



Toward a multilevel scalable parallel Zielonka's algorithm for solving parity games

Luisa D'Amore¹  | Aniello Murano¹ | Loredana Sorrentino¹ | Rossella Arcucci² |
Giuliano Laccetti¹ 

¹University of Naples Federico II, Naples, Italy

²Data Science Institute, Imperial College of
London, South Kensington Campus, London, UK

Correspondence

Luisa D'Amore, University of Naples Federico II,
Via Cintia, Naples, Italy.
Email: luisa.damore@unina.it

Summary

In this work, we perform the feasibility analysis of a multi-grained parallel version of the Zielonka Recursive (ZR) algorithm exploiting the coarse- and fine- grained concurrency. Coarse-grained parallelism relies on a suitable splitting of the problem, that is, a graph decomposition based on its Strongly Connected Components (SCC) or a splitting of the formula generating the game, while fine-grained parallelism is introduced inside the Attractor which is the most intensive computational kernel. This configuration is new and addressed for the first time in this article. Innovation goes from the introduction of properly defined metrics for the strong and weak scaling of the algorithm. These metrics conduct to an analysis of the values of these metrics for the fine grained algorithm, we can infer the expected performance of the multi-grained parallel algorithm running in a distributed and hybrid computing environment. Results confirm that while a fine-grained parallelism have a clear performance limitation, the performance gain we can expect to get by employing a multilevel parallelism is significant.

KEYWORDS

hybrid architectures, multilevel parallelism, parity games, performance analysis, Zielonka algorithm

1 | INTRODUCTION

Parity games are abstract infinite-duration games that represent a powerful mathematical framework to address fundamental questions in computer science and mathematics. In formal system design and verification,^{1–3} they arise as a natural evaluation machinery to automatically and exhaustively check for reliability of distributed and reactive systems. Parity games are also intimately related to other games of infinite duration such as *mean* and *discounted payoff*, *stochastic*, and *multiagent games*.^{4–16} Furthermore, a number of applications are placing great demands on large scale parity games and extreme computational resources are required to verify the desired properties. This is the case, for example, if we want to apply model checking algorithms to verification of specific properties of complex and large scale infrastructures in order to verify consistency of predictions of the system behavior with experimental data (such as the disease spread in complex epidemiological systems, the real-time reliability of the flight control system, safety-critical physical systems in astronomy).

The problem of finding a winning strategy in parity games is known to be in $\text{UPTime} \cap \text{CoUPTime}$.¹⁷ Deciding whether a polynomial-time solution exists is a long-standing open question. Through the years several algorithms to solve parity games have been proposed. Among the others, well known are the Zielonka recursive (ZR),¹⁸ the small progress measures,¹⁹ the strategy improvement,²⁰ the big step²¹ and, more recently, the quasi-polynomial²² ones. All of them have been implemented in PGSolver, a collection of tools (randomly) generate, solve, and benchmark parity games. The tool is written in OCaml by Oliver Friedman and Martin Lange.^{23,24} Along the years, several improvements on the existing parity

games algorithms and their implementations have been considered in order to speed-up their execution time. PGSolver itself is now equipped with some general preprocessing optimizations such as strongly connected component (SCC) decomposition, self-cycles removal, and priority compression.^{19,25} A direction that has been recently exploited concerns with the improvement of the implementation by using more suitable data structures and introducing performing programming languages. A contribution, which results in line with this approach, is an improved (OCaml) Implementation of the ZR algorithm (IOZR, for short) introduced in Reference 26. It makes use of a suitable data structure to handle with efficiently large game graphs along their evaluation. In Reference 26, it has been further exploited the use of *Scala* as programming language in place of OCaml. These two ingredients result in a gain of up to two orders of magnitude (one for each innovation) in running time compared to the classic OCaml implementation of the ZR algorithm (OZR, for short). As a side result, a similar gain can be obtained by using Java in place of Scala.²⁷ More recently, an efficient implementation in C++ of parity games solvers (including Zielonka's algorithm) is discussed in References 27 and 28. By comparing its performance with state-of-the-art solvers, the authors demonstrated that their software tool (named Oink) is competitive with other implementations and outperforms PGSolver for all algorithms, especially Zielonka's algorithm.

2 | RELATED WORKS AND CONTRIBUTION OF THE PRESENT WORK

In this section we introduce the contribution of the present work with respect to the state of the art of the scalable algorithms for parity games. With the term scalability, we refer to the capability of the algorithm to both:

- (i) exploit performance of the available computing architectures in order to minimize the time to solution for a given problem with a fixed dimension (*strong scaling*),
- (ii) use additional computational resources effectively to solve increasingly larger problems (*weak scaling*).

It is worth noting that weak scaling is much more challenging than strong scaling. In this regard, as claimed in Reference 28, Zielonka's algorithm does not weakly scale well. Indeed, scalability of a parity game algorithm is still a challenge because of the state space explosion. A scalable algorithm exploits the power of many computational resources in order to speed the execution time, as it is, instead of introducing simplifications or approximations to reduce the workload.

Several approaches have been considered to reduce the execution time of parity algorithms, but with little success in practice. The appropriate approach depends upon both the specification of the algorithm and the underlying architecture. Among the others, we recall the use of *symbolic algorithms*²⁹ that avoids the explicit construction of the game, *partial order reduction*³⁰ that can be used (on explicitly represented graphs) to reduce the number of independent interleaving of concurrent processes that need to be considered, and *abstraction-refinement*³¹ that attempts to prove properties on a system by first simplifying it and then by refining the simplification if necessary. Previous work on solving parity games in parallel involves the small progress measures algorithm³² or the Bellman-Ford algorithm. In particular, in Reference 33, a multi-core implementation of the small progress measures algorithm is presented. In Reference 34, GPU implementations of various algorithms for solving parity games are given. The authors implemented the strategy improvement algorithm also studied in Reference 35. They used the Bellman-Ford algorithm to compute best responses.

Recent activities of major chip manufacturers show more evidence than ever that future designs of microprocessors and large systems will be heterogeneous in nature, relying on the integration of two major types of components. On the first handle, multi/many-cores CPU technology have been developed and the number of cores will continue to escalate because of the need to pack more and more components on a chip. On the other hand, special purpose hardware and accelerators, especially graphics processing units, are in commodity production. Finally, reconfigurable architectures such as field programmable gate arrays offer several parameters such as operating frequency, precision, amount of memory, number of computation units, and so on. To cope with this scenario, it is becoming increasingly clear that we need to undertake the fundamental step of splitting the computation into (smaller) parts, that is, decomposing the computation. However, due to the multidimensional heterogeneity of modern architectures, not necessarily a decomposition leads to a parallel algorithm with the highest performance. A decomposition approach which exploits only a single type of parallelism have clear performance limitations (such as synchronization points, sequential parts, etc.), that prevent effective scaling with the thousands of processors available in massively parallel computers. This stating motivated the development of multilevel (or multigrained) parallel algorithms: the game is problem is decomposed into a set of sub-problems, or by employing SCC decomposition of the graph. Then, each one of these subproblems is solved by employing a fined grained concurrency and finally local solutions are suitably gathered. We underline that performance gain that we get from this approach is two fold: instead of solving one larger problem we can solve several smaller problems which leads to a reduction of each local algorithm's time complexity, the sub problems reproduce the problem at smaller dimensions and they are solved in parallel, which leads to a reduction of the global software execution time.³⁶⁻⁴⁰ Finally, the algorithm takes advantage of the fact that computation on separate independent SCC can be done in parallel.⁴¹

In this regards, as the present work is an improved version of Reference 27, its main contribution is

- The design of a multilevel ZR algorithm based on a domain decomposition approach. Namely, we aim to extend the approach described in Reference 27, where a one-level parallelism on multicore architectures was introduced.
- An improved performance analysis of the scalability of the algorithm. Namely, we completely revise the performance analysis carried out in Reference 27 where standard metrics of speed up and efficiency were adapted for multicore architectures in a quite unusual way. We derive a systematic analysis of the scalability of the local fine grained parallel algorithm. Furthermore, we use the values of the speed up of the local fine-grained parallel algorithm to derive the estimated scale up of the global multilevel parallel algorithm.

Outline of the work. The rest of this article is structured as follows. In Section 3, we provide some preliminary notion about parity games. We introduce the ZR algorithm, the Attractor as its main subroutine, and the parallel version of the improved variant of the Attractor introduced in Reference 26. In Section 4, we discuss the design of the multilevel parallel algorithm and in Section 5, we perform the scalability analysis of it. In particular, in Section 5.2, we present testing and experimental results on arenas of graphs up to 20K nodes. Finally, in Section 6, we give some conclusions and future work directions.⁴²

3 | PARITY GAMES

Two-player games played on directed graphs are an important framework for the synthesis of a controller for a reactive system in an uncontrollable environment.^{43–45} In this scenario, vertexes of the graph represent the states of the system and edges represent transitions/relations between those states. In this work, we consider two-player turn-based games in which each vertex either belongs to the system (i.e., the first player) or the environment (i.e., the second player). A game is played by moving an imaginary token from vertex to vertex according to transitions (i.e., the relations between the vertexes): the owner of a vertex decides where to move the token. The outcome of the game is an infinite sequence of vertexes called *play* that is obtained by the choices of moves of both players. The games that we consider are *zero-sum*: the first player tries to achieve an objective, while the second player tries to prevent this.

In the literature, for a long time, two-player games with ω -regular objectives have been studied. An effective way to represent games with ω -regular objectives is the class of *parity games*.² These are infinite-duration games played over finite-directed graphs that are partitioned into two disjoint set of vertexes associated with two players. The vertexes are labeled with natural numbers, called *priorities*. The two players move a token from one vertex to another (starting from a designed initial vertex) obtaining a *play*, that is, an infinite sequence of vertexes. The winner of the play is determined by the parity of the highest priority which occur infinitely often.

The motivation for studying parity games comes from the area of formal verification of systems by model checking. In fact the problem of solving parity games is polynomial-time equivalent to the non-emptiness problem of ω -automata on infinite trees with Rabin-chain acceptance conditions, and as well to the modal μ -calculus model checking problem.^{2,46,47} Another motivation to study this problem is that it is one of the few problems which belongs to the complexity class $\text{NPTIME} \cap \text{CoNPTIME}$ and more precisely to $\text{UPTime} \cap \text{CoUPTime}$ ¹⁸ and not yet known to belong to PTime , although, very recently, it has been proved that solving parity games is quasi-polynomial.²²

3.1 | Technical details

In this section, we present some basic concepts about parity games, which will be needed in the sequel. Most of them are standard definitions;⁴³ therefore, an expert reader can skip this part.

We start by giving and explaining the formal definition of parity games.

Definition 1 (Parity games). A parity game is a tuple $G = (V, V_0, V_1, E, \Omega)$ where

- (V, E) forms a directed graph whose set of vertexes is partitioned into $V = V_0 \cup V_1$, with $V_0 \cap V_1 = \emptyset$;
- $E \subseteq V \times V$ is a set of directed edges;
- $\Omega : V \rightarrow \mathbb{N}$ is the priority function that assigns to each vertex a natural number called the priority of the vertex.

For technical reasons, we assume that the edge relation E is total, that is, for every vertex $v \in V$, there is a vertex $w \in V$ such that $(v, w) \in E$ and $v \neq w$. In the following, we also write vEw in place of $(v, w) \in E$ and use $vE := \{w | vEw\}$.

Parity games are played between two players called Player 0 and Player 1. In a game graph, the vertexes in V_0 belong to the Player 0 and are graphically depicted as circles while the vertexes in V_1 belong to the Player 1 and are depicted as squares. Starting in a vertex $v \in V$, both players construct through the game graph an infinite path, that is the *play*, as follows. If the construction reaches, at a certain point, a finite sequence $v_0 \dots v_n$ and $v_n \in V_i$ then Player i selects a vertex $w \in v_n E$ and the play continues with the sequence $v_0 \dots v_n w$. Every play has a unique winner, defined by the maximum priority that occurs infinitely often along the play itself. Precisely, let $j, k \in \mathbb{N}$ be the indexes of positions, the winner of the play $v_0 v_1 v_2 \dots$ is Player i if and only if $\max \{p \mid \forall j. \exists k \geq j : \Omega(v_k) = p\} \bmod(2) = i$.

Let A be an arbitrary set of vertexes. The game $G \setminus A$ is the game restricted to the vertexes $V \setminus A$, that is, $G \setminus A = (V \setminus A, V_0 \setminus A, V_1 \setminus A, E \setminus (A \times V \cup V \times A), \Omega_{V \setminus A})$. It is worth observing that if A is total, then $G \setminus A$ is total as well. This property will be useful when we will consider A to be an Attractor set.⁴⁸

With each game, there is an associated notion of a strategy defined as follows.

Definition 2 (Strategy). A strategy for Player i is a partial function $\sigma_i : V^* V_i \rightarrow V$, such that, for all sequences $v_0 \dots v_n$ with $v_{j+1} \in v_j E, j = 0, \dots, n-1$, and $v_n \in V_i$ we have that $\sigma(v_0 \dots v_n) \in v_n E$.

A well-known property of parity games is that the strategy on a path only depends on the last vertex on that path. This means that the strategies are memoryless.

Definition 3 (Memoryless strategy). A strategy σ for Player i is called memoryless (or positional) if, for all $v_0 \dots v_n \in V^* V_i$ and for $w_0 \dots w_m \in V^* V_i$, we have that if $v_n = w_m$ then $\sigma(v_0 \dots v_n) = \sigma(w_0 \dots w_m)$.

In other words, the memoryless strategies do not consider the history of the play so far, but only the vertex of the play is currently in. We say that, a play $v_0 v_1 \dots$ conforms to a strategy σ for Player i if, for all j we have that, if $v_j \in V_i$ then $v_{j+1} = \sigma(v_0 \dots v_j)$.

Using strategies we extend the notion of winning to games.

Definition 4 (Winning strategy). A strategy σ for Player i is a winning strategy in vertex v of a game G if Player i wins every play starting in v that conforms to the strategy σ . In that case, we say that Player i wins the game G starting in v .

Example: WE consider a running example. It models the interaction between two players, Player 0 and Player 1. Observe that each vertex of the constructed arena is labeled with its name (in the upper part) $V = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ and, its priority (in the lower part) in according to the labeling function $\Omega(v_0) = 4, \Omega(v_1) = 0, \Omega(v_2) = 3, \Omega(v_3) = 1, \Omega(v_4) = 2, \Omega(v_5) = 6, \Omega(v_6) = 8$, and $\Omega(v_7) = 5$. The set of vertexes is partitioned as follows: circles vertexes $V_0 = \{v_0, v_2, v_5, v_7\}$ belong to Player 0 and squared vertexes $V_1 = \{v_1, v_3, v_4, v_6\}$ belong to Player 1. The transition relation is represented by the direction of edges between the vertexes. Furthermore, a possible play starting from the vertex v_0 is $(v_0 v_3 v_5)^\omega$. It is easy to see that it is also a winning play for Player 0 since she has always a possibility to visit infinitely often a vertex with the biggest even priority, trapping Player 1.

Starting from a game G we construct two sets $W_0, W_1 \subseteq V$ such that W_i is the set of all vertexes v from which Player i wins the game G (starting from it). Parity games enjoy determinacy, meaning that for every vertex v either $v \in W_0$ or $v \in W_1$.²

Solving a parity game means deciding, for a given starting position, which of the two players has a winning strategy. Formally, given a parity game, the sets W_0 and W_1 are computed, as well as the corresponding memoryless winning strategies σ_0 for Player 0 and σ_1 for Player 1 on their respective winning regions. The following theorem is proved in Reference 2.

Theorem 1. For each parity game G , there is a unique partition W_0, W_1 of V such that there is a winning strategy for Player 0 (resp, Player 1) from W_0 (resp, W_1).

The construction procedure of winning regions makes use of the notion of *Attractor*. Let $U \subseteq V$ and $i \in \{0, 1\}$. We define the Attractor for Player i , namely the i -Attractor, of U as the least set W s.t. $U \subseteq W$ and whenever $v \in V_i$ and $vE \cap W \neq \emptyset$, or $v \in V_{1-i}$ and $vE \subseteq W$ then $v \in W$. Hence, the i -Attractor of U contains all vertexes from which Player i can move “toward” U and Player $1-i$ must move “toward” U . The i -Attractor of U is denoted by $\text{Attr}_i(G, U)$.

Definition 5 (Attractors). Given a parity game G where U is a set of vertexes of G . The Attractor set $\text{Attr}_i(G, U)$ for Player i is defined inductively as follows:

- $\text{Attr}_i(G, U)^0 = U$,
- $\text{Attr}_i(G, U)^{k+1} = \text{Attr}_i(G, U)^k \cup \{v \in V_i \mid \exists (v, w) \in E : w \in \text{Attr}_i(G, U)^k\} \cup \{v \in V_{1-i} \mid \forall (v, w) \in E : w \in \text{Attr}_i(G, U)^k\}$,
- $\text{Attr}_i(G, U) = \bigcup_{k=0}^{\infty} \text{Attr}_i(G, U)^k$.

Intuitively, the set $\text{Attr}_i(G, U)$ is the largest set of vertexes from which the Player i with $i \in \{0, 1\}$ can attract the token from any vertex in $\text{Attr}_i(G, U)$ to U in a finite number of steps. In other words, $\text{Attr}_i(G, U)^k$ consists of all vertexes such that Player i with $i \in \{0, 1\}$ can force any play to reach U in at most k moves.

3.2 | Classic Zielonka recursive algorithm

In this subsection, we describe the ZR algorithm using the basic concepts introduced above and some observations regarding its implementation in PGSolver. The algorithm to solve parity games introduced by Zielonka comes from a work of McNaughton.⁴⁹ The main difference is that McNaughton limits himself to finite graphs, while Zielonka concerns himself with infinite graphs as well. Because of their fundamental similarity, the ZR algorithm is sometimes referred to as the McNaughton's algorithm in the literature on parity games. These algorithms are functionally identical. The ZR algorithm, as reported in Algorithm 1, uses a divide and conquer technique. It requires to decompose the graph game into multiple smaller arenas, which is done by computing, in every recursive call, the *difference* between the current graph and a given set of vertexes. Notice that latter operation (Algorithm 1, lines 10 and 16) is quite expensive as it requires to generate a new graph at each iteration. Finally, it constructs the winning sets for both players using the solution of subgames. The algorithm removes the vertexes with the highest priority from the game, together with all vertexes (and edges) attracted to this set. The ZR algorithm starts by invoking the routine $\text{win}(G)$, which takes a graph G as input and, after a number of recursive calls over ad hoc built subgames, returns the winning sets (W_0, W_1) for Player 0 and Player 1, respectively. In the base case, when G is empty, the winning regions (\emptyset, \emptyset) can be obtained immediately. In a parity game, the higher priorities dominate all lower ones. Let d be the highest priority appearing in G , $p = d \bmod 2$ be the index of the first player, and $j = 1 - p$ be the index of the adversarial player. Let U be the set of all vertexes with priority d in G . Then, the p -Attractor A of the set U is calculated and removed, so the subgame $G \setminus A$ is recursively solved. When $G \setminus A$ is completely won by Player p , then the whole game is also won by Player p : whenever Player j decides to visit A , Player p will enforce a visit to U , then every play which visits A infinitely often has d as the highest priority occurring infinitely often, which means the play is won by Player p ; otherwise, Player j will stay in $G \setminus A$, which is a winning region for Player p . When $G \setminus A$ is not completely won by Player p , the j -Attractor B which is a winning region for Player j in G is computed. subsequently, the subgame $G \setminus B$ is recursively solved and the winning regions for Player p and Player j are acquired, respectively. It is important to note that Player p indeed wins on W_0' because Player j is neither able to force Player p into W_j' nor B . On the other hand, Player j wins on $B \cup W_j'$. Therefore, W_0' and $B \cup W_j'$ are returned as the winning regions for the whole game.

Algorithm 1. Classic Zielonka recursive algorithm

```

1  function win(G):
2      if V == ∅:
3          (W0, W1) = (∅, ∅)
4      else:
5          d = maximal priority in G
6          U = {v ∈ V | priority(v) = d}
7          p = d % 2
8          j = 1 - p
9          A = Attrp(G, U)
10         (W'0, W'1) = win(G \ A)
11         if W'j == ∅:
12             Wp = W'p ∪ A
13             Wj = ∅
14         else:
15             B = Attrj(G, W'j)
16             (W'0, W'1) = win(G \ B)
17             Wp = W'p
18             Wj = W'j ∪ B
19     return (W0, W1)

```

Example: By using the same game arena, we show how to solve the game by applying Algorithm 1. It is easy to note that the vertex v_6 has the biggest priority; so $d = 8$ and $U = \{v_6\}$. Then, first it is calculated $\text{Attr}_0(G, \{v_6\}) = \{v_5, v_6, v_7\}$. At this point $G \setminus \text{Attr}_0(G, \{v_6\})$ is recursively solved. To do so, the 0-Attractor of v_0 is calculated:

$$\text{Attr}_0(G \setminus \text{Attr}_0(G, \{v_6\}), \{v_0\}) = \{v_0, v_3\},$$

and

$$G \setminus \text{Attr}_0(G \setminus \text{Attr}_0(G, \{v_6\}), \{v_0\}),$$

is recursively solved. Now, the 1-Attractor of v_2 is calculated:

$$\text{Attr}_1((G \setminus \text{Attr}_0(G, \{v_6\}), \{v_0\}), v_2) = \{v_1, v_2, v_4\}.$$

Later, the whole game G is handled, by calculating first the set B containing the 1-Attractor to $\{v_1, v_2, v_4\}$ that returns the set of vertexes $\{v_1, v_2, v_4, v_6, v_7\}$, which are winning for Player 1. After that, the game $G \setminus \{v_1, v_2, v_4, v_6, v_7\}$ is solved. Finally the set

$$W_0 = \{v_0, v_3, v_5\},$$

and

$$W_1 = \{v_1, v_2, v_4, v_6, v_7\}$$

are returned.

3.3 | Improved Zielonka recursive algorithm

The ZR algorithm makes use of a *divide and conquer* technique in order to construct the winning sets for both players by using subgames that remove the vertexes with the highest priority from the game and the ones “attracted.” For this reason, the corresponding subroutine that takes care of this is called *Attractor*. In this section, we consider the Attractor described in Algorithm 2, which is an improved version of the classical one.²⁶¹

Algorithm 2. Zielonka improved Attractor

```

1  function Attr (G, T, Removed, A, i):
2      tmpMap = []
3      for x = 0 to |V|:
4          if x ∈ A : tmpMap[x] = 0
5          else : tmpMap[x] = -1
6      index = 0
7      while index < |A|:
8          for v0 ∈ adj(T, A[index]):
9              if v0 ∉ Removed :
10                 if tmpMap[v0] == -1 :
11                     if player(v0) == i :
12                         A = A ∪ {v0}
13                         tmpMap[v0] = 0
14                 else :
15                     adj_counter = -1
16                     for x ∈ adj(G, v0) :
17                         if x ∉ Removed:
18                             adj_counter += 1
19                     tmpMap[v0] = adj_counter
20                     if adj_counter == 0 :
21                         A = A ∪ {v0}
22                 else if (player(v0) == j and
23                        tmpMap[v0] > 0):
24                     tmpMap[v0] = 1
25                 if tmpMap[v0] == 0:
26                     A = A ∪ {v0}
27             index += 1
28      return A

```

The improved Attractor function uses a HashMap, called tempMap, to keep track of the number of successors for the opponent player's vertexes. It is capable of checking if a given vertex is excluded or not in constant time by adding constant information to the states, that is, a flag (removed, -removed) that has been proved to be very successful in practice. It works without the need to directly modify the graph data structure, but rather it uses a set to keep track of removed vertexes. The improved implementation of the Attractor algorithm, named Attr, takes the following parameters:

¹As an improvement, the *Attractor* introduced in Reference 26 is able to check if a given vertex is excluded or not in constant time. Moreover, it completely removes the need for a new graph in every recursive call. This is done in the calling recursive algorithm in Reference 26, by adding the vertex in a set called *Removed* that is given as input to the *Attractor*, so it can work only on the vertexes present in the evaluated subgraph.

the Graph, its transposed one, a set of excluded vertexes, the starting attracting set, and the corresponding player. It returns the set A containing all the attracted nodes.

Example: The Attractor would be first feed by the ZR algorithm with a set A containing all the vertexes sharing the highest priority. In this case, it only concerns the vertex v_6 having 8 as priority. Then, for all vertexes in A , it would try to attract all the incident vertexes, in this case the vertex v_5 and v_7 and inserted in A , from which it tries to Attract their incident vertexes as well.

In this case, they are automatically attracted because they belong to Player 0 since the priority of the vertex v_6 is even, but it is not able to attract the vertex v_4 , which is incident in the vertex v_7 , because its player could flee to v_1 . The same is true for vertex v_3 that could flee to v_0 .

If using `tmpMap`, we would see for v_4 the initialized value -1 being incremented up to 1 , which makes the vertex not eligible to be attracted, unless other attracted vertexes would be able to exclude its escape routes (v_1), decrementing the value in `tmpMap` so that it becomes 0 . In this case, the algorithm would have attracted only the vertexes v_6 and v_7 , returning the set A containing three vertexes. Subsequent calls on subgraphs would still work on the same graph, but they would just ignore external vertexes, thanks to the `Removed` set, allowing it to not actually build a subgraph. Notably, each starting or newly added vertex in A could work on their own on `tmpMap` trying to attract other vertexes, in parallel.

4 | TOWARD MULTILEVEL PARALLEL FRAMEWORK FOR ZIELONKA ALGORITHM

A common issue of any *divide and conquer*-based algorithm is that, for a given fixed size, the overhead increases as the number of processors involved increases while it decreases as the processors number is fixed and the size of the problem increases. Since we aim to design a scalable ZR algorithm, which is able both to minimize the time to solution for a given problem with a fixed dimension (strong scalability) and to use additional processors to solve increasingly larger problem (weak scaling), the best mapping can be obtained designing a multilevel algorithm which ensures that the number of subproblems generated at the coarsest level will be small (coping with the weak scalability) and at the same time, by distributing further subdomains between processors within each node, it allows the introduction of the parallelism at the finest level of granularity (meeting the strong scalability). A multilevel solution technique borrows ideas to domain decomposition (1-level DD), introducing more levels of decomposition.⁵⁰ Therefore, instead of increasing the number of subdomains to be assigned to each processor, the domain decomposition follows a tree configuration grafting a level of decomposition in the previous. This ensures to keep small the number of subdomains generated at level 1 (coping with the scalability of local games) and at the same time, distributing further subdomains between processors within each node, we introduce levels of parallelism in a hierarchical manner (meeting the scalability of the whole parallel algorithm).

In this regards, the key point of the present work is to formulate and prove the necessary performance results that underpin this parallel framework designed according to the features of the application and of the computing environment. Hence, results we present are concerned with the capability to solve this problem by using such a framework, hoping that these results encourage readers to further extend the parallel framework according to their specific application's requirements. For instance, we could consider the possibility of implementing the strategy described in Reference 51.

4.1 | The multilevel algorithm

As the graph to be decomposed is divided into independent (SCC-closed) subgraphs, which are further divided into smaller independent subgraphs until they become SCCs, the multilevel algorithm consists of several copies of local fine-grained parallel algorithms.^{41,52} Then, here we address the implementation of the fine-grained parallel ZR algorithm which exploits concurrency inside the Attractor module. Algorithm 3 describes the Zielonka parallel Attractor. Since the testbed computing environment we consider for implementing the fine-grained parallelism is an Hyper-Threading architecture which can only process the data in its global memory, to avoid data transfer and in order to reduce the overhead, it was decided to store the device with the entire work data prior to any processing. Hence Algorithm 3 takes as input the graph G , its transposed graph T , the `Removed` set containing all the nodes filtered out, the initial set A and i , the attracting player. It returns the set A containing all the attracted nodes exactly as in Algorithm 2. The input of the resulting algorithm is split into independent chunks which are processed by the parallel tasks in a completely parallel manner.

A new function `asyncAttr` is introduced in Algorithm 3 as its parallel task which, besides the same Graph G , Transposed T , Set `Removed` and player i , takes as input a single node of the set A , namely $A[index]$, and returns a set containing all the nodes attracted from that single one. For each attracted node in A , we assign the task to the first available processor, that would then take care of its edges in parallel (see vertical arrows in Figure 1) so that their result can be combined into the set returned by the Attractor, until no more nodes are attracted.

Algorithm 3. Zielonka par allel Attractor

```

1  function ParAttr (G, T, Removed, A, i ):
2      tmpMap = []
3      for x = 0 to |V|:
4          if x ∈ A : tmpMap[x] = 1
5          else : tmpMap[x] = ∞
6      index = 0
7      while index > |A|:
8          p = numproc(|A| - (index + 1), pmax)
9          for idproc ∈ (0, p) :
10             Aidproc = Aidproc ∪ asyncAttr(G, T,
11                 A[index], Removed, i)
12             A = A ∪ Aidproc
13             index+ = 1
14      return A
15      function numproc(indexLeft, pmax):
16          if indexLeft > pmax :
17              p = indexLeft
18          else :
19              p = pmax
20          return p-1
21
22  function asyncAttr (G, T, index, Removed, i ):
23      A = { }
24      for v0 ∈ adj(T, index):
25          if v0 ∉ Removed :
26              if test(tmpMap[v0] == ∞) and
27                  set(tmpMap[v0] = 0):
28                  if player(v0) == i :
29                      A = A ∪ v0
30                      tmpMap[v0] = 1
31              else: adj_counter = -1
32                  for x ∈ adj(G, v0) :
33                      if x ∉ Removed:
34                          adj_counter+ = 1
35                      tmpMap[v0] += adj_counter
36                      if adj_counter == 1 :
37                          A = A ∪ {v0}
38              else if player(v0) == j :
39                  tmpMap[v0]- = 1
40                  if tmpMap[v0] == 1:
41                      A = A ∪ {v0}
42      return A

```

to avoid concurrency. Using an atomic increments/decrements is enough to address the issue when updating the *tempMap* counter on lines 35 and 39. Regarding *tempMap*, we introduce a new notation described in Table 1. The negative numbers no longer have no meaning to indicate uninitialized nodes. All numbers minor or equal to zero tell us how many edges were excluded before counting, and the value keeps going down until that happens. After counting, each value is never below 1. Nodes with the value set to 1 can be successfully attracted. To indicate uninitialized nodes, the first irrelevant number can be used, namely the maximum number of edges plus one, or infinite, as we indicated at lines 5 and 26 in Algorithm 3.

Example: Figures 1 and 2 show a snapshot of an execution of the Attractor algorithm: Player 0's (Even) nodes are represented by circles while squares represent Player 1's (Odd) nodes. The numbers indicated inside represent the priorities. On the right, there is a depiction of a directed graph representing a parity game where the blue and the red regions are Player 0's and Player 1's winning region, respectively, while the blue and red arrows are Player 0's and Player 1's winning strategies. On the left, the snapshot of execution is represented on an adjacency matrix, particularly, the adjacent edges (horizontal blue arrow) of incoming nodes of "8" (vertical blue arrow) are being examined by the Attractor, as the one from node "5" is. The attracted node are marked by "A," notably, node "5" is being attracted not having any edge going to a non attracted node. In colored nodes, the winning region belonging is known. The values of Improved's *tempMap* are also reported, with 0 marking attracted nodes, -1 unreachable, and 1 non-attracted. As indicated by the vertical arrows, each node is processed in parallel by the Algorithm 3.

5 | SCALABILITY METRICS OF THE MULTILEVEL PARALLEL ALGORITHM

In this section, we introduce the metrics we shall use for evaluating the performance of the multilevel parallel algorithm. We observe that since the configurations of the parallel algorithm, which we have considered in our experiments, can be regarded as a part of the multilevel parallel algorithm, from the analysis of the values of these metrics, we can infer the expected performance of the multilevel parallel algorithm. It is worth noting that the performance analysis we carried out improves that one presented in Reference 27 for multicore architectures. In a multicore computing environment, standard metrics such as speed up and efficiency are not suitable to measure the performance gain or to identify scalability bottlenecks of the parallel algorithm.⁵³ In addition, we do not find any consistent meaning of the weak scaling factor and its measured value, introduced in Reference 27. Instead, following Reference 54, here we develop a systematic performance analysis including the estimate of the expected performance gain of the whole algorithm which is based on the local speed up of its multithreaded computational kernels.

The simplest of these metrics is the elapsed time taken to solve a given problem on a given parallel platform. However, this measure suffers from some drawbacks. Namely, the execution time of a parallel algorithm depends not only on input size but also on the number of processing elements used, and their relative computation and interprocess communication speeds. Furthermore, many factors contribute to the scalability, including the architecture of the parallel computer and the parallel implementation of the algorithm. For these reasons, we focus on metrics for quantifying the performance and the scalability of the algorithm itself.

If $nproc$ denotes the total number of available processing elements, we assume that

$$nproc = p \times q, \quad (1)$$

where q denotes the number of processing elements involved for implementing the coarse-grained parallelism (i.e., the first-level parallelism) and p denotes the number of processing elements involved for implementing the fine-grained parallelism (at the second level). We consider a uniform decomposition, such that the parity games G defined on n nodes was properly decomposed in q parity games each one defined on $\frac{n}{q}$ nodes. We assume that the graph is decomposed in independent subgraphs by means of a SCC decomposition or its generation is decomposed into many sub games.⁴¹

Definition 6. Let $A(G)$ be the ZR algorithm solving the parity game G on n nodes and $\mathcal{A}(G_i)$ is the local algorithm solving the local parity game on $\frac{n}{q}$ nodes. ♣

Definition 7. The multilevel parallel algorithm which solves the parity game G by concurrently solving the local parity games is denoted as

$$\mathcal{A}_q(G) := \{\mathcal{A}(G_1), \mathcal{A}(G_2), \dots, \mathcal{A}(G_q)\}. \quad (2)$$

In order to estimate the scalability of the multilevel parallel algorithm $\mathcal{A}_q(G)$, at the first level of parallelism, as in Reference 54, we use the following performance metric.³⁷

Definition 8 [The multilevel scale up factor]. $\forall i \neq j$ we define the (relative) scale up factor of $\mathcal{A}_q(G)$, in going from 1 to q , the following ratio:⁵⁴

$$Sc_{q,1}^f(\mathcal{A}_q(G)) = \frac{T(n)}{q \cdot T(n/q)}, \quad (3)$$

where $T(\cdot)$ denotes the time complexity of the algorithm. ♣

The scale up factor quantifies the benefit of the decomposition, that is, how much the time complexity of the algorithm $\mathcal{A}(G)$ may be reduced with respect to the parallel algorithm $\mathcal{A}_q(G)$.

Let $SW(\mathcal{A}(G), nproc)$ denote the software implementing $\mathcal{A}(G)$ on a fixed processing architecture made of $nproc$ processing elements. We recall the following

Definition 9 (Floating point Execution time). Let t_{flop} denote the unitary time required for the execution in Processing Element (PE) of one operation. The execution time needed to $SW(\mathcal{A}(G), nproc)$ for performing $T(n)$ operations, is

$$T_{flop}(n) = T(n) \times t_{flop}. \quad (4)$$

In addition to performing essential computation (i.e., computation that would be performed by the serial program for solving the same problem instance), a parallel software may also spend time in interprocess communication, idling, and excess computation (computation not performed by the serial formulation). We will denote this time as the software elapsed execution time.

Definition 10 (Software elapsed execution time).

$$T^{nproc}(n) := T_{flop}^{nproc}(n) + T_{oh}^{nproc}(n) \quad (5)$$

denotes the elapsed execution time of $SW(\mathcal{A}(G))$ given by time for computation plus an overhead which is given by synchronization, memory accesses and communication time also (see Reference 55 for details).

♣

Clearly, from Equations (4) and (5) if $nproc = 1$, it follows that

$$T_{flop}(n) = T_{flop}^1(n).$$

According to Reference 54, we use the following metric quantifying the performance gain of the multilevel parallel algorithm in terms of execution time.

Definition 11 [Multilevel scale-up]. Let

$$Sc_{1,q}^{ML}(\mathcal{A}_q(G)) = \frac{T_{flop}(n)}{q \cdot (T_{flop}(n/q) + T_{oh}(n/q))}, \quad (6)$$

be the multilevel scale-up in going from 1 to q .

♣

By using Equation (5), we could express the multilevel scale up factor in terms of the local speed up. To this end, we need to introduce the following quantities.

Definition 12 [Local speed up]. The ratio:

$$s_p^{loc}(\mathcal{A}(G_i)) = \frac{T_{flop}(n/q)}{T^p(n/q)} \in [1, p]$$

denotes the speed up of the (local) algorithm $\mathcal{A}(G_i)$ running on p processing elements.

♣

Definition 13 [Local surface-to-volume ratio]. Let $\frac{S}{V}^{p,q}$ denote the so-called surface-to-volume ratio. It is a measure of the amount of data exchange (proportional to surface area of domain) per unit operation (proportional to volume of domain). For $\mathcal{A}(G_i)$ this is

$$\frac{S}{V}^{p,q}(\mathcal{A}(G_i)) = \frac{T_{oh}(n/q)}{T_{flop}(n/q)}. \quad (7)$$

♣

We note that these metrics are quite different from those introduced in Reference 27, as they refer to solution of subproblems of size n/q using p processing elements.

Finally, we prove the following results relating the multilevel scale-up as given in Equation (6) to the local speed up.

Proposition 1. If

$$0 \leq \frac{S}{V}^{p,q} < 1 - \frac{1}{s_p^{loc}},$$

then, it holds that

$$Sc_{1,q}^{ML}(\mathcal{A}_q(G)) = \alpha Sc_{1,q}^f(\mathcal{A}_q(G)), \quad (8)$$

with

$$\alpha := \frac{s_p^{loc}}{1 + s_p^{loc} \frac{S}{V}}. \quad (9)$$

Proof.

$$Sc_{1,q}^{ML}(\mathcal{A}_q(G)) = \frac{T_{flop}(n)}{\frac{q T_{flop}(n/q)}{s_p^{loc}} + q T_{oh}(n/q)} = \frac{s_p^{loc} \frac{T_{flop}(n)}{q T_{flop}(n/q)}}{1 + \frac{s_p^{loc} T_{oh}(n/q)}{T_{flop}(n/q)}}. \quad (10)$$

If

$$\alpha := \frac{s_p^{loc}}{1 + \frac{s_p^{loc} T_{oh}(n/q)}{T_{flop}(n/q)}} = \frac{s_p^{loc}}{1 + s_p^{loc} \frac{S}{V}},$$

from Equation (10) it comes Equation (8).

♣

Finally, last proposition allows us to examine the benefit on the multilevel scale up arising from the speed up of the local parallel algorithm, mainly in the presence of a multilevel decomposition, where $s_p^{loc} > 1$.

Proposition 2. *Under the same hypothesis of Proposition 2, it holds that*

$$s_p^{loc} \in [1, q] \Rightarrow Sc_q^{ML} \in]Sc_{1,q}^f, q Sc_{1,q}^q[.$$

Proof.

- if $s_p^{loc} = 1$ then

$$\alpha < 1 \Leftrightarrow Sc_{1,q}^{ML} < Sc_{1,q}^f$$

- if $s_p^{loc} > 1$ then

$$\alpha > 1 \Leftrightarrow Sc_{1,q}^{ML} > Sc_{1,q}^f;$$

- if $s_p^{loc} = p$ then

$$1 < \alpha < q \Rightarrow Sc_{1,q}^{ML} < q \cdot Sc_{1,q}^f;$$

■

♣

This proposition allows us to quantify the upper bound of the scale-up for the multilevel algorithm. Indeed, we let $Id[Sc_{1,q}^f] := q \cdot Sc_{1,q}^f$ to be the ideal scale-up of the multilevel parallel algorithm.

The next result allows to further detail the behavior of the scale-up factor.³⁷

Proposition 3. *Let*

$$T(n) = a_d n^d + a_{d-1} n^{d-1} + \dots + a_0, \quad a_d \neq 0$$

be a polynomial of degree $d > 1$, that is, then scale-up factor is

$$Sc_{1,q}^f(n) = \frac{T(n)}{q \cdot T(n/q)} = \alpha(n, q) q^{d-1}, \quad (11)$$

where

$$\alpha(n, q) = \frac{a_d + a_{d-1} \frac{1}{n} + \dots + \frac{a_0}{n^d}}{a_d + a_{d-1} \frac{p}{n} + \dots + \frac{a_0 p^d}{N n^d}},$$

and

$$\lim_{q \rightarrow n} \alpha(n, q) = \beta \in]0, 1].$$

♣

Corollary 1. *Under the same hypothesis of Proposition 4, if $a_i = 0 \forall i \in [0, d-1]$, then $\beta = 1$, that is,*

$$\lim_{q \rightarrow n} \alpha(n, q) = 1.$$

Finally

$$\lim_{n \rightarrow \infty} \alpha(n, q) = 1.$$

♣

Corollary 2. *If n is fixed, it is*

$$\lim_{q \rightarrow n} Sc_{1,q}^f(\mathcal{A}_q(G)) = \beta \cdot n^{d-1};$$

while, if q is fixed

$$\lim_{n \rightarrow \infty} Sc_{1,q}^f(\mathcal{A}_q(G)) = \text{const} \neq 0.$$

♣

In conclusion, the scale-up factor tends to a fixed value

1. if n is fixed and q increases toward n ;
2. if q is fixed and n increases.

As expected, for a fixed size application or a fixed number of sub problems, the algorithm has a poor (strong) scalability. The reason is that it needs to find the appropriate value of the number of sub graphs, q , giving the right trade off between the scale-up and the overhead of the algorithm. On the other hand, that is, for large scale problems, the algorithm has a good (weak) scalability when both n and q grows up (see Equation (11)).

5.1 | Performance analysis

In this section, we introduce the parameters we will use to evaluate performance of the fine-grained parallel algorithm, hence to estimate the performance of the multilevel parallel algorithm as described in the previous section (Table 2). Table 3 collects values of the performance metrics for both the coarse- and the fine-grained parallelism.

Let p denote the number of threads (see line 15 in Algorithm 3), q denote the number of physical cores. Let G be the game graph with n nodes or states, e edges and d priorities. By approaching the improved Attractor reported in Algorithm 2, it is easy to notice that there are 2 nested loops: the external one that cycles over all attracting and attracted nodes, while the internal one cycles on its edges. The execution time of the ZR Attractor in Algorithm 3 is denoted as (Table 4)

$$T^{p,q}(n), p = 1, \dots, pmax; q = 1, \dots, qmax; p \geq q.$$

It holds the following result:

Theorem 2 (18,26). *The running time complexity of the algorithm is exponential in the number of priorities, and polynomial in the number of edges and vertices.*

- The running time complexity of the improved Attractor is

$$T_A(n) = O(e \cdot n), \quad (12)$$

that, in the worst case, that is, when the adjacent list of node contains $n - 1$ nodes, becomes $O(n^2)$. In this algorithm, it is easy to see that while all reached nodes and edges are processed, the data is stored through two means: tempMap, used at lines 10, 13, 19, 23, and 24 to keep track of the number of successor for he opponent player's nodes, and A, at lines 7, 12, 21, and 26, the set that will contain all attracted nodes.

TABLE 2 Percentage of the Attractor execution time with respect the total execution time of Algorithm 2

Game Size	6000	8000	12000	16000	20000
Tot (secs)	0.215	0.959	1.917	3.614	5.229
Attr (secs)	0.214	0.958	1.911	3.612	5.227
%	99.7%	99.8%	99.8%	99.9%	99.9%

TABLE 3 Performance of the fine- grained algorithm and of the multigrained parallel algorithm

p	1	2	3	4	5	6	7	8
q	1	2	3	4	4	4	4	4
$S_{HT}^{p,q}$	1	2	3	4	4.3	4.6	4.9	5.2
$Sc_{1,q}^f$	1	4	9	16	16	16	16	16
$Id[Sc_{1,q}^{ML}]$	1	8	27	64	80	96	112	128
$Sc_{1,q}^{ML}$	1	8	27	64	68.8	73.6	78.4	83.2

Note: First row: values of $S_{HT}^{p,q}$ for values of Attr provided in Table 2 and where $r = 30\%$. Last three rows are the values of $Sc_{1,q}^f$, $Sc_{1,q}^{ML}$ and $Id[Sc_{1,q}^{ML}]$ which are obtained by using Propositions 2 and 3, where parameter α is assumed to be equal to s_p^{loc} . Note that while $S_{HT}^{p,q}$ is the performance metric to be used at the second level of parallelism (the fine grained parallelism), last three metrics measure the expected performance of the multigrained parallel algorithm.

TABLE 4 Execution times of the fine-grained parallel algorithm

n	$T^{1,1}(n)$	$T^{2,2}(n)$	$T^{3,3}(n)$	$T^{4,4}(n)$
Nodes	secs	secs	secs	secs
4000	0.251	0.154	0.112	0.093
6000	0.536	0.322	0.232	0.191
8000	1.013	0.612	0.435	0.354
10000	1.789	1.067	0.750	0.606
12000	2.011	1.198	0.852	0.693
14000	2.807	1.689	1.206	0.961
16000	3.787	2.239	1.609	1.312
18000	4.379	2.549	1.836	1.483
20000	5.420	3.231	2.286	1.837
n	$T^{5,4}(n)$	$T^{6,4}(n)$	$T^{7,4}(n)$	$T^{8,4}(n)$
Nodes	secs	secs	secs	secs
6000	0.167	0.157	0.151	0.154
8000	0.304	0.275	0.261	0.263
10000	0.521	0.475	0.445	0.433
12000	0.602	0.549	0.518	0.503
14000	0.840	0.774	0.726	0.701
16000	1.138	1.058	0.990	0.951
18000	1.299	1.194	1.128	1.103
20000	1.602	1.506	1.427	1.398

- The time complexity of the Improved Recursive Zielonka Algorithm(T_{IRZA}) is

$$T_{IRZA}(n) = O(e \cdot n^d). \quad (13)$$

- According to the data decomposing we consider in the fine-grained parallelism, from Equation (13), we get

$$T^{p,q}(n) \propto e \left(\frac{n}{q} \right)^d t_{\text{flop}},$$

where t_{flop} is the unitary time.

- We compute the surface-to-volume ratio $\frac{S}{V}^{p,q}(n/q)$. For the fine-grained parallel algorithm, we may assume

$$\frac{S}{V}^{p,q}(n/q) = Oh^{p,q}(n/q) \propto \frac{T_{\text{mem}}^{p,q}(n/(p \cdot q))}{T^{p,q}(n/(p \cdot q))}, \quad (14)$$

where $T_{\text{mem}}^{p,q}$ denotes the time for synchronization and memory accesses. As

$$T_{\text{mem}}^{p,q}(n/q) \propto \frac{n}{p \cdot q} \alpha t_{\text{flop}},$$

where $\alpha > 1$ is a parameter depending on the architecture, it follows that

$$Oh^{p,q}(n) \propto \frac{\frac{n}{q}}{e \left(\frac{n}{p \cdot q} \right)^d} = \alpha \frac{1}{e} \left(\frac{n}{p \cdot q} \right)^{1-d}. \quad (15)$$

In Figure 5, we plot

$$L_{Oh}(q, n) = \frac{1}{e} \left(\frac{n}{q} \right)^{1-d}, \quad (16)$$

representing the estimate of the surface-to-volume value as given in (15) for $p = 1$. We observe that as n increases the surface-to-volume decreases and it may be assumed negligible. This means that in Equation (8), we may let

$$\alpha \approx s_p^{loc}.$$

- the *ideal speed up*, according with its classical definition, equals the number of physical execution units involved which should be cores in this case.⁵⁶ Here we denote with $qmax$ the maximum number of physical cores and with $pmax$ the maximum number of *virtual* cores. We observe that for this architecture, $pmax = 2 qmax$. In order to estimate the *ideal speed up* when using multiple Hyper-Threading processors, in contrast to Reference 27, we recur to the Amdahl's law described in Reference 57

$$S_{HT}^{p,q} = \frac{1}{1 - Attr + \frac{Attr}{q'}}, \quad (17)$$

where $Attr$ is the portion of execution time that the Attractor originally occupied with respect the Zielonka algorithm and where

$$q' = q + r(p - q), \quad (18)$$

is the speed up given by the Hyper-Threading architecture, where r is the percentage of improvement given by the doubling of some registers. Hyper-Threading Technology allows one physical processor package to be perceived as two separate logical processors within the operating system. However, Hyper-Threading Technology cannot have performance expectations equivalent to that of multiprocessing where all the processor resources are replicated.⁵⁸ Measured performance on common server application benchmarks for this technology shows performance gains of up to $r = 30\%$. Observe that when $p = q = q'$, Equation (17) provide the classical definition of speed-up as defined in References 56 and 57. It is worth noting that q' also provide the number of logical computing elements in Hyper-Threading Technology (see Tables 5 and 6 for results).

TABLE 5 Speed-up values of the fine-grained algorithm, compared with respect to $S_{HT}^{p,q}$

n Nodes	$S^{1,1}(n)$	$S^{2,2}(n)$	$S^{3,3}(n)$	$S^{4,4}(n)$
$S_{HT}^{p,q}$	1.0	2.0	3.0	4.0
6000	1.0000	1.6646	2.3103	2.8063
8000	1.0000	1.6552	2.3287	2.8616
10000	1.0000	1.6786	2.3603	2.9019
12000	1.0000	1.6619	2.3275	2.9209
14000	1.0000	1.6767	2.3853	2.9521
16000	1.0000	1.6914	2.3536	2.8864
18000	1.0000	1.7179	2.3851	2.9528
20000	1.0000	1.6775	2.3710	2.9505
n Nodes	$S^{5,4}(n)$	$S^{6,4}(n)$	$S^{7,4}(n)$	$S^{8,4}(n)$
$S_{HT}^{p,q}$	4.3	4.6	4.9	5.2
6000	3.2096	3.4140	3.5497	3.4805
8000	3.3322	3.6836	3.8812	3.8517
10000	3.3405	3.6630	3.8822	3.9980
12000	3.3417	3.6266	3.8664	4.0043
14000	3.4338	3.7663	4.0202	4.1316
16000	3.3278	3.5794	3.8253	3.9821
18000	3.3711	3.6675	3.8821	3.9701
20000	3.3833	3.5989	3.7982	3.8770

TABLE 6 Efficiency as in Equation (19)

n	$E^{1,1}(n)$	$E^{2,2}(n)$	$E^{3,3}(n)$	$E^{4,4}(n)$
Nodes				
6000	1.00	0.83	0.77	0.70
8000	1.00	0.83	0.78	0.72
10000	1.00	0.84	0.79	0.73
12000	1.00	0.83	0.76	0.73
14000	1.00	0.84	0.80	0.74
16000	1.00	0.84	0.78	0.72
18000	1.00	0.86	0.79	0.74
20000	1.00	0.84	0.79	0.74
n	$E^{5,4}(n)$	$E^{6,4}(n)$	$E^{7,4}(n)$	$E^{8,4}(n)$
Nodes				
6000	0.74	0.73	0.72	0.66
8000	0.77	0.80	0.79	0.73
10000	0.80	0.80	0.79	0.76
12000	0.80	0.78	0.79	0.76
14000	0.80	0.81	0.81	0.76
16000	0.77	0.77	0.78	0.76
18000	0.78	0.79	0.79	0.76
20000	0.79	0.77	0.77	0.74

Example: Table 2 shows the percentage of the Attractor execution time with respect the total execution time of Algorithm 2 for game of size up to 20K.

Table 3 shows values of $S_{HT}^{p,q}$ computed starting from values of $Attr$ provided in Table 2 where $r = 30\%$.⁵⁸

Furthermore, with the aim of estimating the expected performance gain of the multilevel algorithm we report the values of $Sc_{1,q}^f$ and $Sc_{1,q}^{ML}$, and $Id[Sc_{1,q}^{ML}]$, obtained by using Propositions 2 and 3. Note that while $S_{HT}^{p,q}$ is the performance metric to be used at the second level of parallelism (the fine grained parallelism), last three metrics, that is, $Sc_{q,1}^{ML}$, $Sc_{q,1}^f$, and $Id[Sc_{1,q}^{ML}]$, are to be intended as the expected performance of the multilevel parallel algorithm.

- Taking into account the Hyper-Threading Technology, we evaluate the values of the efficiency defined as:

$$E^{p,q}(n) = \frac{1}{q'} \frac{T^{1,1}(n)}{T^{p,q}(n)}, \quad (19)$$

where q and p still denote the numbers of cores and threads, respectively, and q' denotes the number of logical processing elements (see Table 6).

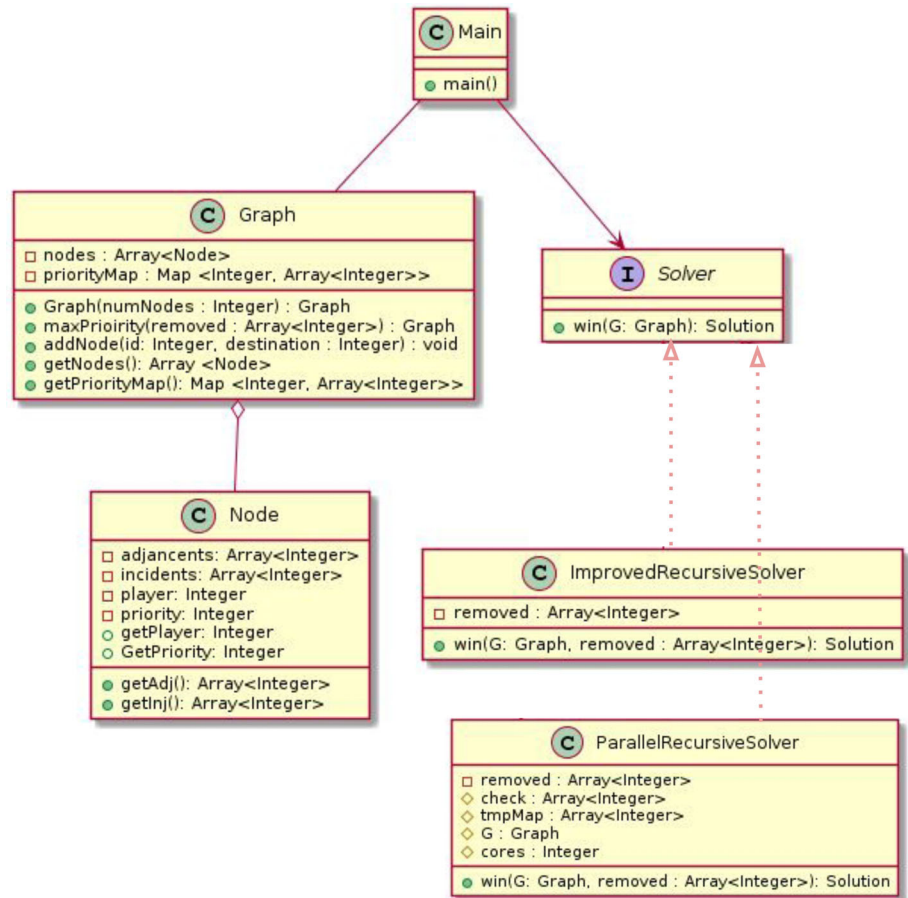
5.2 | Results on a test bed

In this section, we report results of Algorithm 3 on the reference architecture (Intel Hyper-Threading Technology).

5.2.1 | Implementation details

We have implemented Algorithm 3 by using Java 8. The resulting compiled JAR, including all dependencies, is around 4MB on disk. Our Java solver is available at

- <http://ow.ly/haia305hMoV>,
to build the package we use Apache Maven, you can run

FIGURE 3 UML diagram

- “mvn clean compile assembly:single”
to compile and produce a JAR artifact.

5.3 | Framework design

The framework we have designed makes use of the strategy design pattern, guaranteeing an high level of openness and extensibility. The key points are described in the UML Class Diagram in Figure 3.

All the solvers than implement the solver interface can use the same graph class and even the same graph instance if they do not alter it. It use arrays of adjacent and incidents and both graph and node classes have all the attributes needed to express a parity game, being the player that owns the node and the priority. The priority is also stored into a map to allow faster access and filter, providing the possibility to access all nodes having a set priority.

This class is both an active solver and container of all the shared data that the parallel tasks will access. The parallel solver needs a location where all tasks can access and update the information required. Instead of using a communication stream, we just globalize the key components like the tmpMap array, which are initialized each time. This allows the parallel tasks to work almost fully in parallel, not needing to directly communicate with other tasks.

The parallel solver is also designed to be independent from any architecture specification. This means that the number of threads and presumably cores can be both defined in the cores attribute or obtained dynamically. In every case, the parallel tasks will be organized around the defined number of cores, thus not resulting in lower utilization or useless overhead.

5.3.1 | Implementation on a quad core architecture with Hyper-Threading

We have implemented Algorithm 3 on a quad cores Intel(R) i7-4790K CPU at 4.0GHz, 16GB of Ram DDR3 with Hyper-Threading.

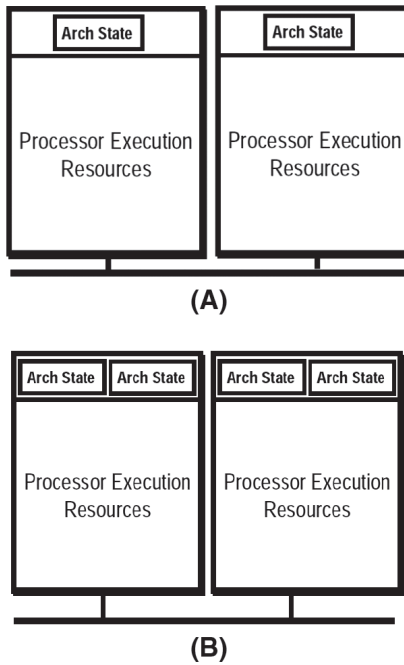


FIGURE 4 (A) Processors without Hyper-Threading Technology; (B) Processors with Hyper-Threading Technology

Hyper-Threading is Intel's proprietary simultaneous multi-threading (SMT) implementation used to improve parallelization of computations (doing multiple tasks at once) performed on x86 microprocessors. Hyper-Threading Technology allows one physical processor package to be perceived as two separate logical processors within the operating system. Processor resources enabled for Hyper-Threading Technology duplicate, tag, or share the majority of resources (see Figure 4⁵⁸). Sharing resources allows a more efficient use of the processor for a significant performance increase, at less than 5% die size and power consumption increase compared to a single processor package.⁵⁹

It makes a single physical processor appear as multiple logical processors. To do this, there is one copy of the architecture state for each processor, and the logical processors share a single set of physical execution resources.

From a software or architecture perspective, this means operating systems and user programs can schedule processes or threads to logical processors as they would on conventional physical processors in a multiprocessor system. From a micro-architecture perspective, this means that instructions from logical processors will persist and execute simultaneously on shared execution resources.

A system with processors that use Hyper-Threading Technology appears to the operating system and application software as having twice the number of processors than it physically has. Operating systems manage logical processors as they do physical processors, scheduling runnable tasks, or threads to logical processors.

In order to exploit the Hyper-Threading Technology, our parallel implementation splits its input data-set into independent chunks which are processed by the parallel tasks in a completely parallel manner.⁶⁰

We have employed an high level/architecture bound implementation to allow more users to easily approach it, also using threads implicitly. We have implemented Algorithm 3 by using Java 8. The algorithm splits its work in parallel task which, besides the same Graph G , Transposed T , Set Removed and player i , takes in input a single node of the set A , namely $A[index]$, and returns a set containing all the nodes attracted from that single one.

Having a deep look at the way the Attractor behaves reveals that it does not really needs to know how many edges are left, but simply if any edge is left. This means that it does not matter if there are one or more edges to be excluded, if one is found it can be excluded regardless.

However, the counting still needs to be done only once, and to do it, each iteration still needs to know if it has been done already or not. This can be achieved with a simple test and set, that is an atomic operation in many other modern programming languages.

For testing we have used multiple instances of random parity games. The graph data structure is represented as a fixed length *array* of objects, where every node contains perfect information such as player, priority, and adjacency and incidence lists. Precisely, following traditions, we have used 1000 random games generated through PGSolver, with a random number of outgoing edges and priorities. In the settings of our arenas, we benchmark graphs with $6K \leq n \leq 20K$ (Figure 5). We tested graphs up to 20K nodes due memory limits on a multicore architecture.

In Figures 4 and 6, we report execution times $T^{p,q}(n)$ for values of $p = 1, \dots, 8$ where p denotes the number of threads (see line 15 in Algorithm 3, by fixing $pmax = 8$) and q still denotes the number of physical cores (Figure 7).

FIGURE 5 Values of $L_{Oh}(q, n)$ in Equation (16) for $6K \leq n \leq 20K$ and $q = 1, 2, 3, 4$

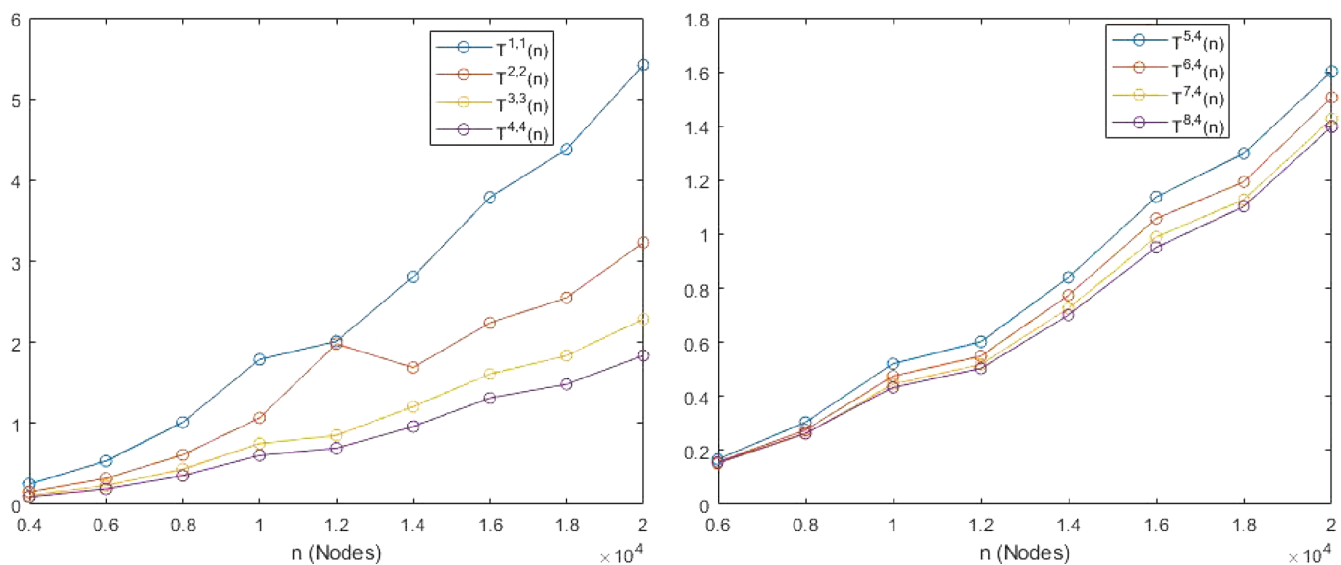
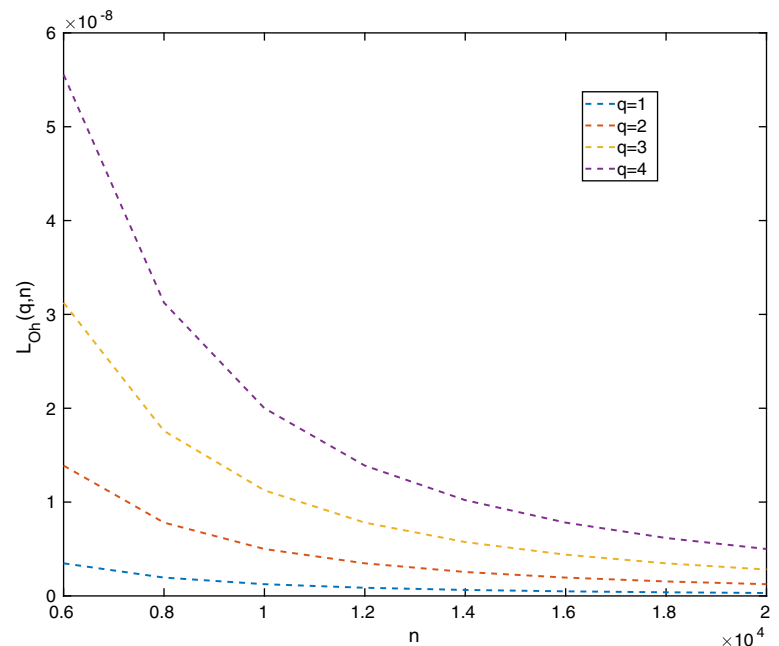


FIGURE 6 Execution times of the fine-grained parallel algorithm as in Table 4

Results in Table 6 and Figure 8 highlight as the use of Simultaneous Multi-Threading, or more specifically, Intel's Hyper-Threading (see last column of Table 6) available on our testing machine, allow to gain back the efficiency, of a quad-core architecture, lost due to memory accesses. This result is strictly in accord with the estimates provided in Reference 58 in which measured performance on common server application benchmarks for this technology show performance gains of up to 30%.

To the aim of giving an overall idea of the performance behavior of the parallel algorithm, the following plots of speed-up and efficiency are shown in the next Figures 6, 7 and 8.

In conclusion, we may say that experimental results confirm that a fine-grained parallel approach which exploits only a single type of parallelism have a clear performance limitations. In this regard, in Table 5 (and Figure 7), we report values of the speed-up of the fine-grained parallel algorithm, compared with the revised expressions of the ideal speed-up and in Table 6, we show the revised values of efficiency. Nevertheless, the values of the expected performance metrics reported in Table 3, tell us that the performance gain we expect to get by using the multilevel parallel algorithm is significant.

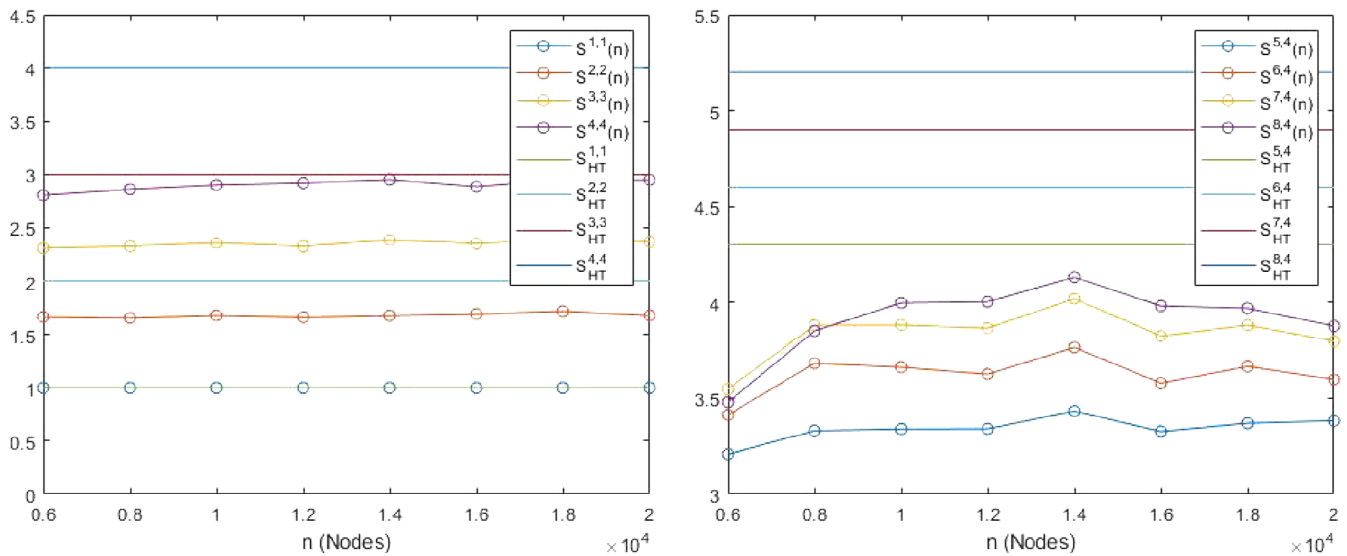


FIGURE 7 Speed-up values of the fine-grained algorithm, compared with respect to $S_{HT}^{p,q}$ as in Table 5

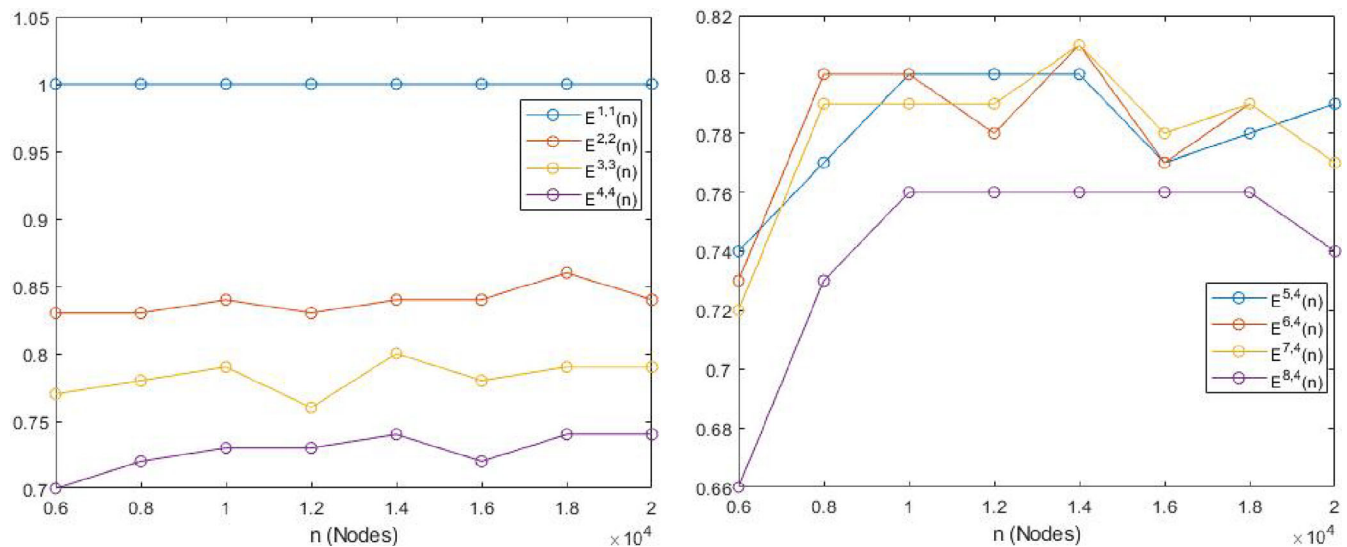


FIGURE 8 Efficiency in Equation (19) as in Table 6

6 | CONCLUSION AND FUTURE WORK

Parity games is an important subject of study in computer science and artificial intelligence. However, the high computational complexity required to solve these kind of games represents a serious bottleneck in their application in complex and general domains. In the last 20 years, several attempts have been carried out in order to introduce fast algorithms or to improve existing ones. An important contribution in this direction is the ZR algorithm, which has been used for some years now as the best performing algorithm in practice. This algorithm have been the subject of several improvements along its implementation, mainly consisting of graph manipulation, improved implementations or just by using better performing programming languages.²⁶

In this article, we take a different and promising scalable approach based on multilevel parallelism. The configurations of the multilevel parallel algorithm which we propose is new and addressed for the first time in this article. The key point of the present work is to formulate the necessary performance analysis that underpin this framework. Hence, results we presented were concerned with the capability of a multigrained algorithm to solve this problem by using such a framework. We hope that these results encourage readers to further extend the framework according to their specific application's requirements (see References 36,61, and 62 for instance). We are now working to implement our approach in a distributed

computing environment or hybrid environments, that is, high performance computing environments where special purpose devices such as Graphics Processing Units are chosen with the aim of accelerating the solution of heavy computational kernels, by combining the fine-grained parallelism with a coarse grained parallelism based on SCC decomposition. Finally, we would like to note that the analysis skips an important part concerning the computation of SCCs and its dynamic distribution among processing elements covering, for instance, the presence of a not uniform SCC decomposition. We are working on this topic by adding a sort of regularization to the SCC decomposition to allow a suitable matching of local solutions (as in Reference 42,63).

ORCID

Luisa D'Amore  <https://orcid.org/0000-0002-3379-0569>

Giuliano Laccetti  <https://orcid.org/0000-0002-0057-2573>

REFERENCES

- Clarke EM, Grumberg O, Peled DA. *Model Checking*. Cambridge, MA: MIT Press; 2002.
- Emerson EA, Jutla C. Tree Automata, μ -Calculus and Determinacy. *FOCS '91*. Washington, DC: IEEE Computer Society; 1991:368-377.
- Kupferman O, Vardi MY, Wolper P. An automata theoretic approach to branching-time model checking. *J ACM*. 2000;47(2):312-360.
- Aminof B, Malvone V, Murano A, Rubin S. Graded strategic logic: reasoning about uniqueness of Nash equilibria. Paper presented at: Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems, Singapore; 2016:698-706.
- Aminof B, Kupferman O, Murano A. Improved model checking of hierarchical systems. *Inf Comp*. 2012;210:68-86.
- Berthon R, Maubert B, Murano A, Rubin S, Vardi MY. Strategic logic with imperfect information. Paper presented at: Proceedings of the 32 Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), Beijing (China); 2017:1-12.
- Chatterjee K, Henzinger TA, Jurdzinski M. Mean-payoff parity games. Paper presented at: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science LICS'05, Chicago, IL; 2005:178-187; IEEE.
- Ceremak P, Lomuscio A, Mogavero F, Murano A. Practical verification of multiagent systems against SIK specifications. *Inf Comput*. 2018;261:588-614.
- Chatterjee K, Doyen L, Henzinger TA, Raskin J-F. Generalized mean-payoff and energy games. *FSTTCS '10, Series LIPIcs 8*. Richmond, Germany: Leibniz International Proceedings in Informatics Schloss Dagstuhl – Leibniz-Zentrum für Informatik; 2010:505-516.
- Malvone V, Murano A, Sorrentino L. Concurrent multi-player parity games. *AAMAS '2016*. New York, NY: ACM; 2016:689-697.
- Mogavero F, Murano A, Perelli G, Vardi MY. Reasoning about strategies: on the model-checking problem; December, 2011. arXiv:1112.6275.
- Mogavero F, Murano A, Perelli G, Vardi MY. Reasoning about strategies: on the satisfiability problem. *Log Methods Comput Sci*. 2017;13(1):1-37.
- Mogavero F, Murano A, Perelli G, Vardi MY. Reasoning about strategies: on the model-checking problem. *ACM Trans Comp Log*. 2014;15(4):1-47.
- Mogavero F, Murano A, Sauro L. On the boundaries of behavioral strategies. Paper presented at: Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science; June 2013:263-272.
- Rubin S, Zueleger F, Murano A, Aminof B. Verification of asynchronous mobile-robots in partially-known environments. *Principles and Practice of Multi-Agent Systems, PRIMA, Lecture Notes in Computer Science*. Vol 9387. Cham: Springer; 2015:185-200.
- Mogavero F, Murano A, Sorrentino L. On promptness in parity games. *Fund Inform*. 2013;139(3):277-305.
- Jurdzinski M. Deciding the winner in parity games is in $UP \cap co-UP$. *Inf Process Lett*. 1998;68(3):119-124.
- Zielonka W. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor Comput Sci*. 1998;200(1-2):135-183.
- Jurdzinski M. Small progress measures for solving parity games. Paper presented at: Proceedings of the Annual Symposium on Theoretical Aspects of Computer Science; vol 1770, 2000:290-301; Springer, Berlin, Heidelberg.
- Vöge J, Jurdzinski M. A discrete strategy improvement algorithm for solving parity games. Paper presented at: Proceedings of the International Conference on Computer Aided Verification; 2000:202-215; Springer, Berlin, Heidelberg.
- Schewe S. Solving parity games in big steps. Paper presented at: Proceedings of the International Conference on Foundations of Software Technology and Theoretical Computer Science; 2007:449-460. Springer, Berlin, Heidelberg.
- Calude CS, Jain S, Khoussainov B, Li W, Stephan F. Deciding parity games in quasipolynomial time. Paper presented at: Proceedings of the 2017 Symposium on Theory of Computing; 2017:252-263; ACM, New York, NY.
- Friedmann O, Lange M. Solving parity games in practice. Paper presented at: Proceedings of the International Symposium on Automated Technology for Verification and Analysis; 2009:182-196; Springer, Berlin, Heidelberg.
- Friedmann O, Lange M. *The PGSolver Collection of Parity Game Solvers*. Germany: University of Munich; 2009.
- Antonik A, Charlton N, Huth M. Polynomial-time under-approximation of winning regions in parity games. *Electr Notes Theoret Comput Sci*. 2009;225:115-139.
- Di Stasio A, Murano A, Prignano V, Sorrentino L. Solving parity games in scala. Paper presented at: Proceedings of the 11th International Symposium, Formal Aspects of Component Software FACS 2014, Bertinoro, Italy, September 10-12, 2014, Revised Selected Papers; 2014:8997, Springer, New York, NY.
- Arcucci R, Marotta U, Murano A, Sorrentino L. Parallel parity games: a multicore attractor for the Zielonka recursive algorithm. *Proc Comput Sci*. 2017;108:525-534.
- Dijk TV. Parallel parity games: a multicore attractor for the Zielonka recursive algorithm. Paper presented at: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems; 2018:291-308; Springer, New York, NY.
- Kant G, Van De Pol J. Generating and solving symbolic parity games; 2014. arXiv preprint arXiv:1407.7928.
- Godefroid P, van Leeuwen J, Hartmanis J, Goos G, Wolper P. *Partial-order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Vol 1032. Heidelberg, Germany: Springer; 1996.
- Grumberg O, Lange M, Leucker M, Shoham S. When not losing is better than winning: abstraction and refinement for the full μ -calculus. *Inf Comput*. 2007;205(8):1130-1148.
- van de Pol J, Weber M. A multi-core solver for parity games. *Electr Notes Theor Comput Sci*. 2008;220(2):19-34.
- Hoffmann P, Luttenberger M. Solving parity games on the GPU. *Automated Technology for Verification and Analysis*. Cham: Springer; 2013:455-459.

34. Fearnley J. Efficient parallel strategy improvement for parity games. In: Majumdar R, Kunčák V, eds. *Computer Aided Verification*. Vol 10427. New York, NY: Springer; 2017.
35. Bertero M, Bonetto P, Carracciolo L, et al. MedIGrid: a medical imaging application for computational Grids. Paper presented at: Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS 2003, Nice (France); 2003; IEEE.
36. D'Amore L, Arcucci R, Carracciolo L, Murli A. A scalable approach for variational data assimilation. *J Sci Comput*. 2014;61(2):239-257.
37. D'Amore L, Arcucci R, Carracciolo L, Murli A. DD-OceanVar: a domain decomposition fully parallel data assimilation software for the mediterranean forecasting system. *Proc Comput Sci*. 2013;18:1235-1244.
38. D'Amore L, Arcucci R, Marcellino L, Murli A. A parallel three-dimensional variational data assimilation scheme, numerical analysis and applied mathematics. Paper presented at: Proceedings of the International Conference on Numerical Analysis and Applied Mathematics, Halkidiki (Greece); vol. 1389, 2011:1829-1831; AIP Publishing.
39. D'Amore L, Casaburi D, Galletti A, Marcellino L, Murli A. Integration of emerging computer technologies for an efficient image sequences analysis. *Integr Comput Aid Eng*. 2011;18(4):365-378.
40. Jiri B, Jakub C, Jacob P. Distributed algorithms for SCC decomposition. *J Logic Comput*. 2011;21(1):23-44.
41. Grädel E, Wolfgang T, Wilke T. *Automata, Logics, and Infinite Games, 2500 of LNCS*. New York, NY: Springer; 2002.
42. Carracciolo L, D'Amore L, Murli A. Towards a parallel component for imaging in PETSc programming environment: a case study in 3-D echocardiography. *Concurr Comput Pract Exp*. 2006;32(1):67-83.
43. Papadimitriou C. Algorithms, games, and the internet. Paper presented at: Proceedings of the 33rd Annual ACM Symposium on Theory of Computing, Hersonissos Greece; 2001:749-753; ACM.
44. Reif JH. The complexity of two-player games of incomplete information. *J Comput Syst Sci*. 1984;29(2):274-301.
45. Bonatti PA, Lutz C, Murano A, Vardi MY. The complexity of enriched mu-calculi. *Log Methods Comput Sci*. 2008;4(3):1-27.
46. Kupferman O, Vardi MY. Weak alternating automata and tree automata emptiness. *STOC'98*. New York, NY: ACM; 1998:224-233.
47. Friedmann O. Recursive algorithm for parity games requires exponential time. *RAIRO-Theor Inform Appl*. 2011;45(4):449-457.
48. McNaughton R. Infinite games played on finite graphs. *Ann Pure Appl Log*. 1993;65(2):149-184.
49. Murli A, D'Amore L, Gregoretti F, Oliva G, Laccetti G. A multi-grained distributed implementation of the parallel block conjugate gradient algorithm. *Concurr Comput Pract Exp*. 2010;22(15):2053-2072.
50. Cuomo S, De Michele P, Galletti A, Marcellino L. A parallel PDE-based numerical algorithm for computing the optical flow in hybrid systems. *J Comput Sci*. 2016;22(1):228-236.
51. Antonelli L, Carracciolo L, Ceccarelli M, D'Amore L, Murli A. Total variation regularization for edge preserving 3D SPECT imaging in high performance computing environments. Paper presented at: Proceedings of the International Conference on Computational Science; 2002:171-180; Springer, Berlin, Heidelberg/Germany.
52. Juurlink B, Meenderink C. Amdahl's law for predicting the future of multicores considered harmful. *ACM SIGARCH Comput Arch News*. 2012;40(2):1-9.
53. D'Amore L, Mele V, Romano D, Laccetti G. A multilevel algebraic approach to performance analysis of parallel algorithms. *Comput Inform*. 2019;4:817-850.
54. Fischer MJ, Ladner RE. Propositional dynamic logic of regular programs. *J Comput Syst Sci*. 1979;18(2):194-211.
55. McCool M, Reinders J, Robison A. *Structured Parallel Programming: Patterns for Efficient Computation*. Waltham, MA 02451, USA: Elsevier; 2012.
56. Amdahl GM. Validity of the single processor approach to achieving large scale computing capabilities. Paper presented at: Proceedings of the AFIPS Spring Joint Computer Conference, ATLANTIC CITY, NJ; 1967:56.
57. Marr DT, Binns F, Hill DL, et al. Hyper-threading technology architecture and microarchitecture. *Intel Technol J*. 2002;6(1):1-12.
58. Barone GB, Boccia V, Bottalico D, et al. An approach to forecast queue time in adaptive scheduling: how to mediate system efficiency and users satisfaction. *Int J Parall Progr*. 2017;45(5):1164-1193.
59. Marr DT, Binns F, Hill DL, et al. Upton, hyper-threading technology architecture and microarchitecture. *Intel Technol J*. 2002;6(1):8947-8950.
60. Laccetti G, Montella R, Palmieri C, Pelliccia V. The high performance internet of things: using GVirtus to share high-end GPUs with ARM based cluster computing nodes. *LNCS 8384*. Vol 734-744. Berlin, Germany: Springer; 2014.
61. Murli A, Boccia V, Carracciolo L, D'Amore L, Laccetti G, Lapegna M. Monitoring and migration of a PETSc-based parallel application for medical imaging in a grid computing PSE. *Grid-Based Problem Solving Environments*. IFIP The International Federation for Information Processing. Vol 239. Boston, MA: Springer; 2007:421-432.
62. Murli A, D'Amore L. Regularization of a Fourier series method for the Laplace transform inversion with real data. *Inverse Probl*. 2002;18(4):1185-1205.
63. Tullsen DM, Eggers SJ, Levy HM. Simultaneous multithreading: maximizing on-chip parallelism. *ACM SIGARCH Comput Arch News*. 1995;23:392-403.

How to cite this article: D'Amore L, Murano A, Sorrentino L, Arcucci R, Laccetti G. Toward a multilevel scalable parallel Zielonka's algorithm for solving parity games. *Concurrency Computat Pract Exper*. 2020:e6043. <https://doi.org/10.1002/cpe.6043>