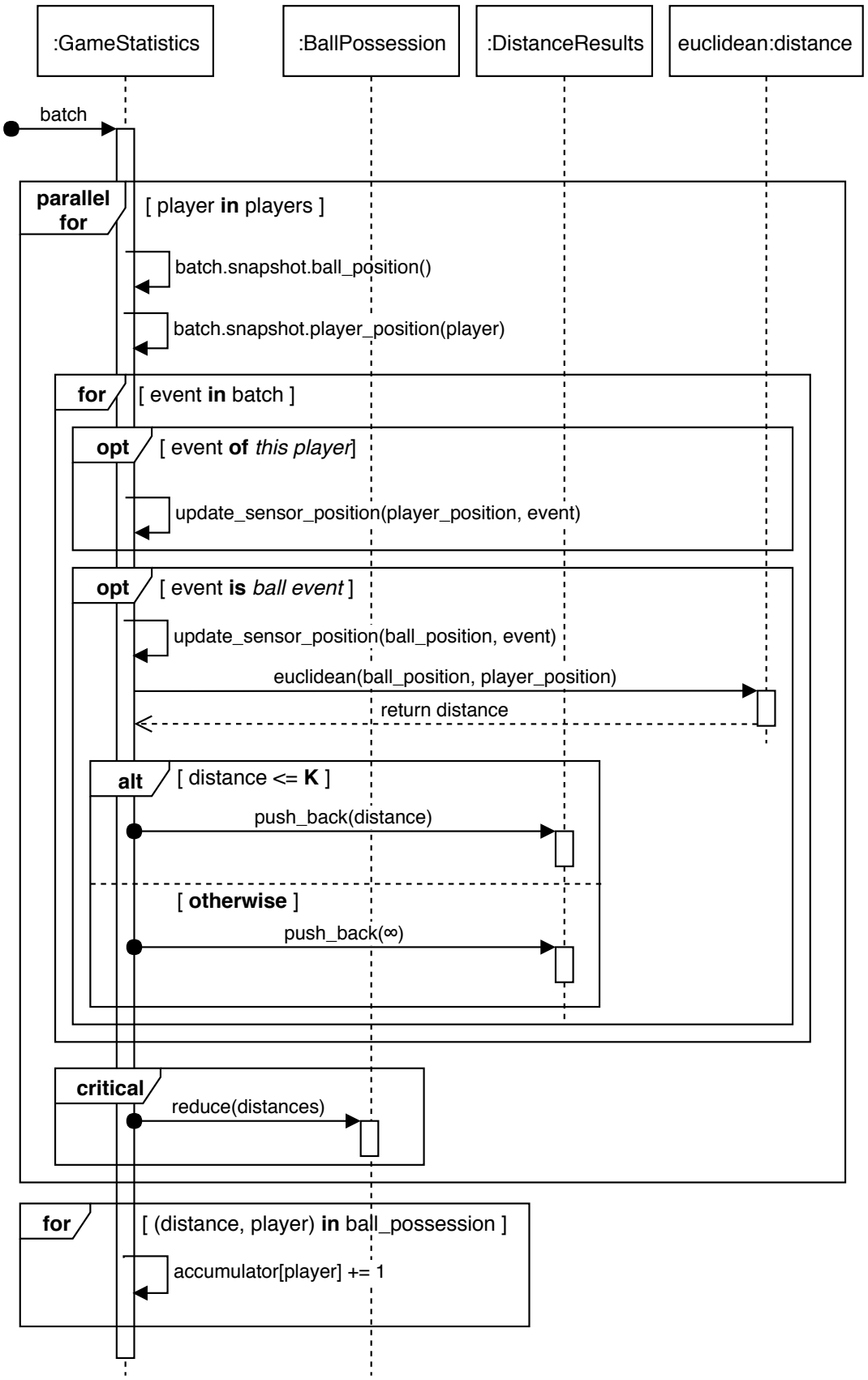

Figure 6: Sequence diagram of the process to compute ball possessions.

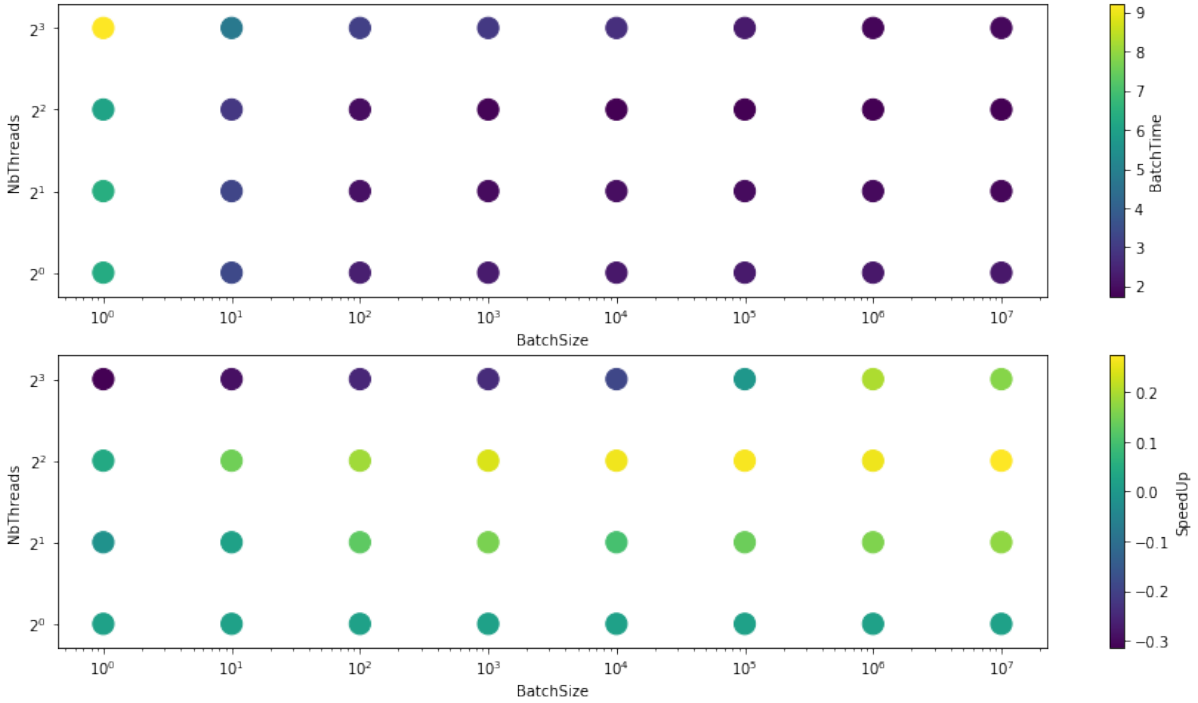

Figure 7: On top, the application response time per-batch-processing with respect to the number of threads and the batch size. Below, the speed-up with respect to the single-threaded experiment.

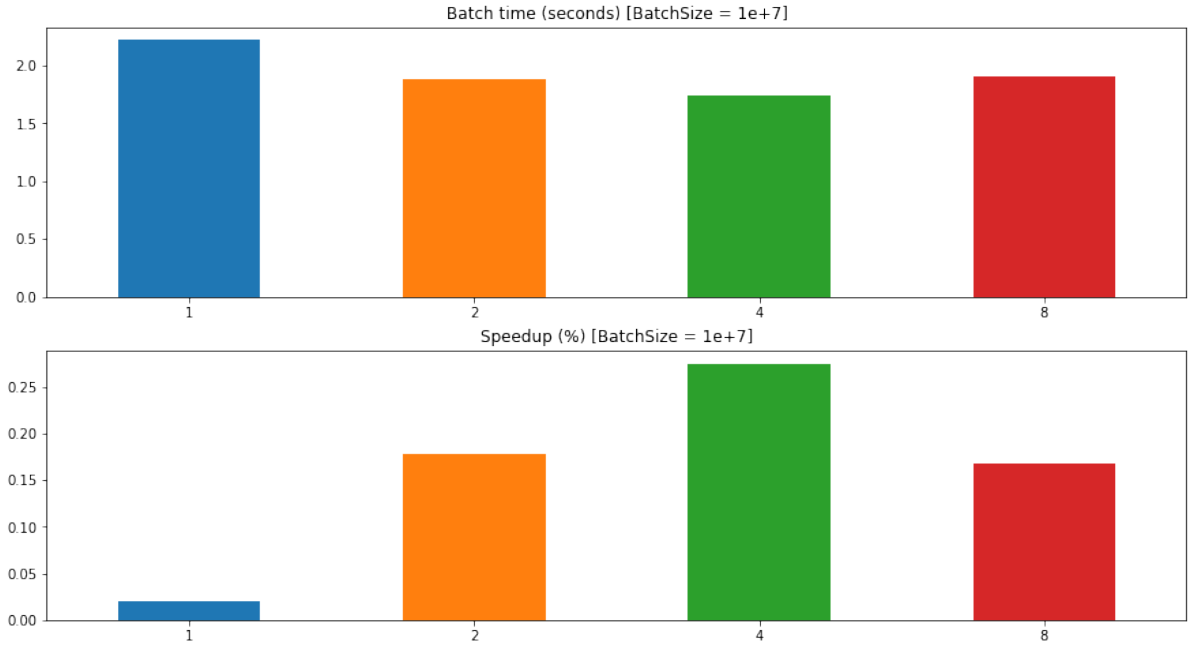

Figure 8: On top, the application response time per-batch-processing with respect to the number of threads (fixed batch size). Below, the speed-ip with respect to the single-threaded experiment.