Appendix A — Parallelization of the RulEvolution-TicTacToe System

A.1 Adopted Parallelization Framework

The parallel version of the *RulEvolution-TicTacToe* system was developed using the **OpenMP** library, chosen for its simplicity of integration with C++ and its capability to manage concurrent execution of multiple threads within the same process through a small set of directives. This choice proved ideal for a system of modest computational complexity, whose data structure is nevertheless easily parallelizable.

The code sections that benefit most from parallelization are:

- the independent evaluation of logical rules (`RulEvolutionRules::evaluate()`), where each rule can compute its contribution to the available cells in parallel;
- the **Super-Training** phase, in which multiple games are executed simultaneously on different threads, fully exploiting the available cores (the system was tested on a personal computer equipped with an ARM Snapdragon X Elite processor featuring 12 cores).

The program was compiled with the `/openmp` flag and the `USE_OMP` macro enabled, automatically activating the OpenMP parallelization directives.

A.2 Introduction of the Super-Training Mode

To enable large-scale learning, a special mode named **Super-Training** was introduced. In this mode, the program explicitly excludes any interaction with a human player — which would considerably slow down the training process — and restricts the simulation to two predefined non-human configurations:

- *Stochastic vs RulEvolution*;
- *RulEvolution vs RulEvolution.*

The system then automatically generates a batch of parallel games between autonomous agents. Each game is handled by a separate thread, equipped with its own copy of the `LearningModule`. This design prevents simultaneous access conflicts (*data races*) to shared structures. At the end of the parallel execution, the results are merged into a single global instance of the learning system.

A.3 Weight Merging Strategy

To coherently integrate the results from different threads, a simple **arithmetic mean** of the weights computed by each local learning process was adopted. This approach was motivated by the following considerations:

1. the arithmetic mean is computationally lightweight, minimizing the merging overhead;
2. the combined quantities (the rule weights) represent linear and independent estimates, so their arithmetic mean preserves their semantic meaning;

3. this method guarantees numerical stability and can be easily extended to an arbitrary number of threads or distributed nodes.

The arithmetic mean thus appears to be an appropriate aggregation method for linear, independent estimators such as the rule weights used in this system.

A.4 Comparative Tests: Sequential and Parallel Execution

Four types of tests were performed:

a) Standard training in sequential mode;

b) Standard training in parallel mode (independent matches);

c) Super-Training in sequential mode (sequential games);

d) Super-Training in parallel mode (simultaneous game batches).

Each test consisted of 100 training matches. Execution times were measured using the `omp_get_wtime()` function (with a fallback to `std::clock()` in the absence of OpenMP).

A representative experimental result on an ARM64 architecture (12 cores) is shown below:

Mode	**Matches**	**Time (s)**
Standard — Sequential	100	5.9
Standard — Parallel	100	2.2
Super-Training — Sequential	100	3.6
Super-Training — Parallel	100	1.0

The results show a temporal improvement proportional to the number of active cores, with an average **speed-up** of approximately 3.5× for the parallel batch, confirming the correct implementation of the OpenMP directives and the absence of significant synchronization bottlenecks.

A.5 Tools for Detecting Data Races

OpenMP does not provide a built-in function to detect *data races* during execution. However, external analysis tools such as the **LLVM/Clang Thread Sanitizer** and the **Intel Inspector** can identify potential conflicts:

- `clang -fsanitize=thread` (Thread Sanitizer);
- `Intel Inspector XE` or `Visual Studio Concurrency Visualizer`.

In the present case, no conflicts were reported by these tools, confirming that the use of local learning-module copies within each thread effectively eliminated any race conditions.

A.6 Rationale for Not Using GPU (CUDA)

GPU-based parallelization (CUDA or OpenCL) was not adopted for the following reasons:

1. the analyzed problem (the *TicTacToe* game) is computationally too lightweight to justify data transfer overhead to GPU memory;

2. the nature of the rules is mainly logical–decisional rather than numerical, making their execution on a GPU inefficient — parallelization on GPUs is most advantageous when operations are numerically intensive and vectorizable;

3. the primary goal of this work was the conceptual validation of the *RulEvolution* paradigm rather than the optimization of raw performance.

Nevertheless, extending the system to GPUs represents a natural evolution of the project. For more complex applications, involving numerous rules or intensive numerical calculations — such as multidimensional fuzzy systems, large-scale games, or astronomical spectrum analysis — the use of CUDA or libraries like *Thrust* and *OpenACC* could provide a substantial efficiency boost.

A.7 Conclusions

The parallelization of the *RulEvolution-TicTacToe* system confirmed that the proposed paradigm naturally lends itself to distribution across multiple processors, thanks to its inherently independent rule-based structure. The *Super-Training* mode enabled simultaneous training free from conflicts and scalable with the available hardware resources.

Although the presented case study is elementary, the results demonstrate:

- the coherence of the multi-threaded approach;
- the numerical stability of the weight-merging process;
- the validity of distributed learning as a foundation for future developments.

This experiment represents a preliminary step toward applying the *RulEvolution* paradigm to more complex contexts, where parallelization and the use of hybrid CPU–GPU architectures could play a decisive role in improving both efficiency and scalability.