Monte Carlo Search Tree: a parallel C++ implementation

Andrea Mascaretti, Michele Pellegrino

Politecnico di Milano

29th October 2018

Game Theory

- Game Theory is the name given to the methodology of analysing interactive decision making problems using mathematical tools ([MSZ13])
- Agents (endowed with a utility function associating a degree of preference over a set of outcomes) are called to make some choices, maximising their own utility

Game Theory

- The extensive-form representation ([Gat18]) of a perfect-information game is a tuple $(N, A, V, T, \iota, \rho, \chi, U)$
  - $N = \{1, 2, \ldots, n\}$ is the set of players
  - $A = \{A_1, A_2, \ldots, A_n\}$ is the actions, where $A_i = \{a_1, a_2, \ldots, a_{m_i}\}$ is the set of actions of player i
  - $V = \{V_1, V_2, \ldots, V_n\}$ is the set of decision nodes, $V_i = \{w_1, \ldots, w_{l_i}\}$ being the set of decision nodes of player i
  - T is the set of terminal nodes
  - $\iota : V \to N$ is the function returning the player at the node

Game Theory

- The extensive-form representation ([Gat18]) of a perfect-information game is a tuple $(N, A, V, T, \iota, \rho, \chi, U)$
  - $\rho : V \to \mathcal{P}(A)$ is the function return the available actions at a given node
  - $\chi : V \times A \to V \cup T$ is the function assigning the next node (either decision or terminal) to each pair $\langle w, a \rangle$ such that $a \in \rho(w)$. It is not defined otherwise
  - $U = \{U_1, U_2, \ldots, U_n\}$ is the set of utility functions, with $U_i : T \to \mathbb{R}$ being the utility function of player i

Game Theory

- A zero-sum game is a game for which $\forall t \in T, \sum_{i=1}^{n} U_i(t) = 0$
- We will consider two two-player zero-sum games, namely Tic-tac-toe and Nim.
- In every two-player game in which there is set of outcomes such that either there is a draw or we have a winner, one and only one of the following alternative holds
  - Player 1 has a winning strategy
  - Player 2 has a winning strategy
  - Each of the two player has a strategy guaranteeing at least a draw

Game Theory

- The aim of players competing is that of finding a specification of the moves to take at every node, responding to every action played by the other players

Adversarial Search

- The problem of finding an optimal solution can be seen as a search problem ([RN16])
- Search problem are solved by means of *search trees*. Every node of the tree represents a state of the game (*i.e.* a node in the extensive-form representation)
- Other important information is stored in the node: the parent and children nodes and the last action
- There exist algorithms to find optimal solutions for two-player zero-sum games of the type described (namely the Minimax and the Alpha-Beta pruning algorithms). However, such algorithms are impractical for trees with a high-branching factor

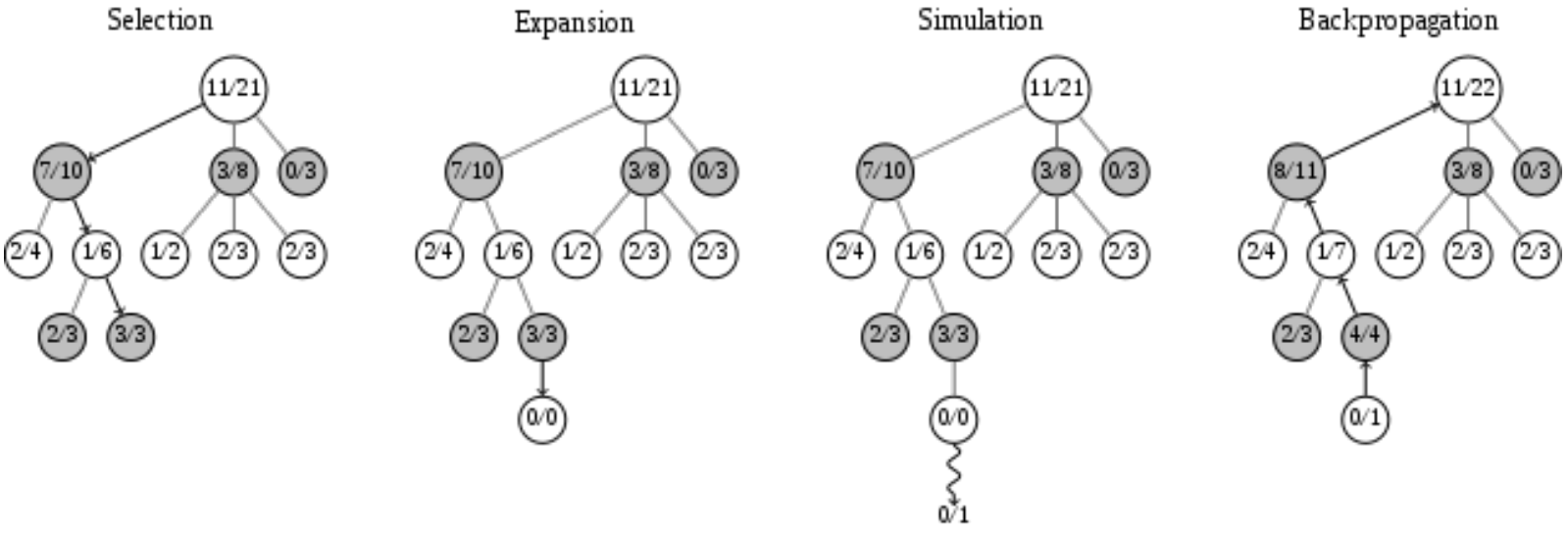
Monte Carlo Tree Search

- Monte Carlo Tree Search is a heuristic technique employed to deal with the issue of search trees with high-branching factors
- It has become popular after AlphaGo managed to beat professional player Lee Sedol in an official match
- `mtsc.ai` presents an overview of the algorithm and a bibliography on the topic, as well as a Python and Java implementation

Monte Carlo Tree Search

- The algorithm relies on four steps
  1. **Selection**. Starting from a root node r, successive child node are selected until a leaf node l is reached. The root is the current game state and a leaf node of the search tree is a node from which no simulation has been initiated
  2. **Expansion**. If the game status of l is not in T, create one (say c) or more child nodes and pick one
  3. **Simulation**. From the child node, complete a number of roll-outs, randomly playing until a terminal node is reached
  4. **Backpropagation**. Given the results, update the information on the nodes from c to r

Monte Carlo Tree Search



Monte Carlo Tree Search

- The issue is that of maintaining a good balance between *exploration* and *exploitation*. In other words, we want to focus on nodes that appear promising, but not at the expense of not exploring others that might be better and about which we simply do not have enough information

Monte Carlo Tree Search

- To decide, we implemented the criterion known as UCT:

$$\mathrm{UCT} = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

- w_i is the number of wins after the ith move
- n_i is the number of simulations for the node after the ith move
- N_i is the total number of simulations after the ith move
- c is the exploration parameter, theoretically equal to $\sqrt{2}$ (see [KS06])

Class Game

template <class Action> Game	
+ get_terminal_status(void):	bool
+ get_agent_id(void):	int
+ apply_action(Action):	void
+ get_actions(void):	std::vector<Action>
+ random_action(void):	Action
+ evaluate(void):	int
+ set_seed(int):	void

Class OxoGame

OxoGame: public Game<OxoAction>	
- is_terminal:	bool
- agent_id:	int
- board:	std::array<std::array<int, 3>, 3>
- last_action:	OxoAction
- winner_exists:	bool
- random_action_seed:	int
- gen:	std::default_random_engine
- update_terminal_status(void):	void
+ get_terminal_status(void):	bool
+ get_agent_id(void):	int
+ apply_action(OxoAction):	void
+ get_actions(void):	std::vector<OxoAction>
+ random_action(void):	OxoAction
+ evaluate(void):	int
+ set_seed(int):	void
+ get_last_action(void):	OxoAction

Class NimGame

template<unsigned int N1, unsigned int N2, unsigned in N3> NimGame: public Game<NimAction>	
- is_terminal:	bool
- agent_id:	int
- last_action:	NimAction
- random_action_seed:	int
- gen :	std::default_random_engine
- board:	std::array<unsigned int, 3>
- update_terminal_status(void):	void
+ get_terminal_status(void):	bool
+ get_agent_id(void):	int
+ apply_action(NimAction):	void
+ get_actions(void):	std::vector<NimAction>
+ random_action(void):	NimAction
+ evaluate(void):	int
+ set_seed(int):	void
+ get_last_action(void):	NimAction

Action structure

OxoAction: public Action	
+ row:	int
+ column:	int
+ to_string(void):	std::string
+ operator==(OxoAction):	bool

NimAction: public Action	
+ pile:	unsigned int
+ number	unsigned int
+ to_string(void):	std::string
+ operator==(NimAction):	bool

Class Node

template< Game, Action > Node	
- game_state:	Game
- parent:	Node*
- last_move:	Action
- children:	vector<NodePointerType>
- possible_moves:	vector<Action>
- player:	int
- wins:	double
- visits:	unsigned
+ update(double, unsigned):	void
+ all_moves_tried(void):	bool
+ has_children(void):	bool
+ make_child(Move):	NodePointerType
+ print_node(void):	void
- erase_move(Move):	void

Class for MC Search Tree

template< Game, Action > MonteCarloSearchTree	
- root:	NodePointerType
- current_game_node:	NodePointerType
- seed:	int
- rng:	random_engine
- is_parallel:	int
- outer_iter:	unsigned
- inner_iter:	unsigned
- ucb_constant:	double
+ uct_search(void):	Move
+ change_current_status(Move):	void
+ print_current_status_info(void):	void
- compute_ucb(double):	NodePointerType
- best_child_ucb(NodePointerType):	NodePointerType
- select(void):	NodePointerType
- expand(NodePointerType):	NodePointerType
- rollout(NodePointerType):	double
- back_propagation(NodePointerType, double):	void

Parallelization strategy

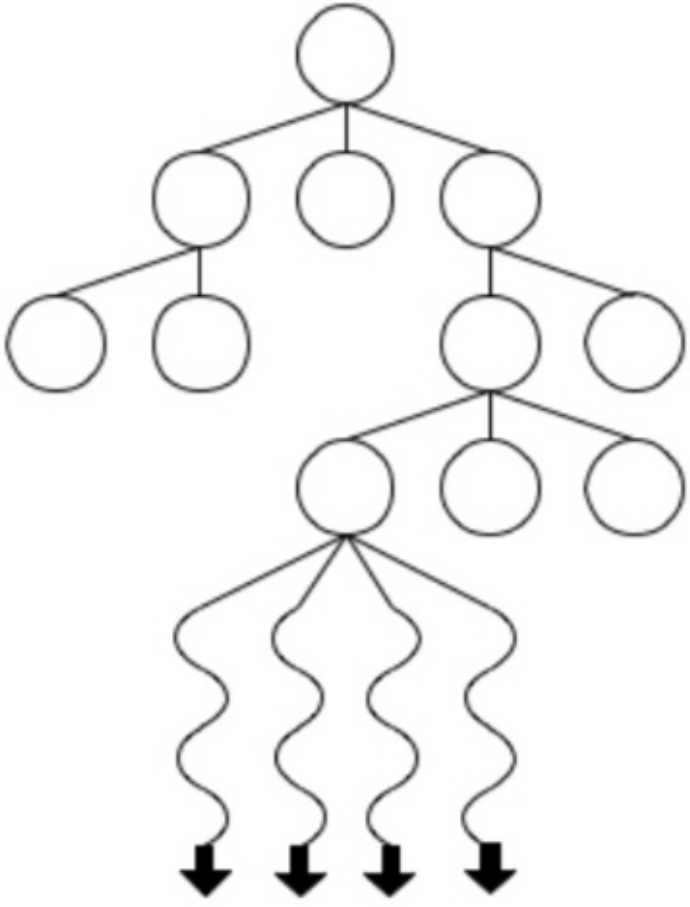
- [CWvdH08] reports three main parallelization methods for MCTS
- **Leaf parallelization** consists in having only one thread traversing the tree; independent games are played for each available thread, starting from the *same* leaf node
- **Root parallelization** consists in building several trees (one per thread), and then merging them at the end of the search procedure
- **Tree parallelization** consists in playing several independent games from *different* leaves, one per thread

Comparison

Tests performed on Go show, according to [CWvdH08]:

- leaf parallelization to be the weakest strategy (having a speedup of 2.4 on 16 cores)
- root parallelization to be the strongest (having a speedup of 14.9 on 16 cores)
- tree parallelization to be intermediate (having a speedup of 8.5 on 16 cores); the authors claims, however, that this strategy allows for better exploration of the solution space

Leaf parallelization



We opted for the leaf parallelization strategy, as it results to be the simplest to implement using MPI; the other two strategies require either having access to shared memory (tree parallelization), or large communication overheads (root parallelization)

Overview of the algorithm

Algorithm 1: Monte Carlo Search Tree

```
Data: seed, iter_in, iter_out, ucb_const
Initialize MCST;
while current status is not terminal do
    for i = 0 : iter_out do
        Select best UCB node;
        Expand best UCB node;
        for j = 0 : iter_in do
            Simulate random games, starting from the state of the
              expanded leaf;
        end
        Aggregate results;
        Perform back-propagation;
    end
    Select best move;
    Move current status;
end
```

Data Analysis: Nim

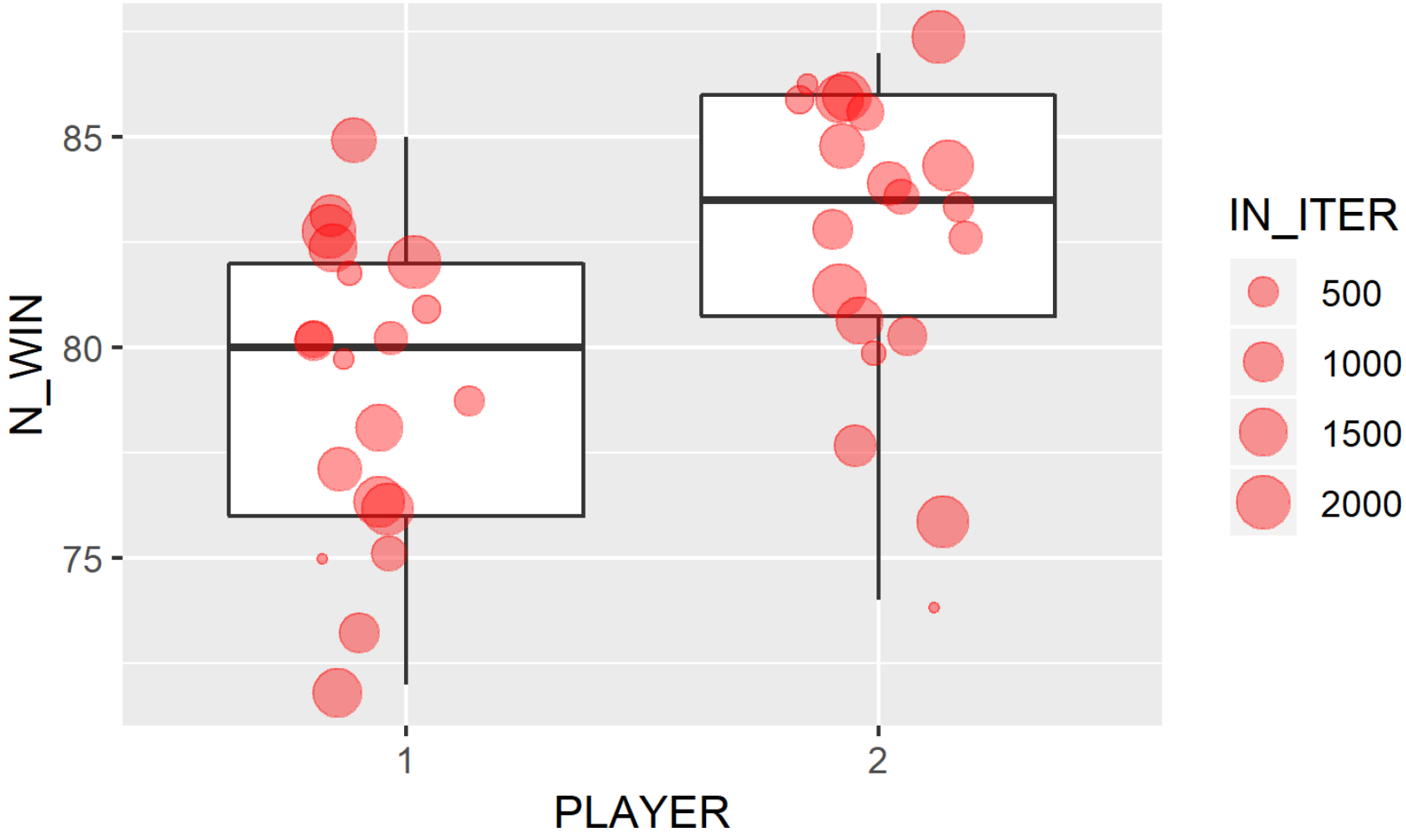


Figure 1: Boxplot of wins according to player at Nim

Data Analysis: Oxo

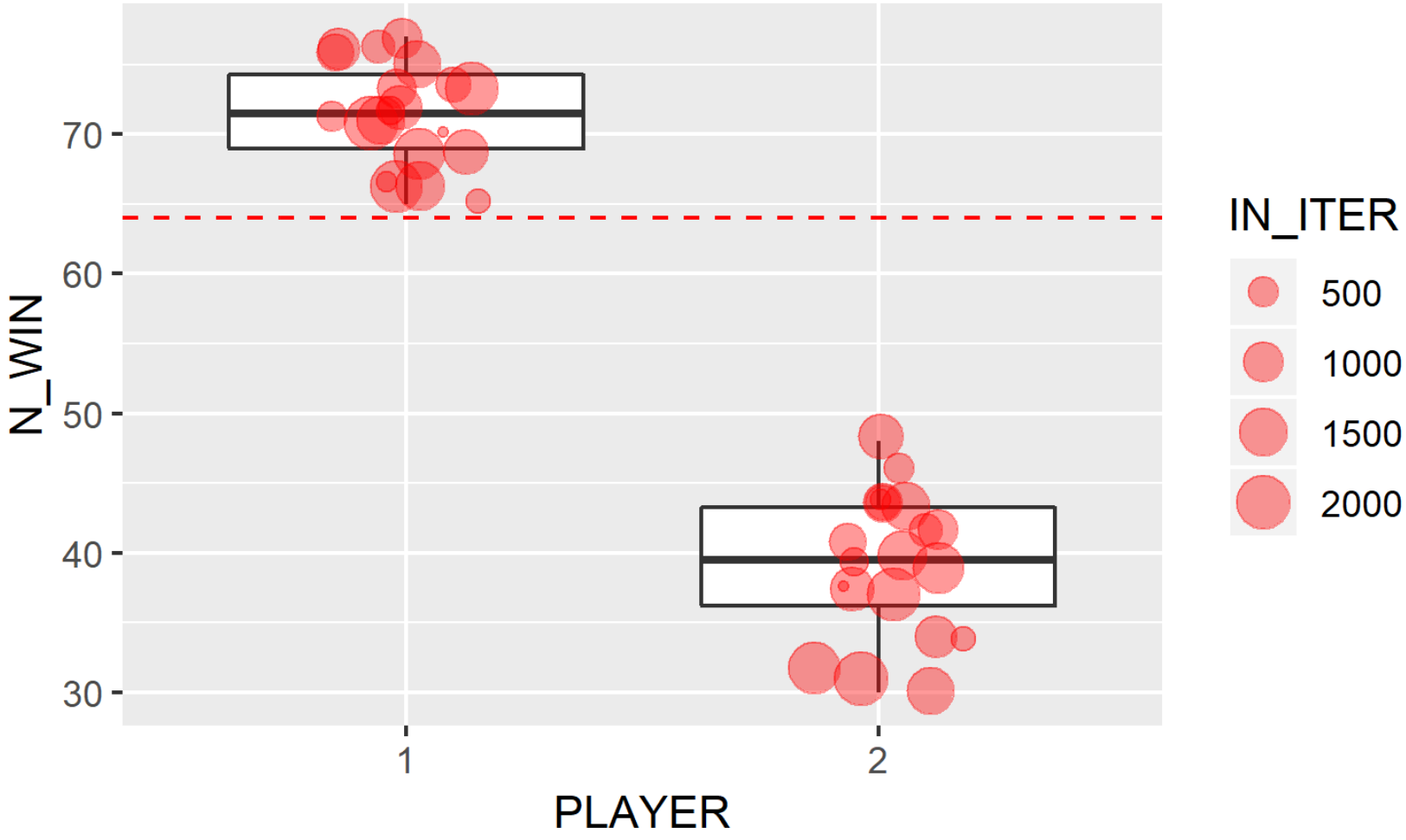


Figure 2: Boxplot of wins according to player at Oxo

Data Analysis: Oxo

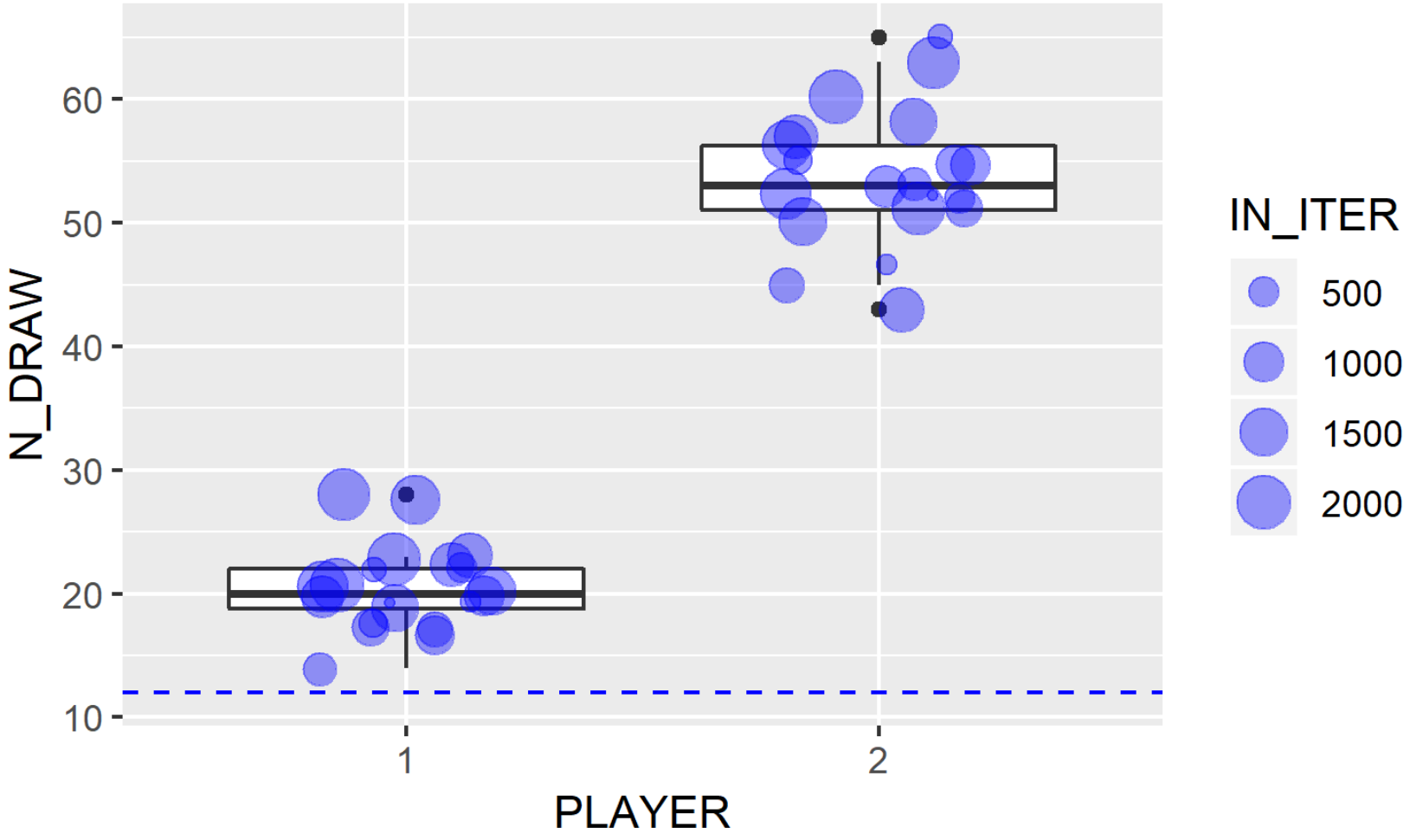


Figure 3: Boxplot of draws according to player at Oxo

Data Analysis: Oxo

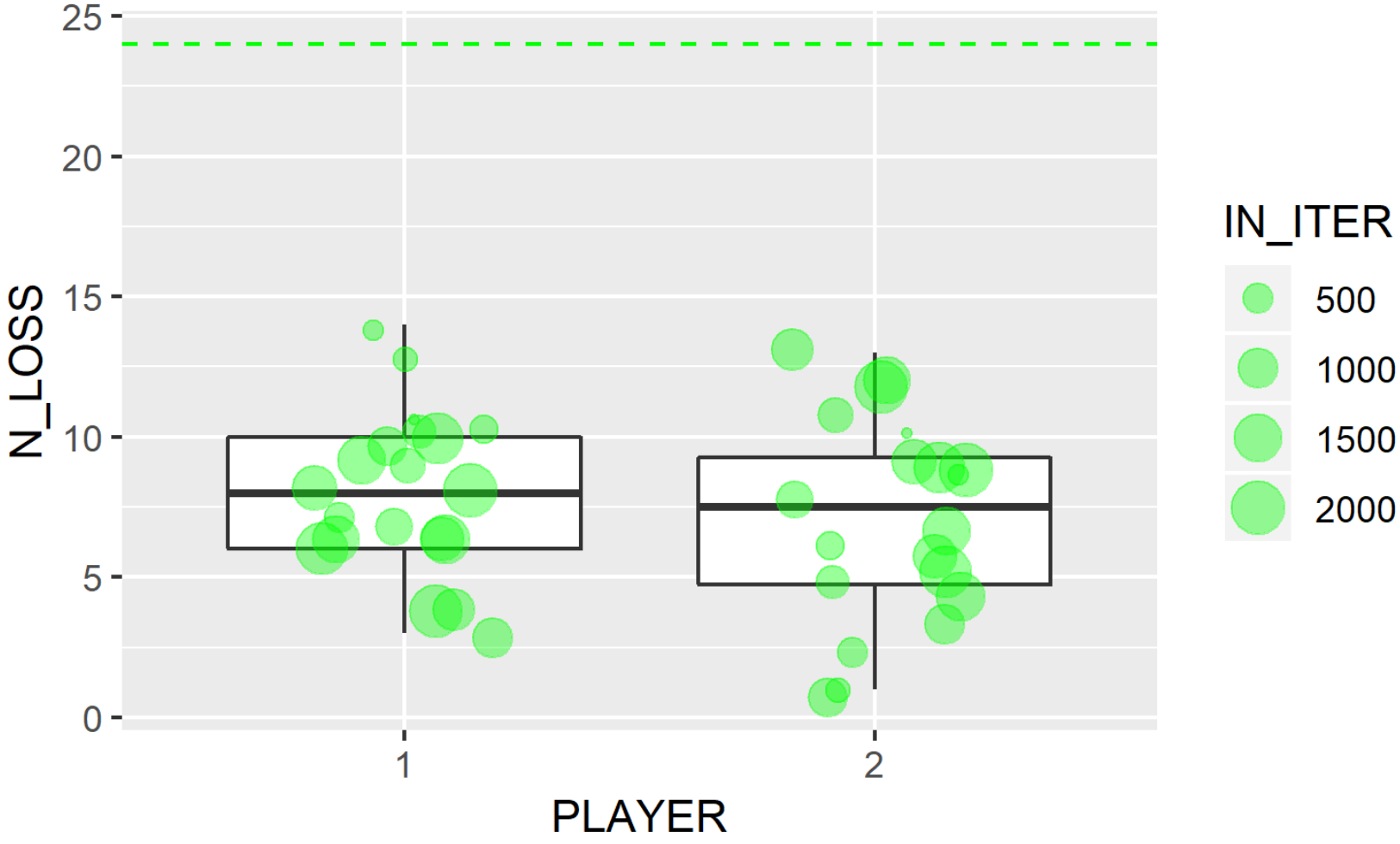


Figure 4: Boxplot of losses according to player at Oxo

Speedup: Nim

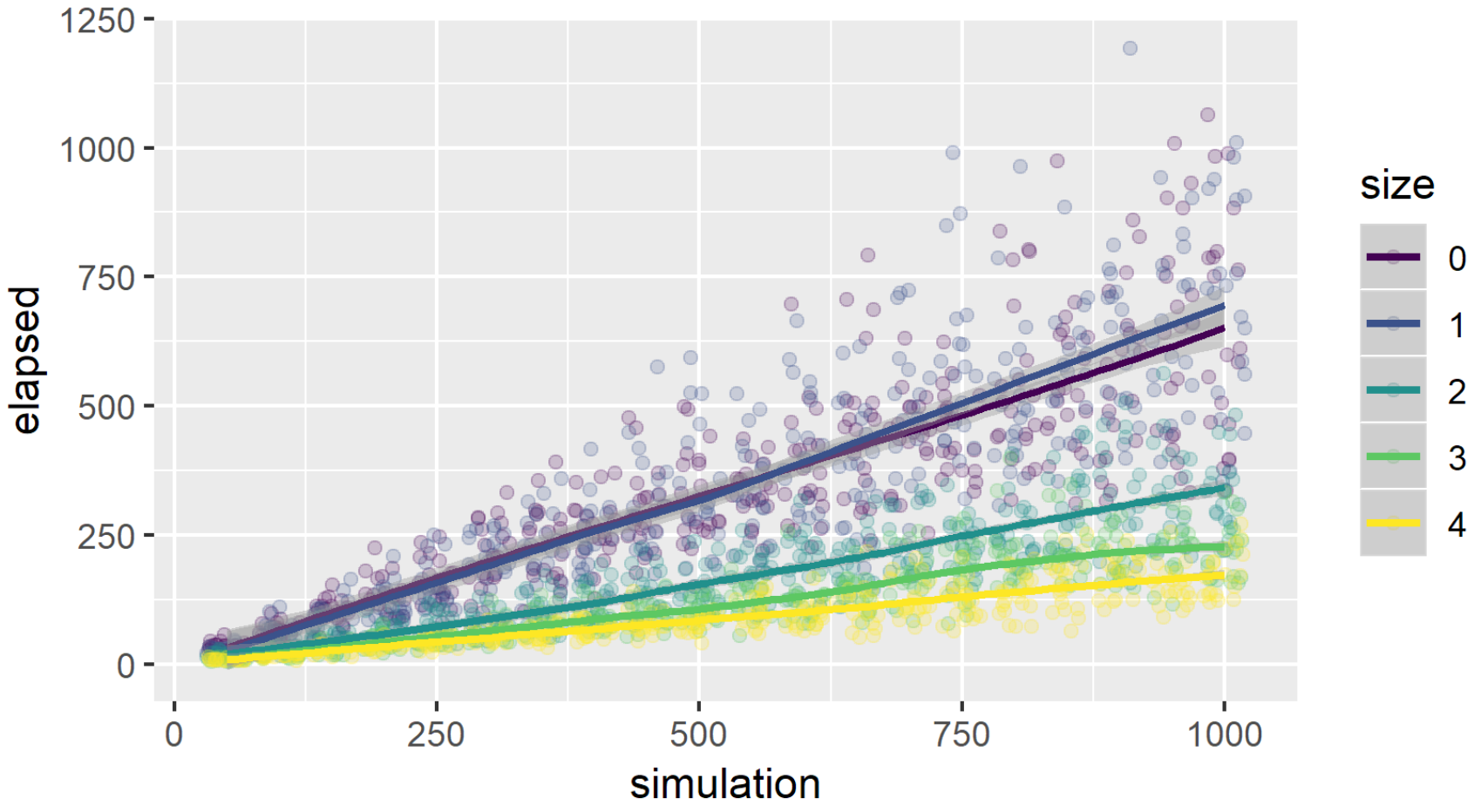


Figure 5: Elapsed time (ms) over number of game simulations, for each number of processes (0 is the serial code)

Speedup: Nim

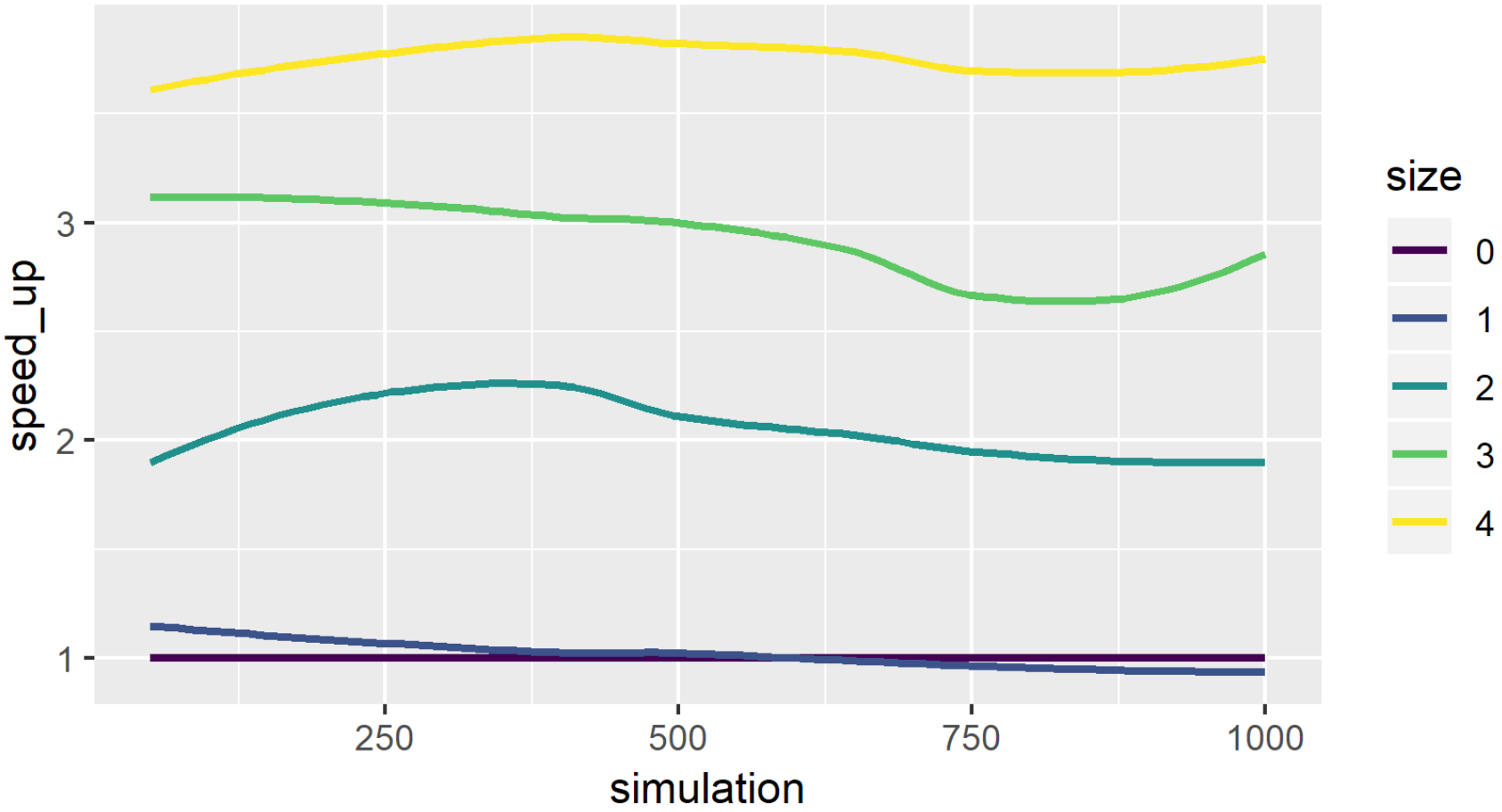


Figure 6: Plotting the time ratio between scalar-algorithm time and multiprocess-algorithm time at a fixed number of iterations

Speedup: Oxo

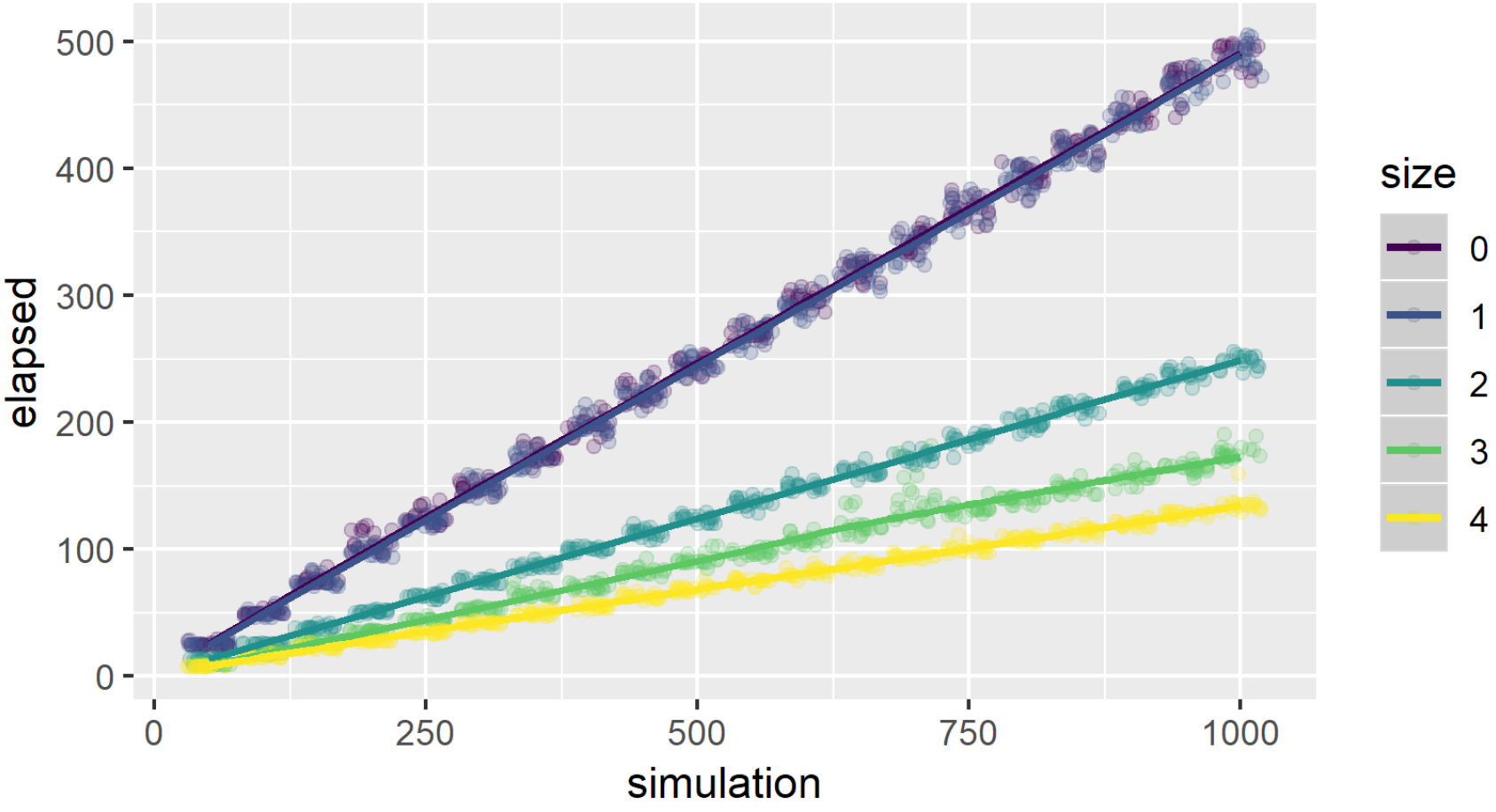


Figure 7: Elapsed time (ms) over number of game simulations, for each number of processes (0 is the serial code)

Speedup: Oxo

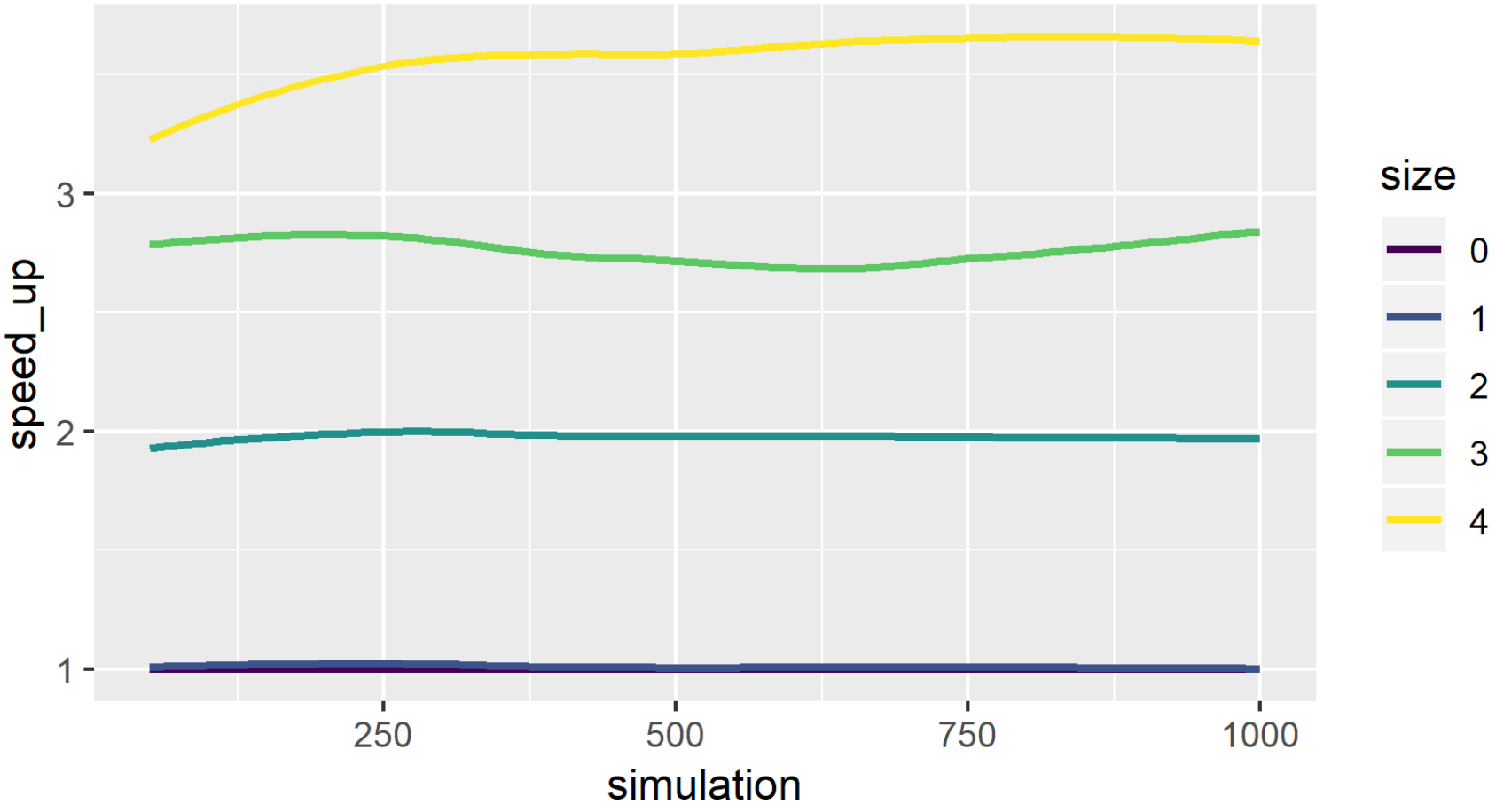


Figure 8: Plotting the time ratio between scalar-algorithm time and multiprocess-algorithm time at a fixed number of iterations

Final remarks

Possible further developments of the code:

- implement game-specific heuristics, in order to speedup and enhance the algorithm
- store convenient openings and closures (e.g. moves that automatically win the game)
- try other parallelization strategies (like root parallelization)
- devise a way to save the tree on file, in order to be be used for further matches
- extend the code to the case of multiple players (more than 2)
- test the algorithm on more complex games

References I


Guillaume M. J. B. Chaslot, Mark H. M. Winands, and H. Jaap van den Herik.
Parallel monte-carlo tree search.
In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *Computers and Games*, pages 60–71, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.


Nicola Gatti.
Economics and computation (lecture notes), 2018.


Levente Kocsis and Csaba Szepesvári.
Bandit based monte-carlo planning.
In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, pages 282–293, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

References II


Michael Maschler, Eilon Solan, and Shmuel Zamir.
Game Theory.
Cambridge University Press, 2013.


Stuart Russell and Peter Norvig.
Artificial Intelligence: A Modern Approach, Global Edition.
Addison Wesley, 2016.