OpenMP Project Documentation

Luca Santini 883026, Riccardo Remigio 874939

May 31, 2018

1 Introduction

The purpose of this software is to compute the real-time ball possession statistics of a soccer match by using data coming from sensors. Data are originating from wireless sensors embedded in player's shoes and in the ball used during the match. So, in order to satisfy the real-time constraints we used parallelization, in specific we chose to use multithreading through OpenMP.

2 Requirements

Data originates from sensors located near the players' shoes (1 sensor per leg) and in the ball (1 sensor). The goalkeeper is equipped with two additional sensors, one at each hand. The sensors in the players' shoes and hands produce data with 200Hz frequency, while the sensor in the ball produces data with 2000Hz frequency. The total data rate reaches roughly 15.000 position events per second. Every position event describes the position of a given sensor in a three-dimensional coordinate system. A player is considered in possession of the ball when

- he is the player closest to the ball
- he is not farther than K meters from the ball, where K is a parameter taken as input.

Ball possession is undefined whenever the game was paused and the statistics need to be output for every T time units as the game unfolds. Summing up, the software needs to take as input

- an integer value defining K, ranging from 1 to 5
- an integer value defining T, ranging from 1 to 60

It needs to output a string for every T time units of play with the ball possession statistics for each player and for the whole team.

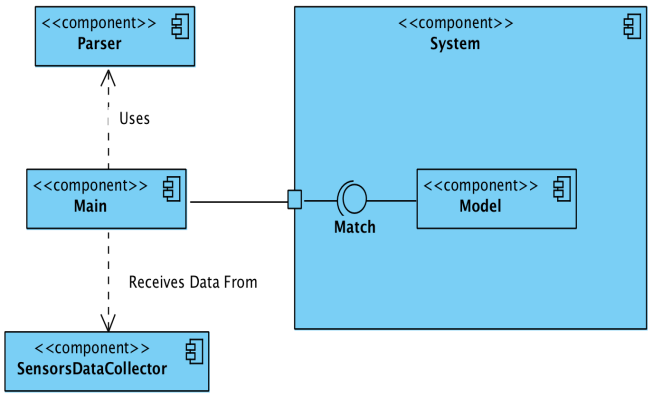
3 Parallelization

As the number of computations to be performed by our software should not be too high to necessarily require execution in a distributed system we chose to execute it on a single machine. The software, as we told before, has to evaluate a high number of events per second and, in order to determine the correct ball possession each event cannot be evaluated independently from the others because when the program process an event representing a ball position, it needs to know the most recent position of all the player sensors and so it has to take these positions from the previous events. If we split the events and we assign one chunk of events to each process, it is possible that a process needs the previous chunks to retrieve the most recent position of some sensors. In this case, each process must have the complete events vector or the processes have to continuously communicate among them. The first solution has a drawback because if we run the software on a single machine, it must have in memory a copy of the vector for each process. Also the second solution has a drawback because the very high number of messages that the processes would have to exchange could probably create an high overhead that would slow down the computation time. In order for each process to have access to the same shared vector of events without the drawbacks described before we chose to use shared memory parallelization and so, to use multithreading instead of multiprocessing.

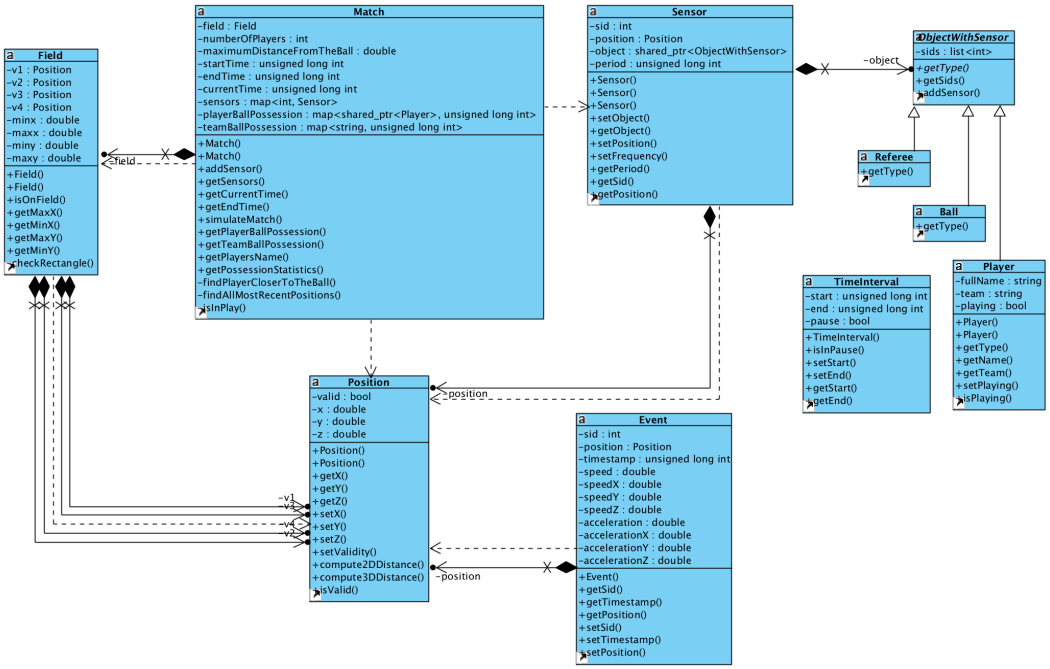
4 Architectural design

4.1 Component View

As said before we developed this software in order to run on a single machine. To test our program we had available a dataset of events of a complete game match, so we created two stubs component(Main and Parser) to parse the dataset and to simulate the real-time execution by passing to the program chunks of dataset each of them corresponding to a period T of real game time.
An high level view of the architecture can be seen on this component diagram.

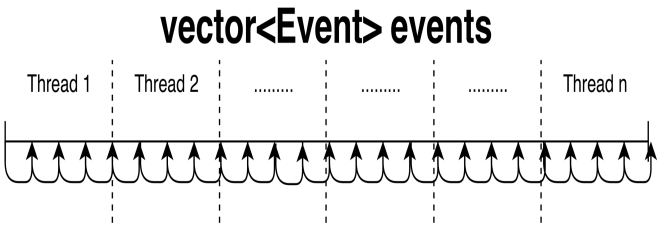


The component Match is the main component of that system that uses the other classes and calculate all the statistics based on the events that receive. All the computations are done mainly in the member function simulateMatch that is called by the Main. A more detailed view of the system can be observed from the class diagram.



4.2 RunTime View

The system works as follow:
simulateMatch receives as input a vector of ordered events and a vector of intervals of time in which the match is in pause. The main thread initializes the variables and creates a number of threads equal to the maximum number of threads supported by the machine in which the program is running. The vector of events is divided into chunks, one for each thread.

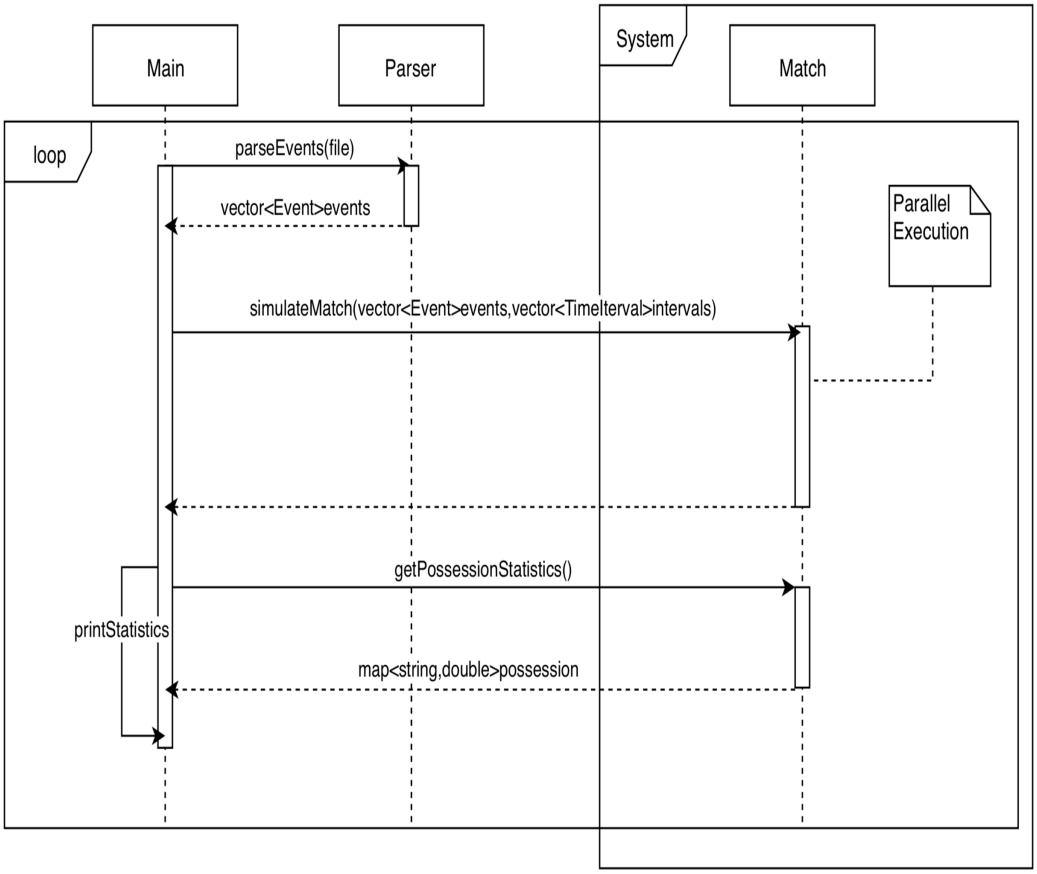


Each thread has a local copy of all the sensors with positions set to zero(tempSens) and

a local collection with the possession for each player set to zero(tempPossession).
Each thread starts to process the chunk of events assigned to it

- If the scanned event refers to a player, then the thread stores the position on its local copy of sensors(called tempSens)
- If the scanned event refers to a ball in the field when the game is in progress, then the thread:
  - checks to have the most recent positions of all the players and eventually retrieves them by following a specific procedure
  - looks for the player closest to the ball and, if its distance from the ball is less than the K parameter, gives him a time possession equals to the ball sensor period.

At the end of the cycle the thread that has scanned the last chunk of events has the most updated position, so it saves his positions in the global collection(Sensors).
Then each thread adds for each player its own possession value to the global collection(playerBallPossession).



5 Conclusions

To ensure that the system is able to calculate fast enough the possession statistics we printed out the effective computation time (execution of simulateMatch) and we can state that by running the software in a reasonable recent machine the time constraints were widely satisfied. We also compared our results with the real results calculated "by hand" and we noticed that by using K=1.5 the results almost coincides.