Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 108C (2017) 525–534

Procedia Computer Science

International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland

Parallel Parity Games: a Multicore Attractor for the Zielonka Recursive Algorithm

Rossella Arcucci, Umberto Marotta, Aniello Murano, and Loredana Sorrentino

University of Naples "Federico II", Naples, Italy.
{name.surname}@unina.it

Abstract

Parity games are abstract infinite-duration two-player games, widely studied in computer science. Several solution algorithms have been proposed and also implemented in the community tool of choice called PGSolver, which has declared the Zielonka Recursive (ZR) algorithm the best performing on randomly generated games. With the aim of scaling and solving wider classes of parity games, several improvements and optimizations have been proposed over the existing algorithms. However, no one has yet explored the benefit of using the full computational power of which even common modern multicore processors are capable of. This is even more surprisingly by considering that most of the advanced algorithms in PGSolver are sequential.

In this paper we introduce and implement, on a multicore architecture, a parallel version of the Attractor algorithm, that is the main kernel of the ZR algorithm. This choice follows our investigation that more of the 99% of the execution time of the ZR algorithm is spent in this module. We provide testing on graphs up to 20K nodes generated through PGSolver and we discuss performance analysis in terms of strong and weak scaling.

Peer-review under responsibility of the scientific committee of the International Conference on Computational Science

Keywords: Parity Games, multicore Attractor, parallel Zielonka Recursive algorithm

1 Introduction

A parity game [10, 21] is an abstract infinite-duration game played on a directed graph, where each node is colored by a priority. Parity games represent a powerful mathematical framework to address fundamental questions in computer science and mathematics. Their connection with other infinite duration games is strict, for example 'mean' and 'discounted' payoff, 'stochastic', and 'multi-agent' games [5, 6, 19]. In formal system design and verification [7, 10, 17], parity games arise as a natural evaluation machinery to automatically and exhaustively check for reliability of distributed and reactive systems.

Through the years several algorithms to solve parity games have been proposed. Among the others, well known are the Zielonka Recursive (ZR)[21], the Small Progress Measures [16], the Strategy Improvement [20] and the Big Step [18] algorithms. All of them have been implemented in PGSolver, a collection of tools to solve, benchmark and generate parity games. The tool

1877-0509
Peer-review under responsibility of the scientific committee of the International Conference on Computational Science
10.1016/j.procs.2017.05.120

is written in OCaml by Oliver Friedman and Martin Lange [12, 13]. Noteworthy, PGSolver declared the Zielonka Recursive Algorithm as the best performing in practice.
Along the years several improvements on the existing parity games algorithms and implementations have been considered in order to speed-up their execution time. PGsolver itself brings some general preprocessing optimizations such as SCC decomposition, self-cycles removal, and priority compression [1, 16]. A direction that has been positively exploited recently concerns improving the code implementation along with the use of better performing programming languages. For a long time, the scientific community relied on OCaml as the best performing language to be used in this settings. In [9] an Improved Recursive Algorithm is introduced and presented a tool written in Scala to solve parity games, showing a capability of gaining up to two orders of magnitude in running time compared to PGSolver. Here, as a side result, we also show that a similar gain can be obtained by using Java instead of Scala. The sequence reported in (1) schematically represents the improvements over Zielonka Recursive algorithm. Although PGSolver is no longer state of the art anymore, it is still a main reference point for comparison. To have a real feeling of such an improvement, observe that the OCaml Classic Implementation (OCI, for short) [13], included in PGSolver, with one of the biggest (completely) random game it could handle, having 8000 nodes, takes 418 seconds on our machine to be solved. With a simple improvement and the same programming language, the Improved Implementation (IRI, for short) [9] takes 47 seconds being 10 times faster. With the introduction of the Java implementation (JRI, for short), solving the same game on the same machine takes just 0.7 seconds being over 50 times faster.

$$OCI \ <_{\times 10} \ IRI \ <_{\times 50} \ JRI \tag{1}$$

The problem of declaring the winner in a parity game is known to be in **UPTime** $\cap$ **CoUPTime** [15]. The time complexity of the recursive parity algorithms encourages the implementation of software in High Performance Computing (HPC) environment expecially for big size games. In [14] the authors present a GPU-specific implementation of algorithms for solving parity games. They compare the obtained results with the PGSolver implementation of some algorithm. Precisely, they provided a comparison with a multicore CPU version of the Strategy Improvement algorithm. Conversely and noteworthy, no multicore version of the ZR algorithm are provided or presented. Also, they provide comparisons of execution times but a study in terms of performance gain for their algorithm is not provided.
In this paper, we face the development of a parallel ZR algorithm able to exploit the resources of the available multicore architecture. One important issue is the study of scalability of the algorithm itself [4]. Here, scalability refers to the capability of the algorithm to (i) exploit performance of the available computing architectures in order to get a solution in a reduced execution time (strong scaling), (ii) use additional computational resources effectively to solve increasingly larger problems (weak scaling). We start by implementing an Improved version of the ZR algorithm we describe in the next section. As we have seen that more than the 99% of the execution time (see Table 1) of the ZR Algorithm is spent in its Attractor, we focus our attention on the development of a parallel version of this module.

Game Size	4000	8000	12000	16000	20000
Tot (seconds)	0.215	0.959	1.917	3.614	5.229
Attr (seconds)	0.214	0.958	1.911	3.612	5.227
%	99.70%	99.83%	99.77%	99.96%	99.97%

Table 1: Percentage of the Attractor execution time with respect the Total execution time.

We have implemented the parallel algorithm (JRP, for short) on a multicore processor with Hyper-Threading. As we show in Section 4, we have obtained a gain in terms of reductions of execution times by using Hyper-Threading such that:

$$JRI \ <_{\times q} \ JRP \tag{2}$$

where q is the number of cores involved into the computations. Then, from (2) and (1), we have obtained a gain of $500 \times q$ with respect the OCI version, thus performing 2000 times faster on our tests on random graphs by using a machine with 4 hyper-threaded cores. Then, if the solution of a game with OCI would require one year, by using JRP on an hyper-threaded processor (like Intel's Xeon E5-2699 v4 Processor) it would require less than one hour.
The paper is structured as follows. In Section 2, we provide some preliminary notion about the Parity Games. We introduce the Zielonka Recursive Algorithm and the Attractor routine on which is based the solution of the game. In Section 3 we introduce the parallel version of the Improved Attractor introduced in [9]. The parallel approach we employ is based on a decomposition of the problem with a distribution of the nodes along the cores. In Section 4, we provide the implementation of the proposed paralled algorithm and discuss its performance in terms of weak and strong scaling. In this section we also report the link where our Java solver is available. Finally, in Section 5 we give some conclusions and future work directions.

2 Parity Games

In this section, we report some basic concepts about parity games, including the Zielonka Recursive Algorithm. As they are common definitions, an expert reader can also skip this part.
A parity game is a tuple $G = (V, V_0, V_1, E, \Omega)$ where (V, E) forms a directed graph whose set of nodes is partitioned into $V = V_0 \cup V_1$, with $V_0 \cap V_1 = \emptyset$, and $\Omega : V \to \mathbb{N}$ is the priority function that assigns to each node a natural number called the priority of the node.
We assume E to be total, i.e. for every node $v \in V$, there is a node $w \in V$ such that $(v, w) \in E$. In the following we also write vEw in place of $(v, w) \in E$ and use $vE := \{w \mid vEw\}$.
Parity games are played between two players called Player 0 and Player 1. Starting in a node $v \in V$, both players construct an infinite path (the play) through the graph 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 node $w \in v_nE$ 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. 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 $\max\{p \mid \forall j . \exists k \geq j : \Omega(v_k) = p\} \bmod (2) = i$.
A strategy for Player i is a partial function $\sigma_i : V^* V_i \to V$, such that, for all sequences $v_0 \dots v_n$ with $v_{j+1} \in v_jE$, with $j = 0, \dots, n-1$, and $v_n \in V_i$ we have that $\sigma(v_0 ... v_n) \in v_nE$. 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 $v_{j+1} = \sigma(v_0 \dots v_j)$. A strategy σ for Player $i(\sigma_i)$ is a winning strategy in node v 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. A strategy σ for Player i is called memoryless 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 $\sigma(v_0 \dots v_n) = \sigma(w_0 \dots w_m)$. That is, the value of the strategy on a path only depends on the last node on that path. Starting from G we construct two sets $W_0, W_1 \subseteq V$ such that W_i is the set of all nodes v such that Player i wins the game G starting in v. Parity games enjoy determinacy, meaning that for every node v either $v \in W_0$ or $v \in W_1$ [10].
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 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 nodes from which Player i can move "towards" U and Player $1-i$ must move "towards" U. The i-attractor of U is denoted by $Attr_i(G,U)$. Formally, for all $k \in \mathbb{N}$, the i-attractor is defined as $Attr_i(G,U) = \bigcup_{k=0}^{\infty} Attr_i(G,U)^k$ where $Attr_i(G,U)^0 = U$ and $Attr_i(G,U)^{k+1} = Attr_i(G,U)^k \cup \{v \in V_i \mid \exists (v,w) \in E : w \in Attr_i(G,U)^k\} \cup \{v \in V_{1-i} \mid \forall (v,w) \in E : w \in Attr_i(G,U)^k\}$. Intuitively, the set $Attr_i(G,U)$ is the largest set of nodes from which the Player i with $i \in \{0,1\}$ can attract the token from any node in $Attr_i(G,U)$ to U in a finite number of step.
In other words, $Attr_i(G,U)^k$ consists of all nodes such that Player i with $i \in \{0,1\}$ can force any play to reach U in at most k moves.
Let A be an arbitrary attractor set. The game $G \setminus A$ is the game restricted to the nodes $V \setminus A$, i.e. $G \setminus A = (V \setminus A, V_0 \setminus A, V_1 \setminus A, E \setminus (A \times V \cup V \times A), \Omega_{V \setminus A})$. It is worth observing that the totality of $G \setminus A$ is ensured from A being an attractor.[11]

2.1 The Zielonka Recursive Algorithm

The ZR algorithm makes use of a *divide and conquer* technique. It constructs the winning sets for both players using sub-games that remove the nodes 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*. Here, we consider the Attractor described in Algorithm 1, which is an improved version of the classical one [9].[1]
The ZR algorithm starts by invoking the routine $win(G)$, which takes a graph G as input and, after a number of recursive calls over ad hoc built sub-games, returns the winning sets (W_0, W_1) for Player 0 and Player 1, respectively. The running time complexity of the algorithm is exponential in the number of priorities, and polynomial in the number of edges and nodes. See [9, 21] for more details.

3 The Parallel Algorithm

In this section we introduce, in Algorithm 2 , the parallel version of the Zielonka's Attractor algorithm. We start from the improved Attractor proposed in [9]and reported in Algorithm 1. The parallel approach we introduce is based on a decomposition of the problem [2, 3] with a distribution of the nodes along the cores. The Input of the resulting algorithm is split into independent chunks which are processed by the parallel tasks in a completely parallel manner. Algorithm 2 still 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 1.

A new function *asyncAttr* is introduced in Algorithm 2 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.

[1] As an improvement, the *Attractor* introduced in [9] is able to check if a given node 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 [9], by adding the node in a set called *Removed* that is given as input to the Attractor, so it can work only on the nodes present in the evaluated sub-graph.

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 1 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
```

Algorithm 2 Zielonka Parallel 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(p, |A| − index, pmax)
9          for id_proc ∈ (0, p) :
10       A_{id_proc} = A_{id_proc} ∪ asyncAttr(G, T,
11                A[index], Removed, i)
12           A = A ∪ A_{id_proc}
13     index+ = 1
14   return A
15
16 function numproc(p,indexLeft,pmax):
17     if indexLeft < pmax :
18       p = indexLeft
19     else :
20       p = pmax
21     return p
22
23 function asyncAttr(G,T,index,Removed,i):
24   A = {}
25   for v0 ∈ adj(T, index):
26     if v0 ∉ Removed :
27       if test(tmpMap[v0] == ∞) and
28              set(tmpMap[v0] = 0):
29         if player(v0) == i :
30           A = A ∪ v0
31           tmpMap[v0] = 1
32         else:
33           adj_counter = 1
34           for x ∈ adj(G, v0) :
35             if x ∉ Removed:
36               adj_counter+ = 1
37           tmpMap[v0]+ = adj_counter
38           if adj_counter == 1 :
39             A = A ∪ v0
40       else if player(v0) == j :
41         tmpMap[v0]− = 1
42       if tmpMap[v0] == 1:
43         A = A ∪ v0
44   return A
```

The function *numproc*, at line 15, determines the number of threads p involved as function of the number of nodes processed and the maximum number of concurrently executable threads p_{max}. In Algorithm 1, on each iteration two important aspects have to be faced: (a) whether it is the first to stumble upon that edge (see line 10) and to count all the adjacent edges (see line 18); (b) how many edges remain to be excluded before the node is attracted (see lines from 14 to 21); this information is stored in *tempMap*.

These aspects are also faced in Algorithm 2. Regarding the point (a), we have that the counting needs to be done just 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 (see line 26). The point (b) is

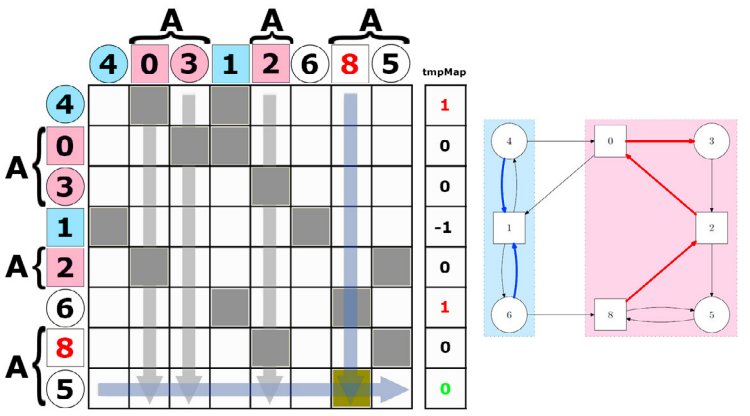

Figure 1: Snapshot of an execution of the Attractor algorithm: Player 0's (Even) nodes are represented by circles while squares represent Player 1's (Odd) nodes. The numbers indicated inside represent the priorities. On the right there is a depiction of a directed graph representing a Parity Game where 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 unreached and 1 non-attracted. As indicated by the vertical arrows, each node is processed in parallel by the Algorithm 2.

actually redundant, as it does not really need 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 (see lines 38 and 39). This trick is useless on the sequential algorithm as all the edges are always counted first, but on a parallel algorithm this allows an edge to be excluded even before they are counted with no need to wait, and the same is true for counting that can be translated in a simple addition to all the removed edges. Another important aspect to consider is 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 2.

Read	Meaning in Algorithm 2	Meaning in Algorithm 1
$tmpMap[v_0] < 0$	At least one edge of v_0 has been excluded before the edges have been counted.	v_0 has not been reached yet.
$tmpMap[v_0] == 0$	Someone is counting v_0's edges.	v_0 can be attracted
$tmpMap[v_0] == 1$	v_0 can be attracted	At least one edge of v_0 needs to be excluded for it to be attracted.
$tmpMap[v_0] > 1$	At least one edge of v_0 needs to be excluded for it to be attracted.	At least two edges of v_0 need to be excluded for it to be attracted.
$tmpMap[v_0] \to \infty$	v_0 has not been reached yet.	

Table 2: tmpMap's notation.

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 line 5 and 26 in Algorithm 2.

4 Implementation and Performance Analysis

We have implemented Algorithm 2 by using Java 8 on a quad cores Intel(R) i7-4790K CPU @4.0GHz, 16GB of Ram DDR3 with Hyper-Threading.
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 "mvn clean compile assembly:single" to compile and produce a JAR artifact.
We 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 $4K \leq n \leq 20K$.

In Table 3, we present the execution times we denote with $T^p(n)$ for values of $p = 1, \ldots, 8$ where p denotes the number of threads (see line 15 in Algorithm 2, by fixing $pmax = 8$).

n Nodes	$T^1(n)$ Seconds	$T^2(n)$ Seconds	$T^3(n)$ Seconds	$T^4(n)$ Seconds	$T^6(n)$ Seconds	$T^8(n)$ Seconds
4000	0.536	0.322	0.232	0.191	0.157	0.154
8000	1.013	0.612	0.435	0.354	0.275	0.263
12000	2.011	1.198	0.852	0.693	0.549	0.503
16000	3.787	2.239	1.609	1.312	1.058	0.951
20000	5.420	3.231	2.286	1.837	1.506	1.398

Table 3: Execution times.

We evaluate the performance of the parallel algorithm on the reference architecture by measuring scalability in terms of strong and weak scaling. For the strong scaling analysis we evaluate the speed-up we denote as:

$$SS^p(n) = \frac{T^1(n)}{T^p(n)} \tag{3}$$

for $n = 4K, \ldots, 20K$ and $p = 1, \ldots, 8$ as showed in Figure 2. The *ideal speed-up*, according with its classical definition is equals the number of physical execution units involved which is cores in this case. Values of speed-up show how the performance improve as the size of the game increases. This is due to the overhead decreasing.

We also evaluate the values of the efficiency:

$$SE^p(n) = \frac{T^1(n)}{qT^p(n)} \tag{4}$$

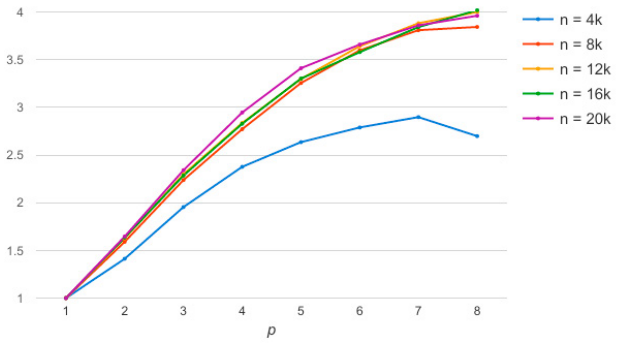

Figure 2: Strong Scaling: $SS^p(n)$ as function of p for $n = 4K, 8K, 12K, 16K, 20K$.

where q and p denote the numbers of cores and threads, respectively. Results in Table 4 underline as the use of Simultaneous Multi-Threading, or more specifically, Intel's Hyper-Threading (see last line of Table 4) available on our test 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 [8] in which measured performance on common server application benchmarks for this technology show performance gains of up to 30%.

q	p	$SE^p(4K)$	$SE^p(8K)$	$SE^p(12K)$	$SE^p(16K)$	$SE^p(20K)$
1	1	1.00	1.00	1.00	1.00	1.00
2	2	0.83	0.83	0.84	0.85	0.84
4	4	0.70	0.71	0.72	0.72	0.74
4	8	0.87	0.96	0.99	0.99	0.97

Table 4: Strong Scaling Efficiency.

For the weak scaling analysis, the problem size and the number of processors increase. In this case the scalability means the ability of maintaining a fixed time-to-solution solving larger problems on larger architectures. We evaluate the scaling factor $WS^q(n)$ defined as the ratio:

$$WS^q(n) = \frac{T^1_{flop}(n)}{qT^p_{flop}(q \cdot n)} \tag{5}$$

where $T^1_{flop}(n)$ is the computing time required for the execution of $T(n)$ floating point operations and where the values of n and q are fixed such that $n = q \cdot 4K$. As the time complexity of the ZR Attractor algorithm is $T(n) = O(e \cdot n^d)$ where n, e, and d denote the number of nodes, edges, and priorities in the game, respectively, the Theoretical scaling factor in (5) is such that:

$$WS^q(n) \simeq \frac{e \cdot n^d}{qe \cdot \left(\frac{q \cdot n}{q}\right)^d} \simeq \frac{1}{q} \tag{6}$$

We observe that, considering the implementation of Algorithm 2 in a software on our architecture, the values of $WS^q(n)$, we denote here with $Measured\ WS^q(n)$, are obtained as

$$Measured\ WS^q(n) = \frac{T^1_{flop}(n)}{qT^p_{flop}(q \cdot n) + qT^p_{mem}(q \cdot n)} \tag{7}$$

where $T^p_{mem}(q \cdot n)$ denotes the time for processors synchronization and memory accesses. Then, it is

$$Measured\ WS^q(n) = \frac{\frac{T^1_{flop}(n)}{qT^p_{flop}(q \cdot n)}}{1 + \frac{qT^p_{mem}(q \cdot n)}{qT^p_{flop}(q \cdot n)}} < \frac{T^1_{flop}(n)}{qT^p_{flop}(q \cdot n)} = WS^q(n) \tag{8}$$

Table 5 shows values of the $Measured\ WS^q(n)$ compared with the values given by (6). Weak scalability results provide informations about the performance of the algorithm on the available Hyper-Threading architecture for solving a game of bigger size without a meaningful increase of the execution time. The blue values obtained by using Hyper-Threading exhibit improvements in terms of weak scalability with six threads running. As expected, performance decrease with eight threads running due to the memory overhead.

q	p	$Measured\ WS^q(n)$	$Theoretical\ WS^q(n)$
1	1	1.00	1.00
2	2	0.44	0.50
3	3	0.21	0.33
4	4 | 6 | 8	0.10 | 0.24 | 0.14	0.25

Table 5: Weak Scaling where $n = 4K$

The obtained performance results confirm that the parallel approach which we have introduced ,based on a decomposition of the problem with a distribution of the nodes along the cores, allows us to lead to an effective reduction of the overall execution time needed to solve the problem on architectures such as multicore processors with Hyper-Threading.

5 Conclusion

Parity game is an important subject of study in computer science. However, the high computational complexity required to solve these games represents a serious bottleneck in their application in complex and general domains. In the last twenty years, several attempts have been caried 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 and its implementation have been the subject of several improvements, mainly consisting of graph manipulation, improved implementations or just by using better performing programming languages [9].

In this paper, we have taken for the first time a different and promising scalable approach. Precisely we have shown that it is possible to run efficiently the ZR algorithm on multiple cores by parallelizing its Attractor, which has been observed to take more than 99% of execution time on random arenas. We have implemented the parallel algorithm using Java 8 and compared it to its sequential implementation in the same language and the widely used one in PGsolver (i.e., in OCaml). Our benchmarks show that the parallel implementation performs even faster than 4 times with 4 hyper-threaded cores, on random games, on average, compared to the sequential counterpart, thus achieving an overall (bounded) linear-time speedup. Compared to PGsolver's implementation, the execution gets as high as 2000 times faster.

References

[1] A. Antonik, N. Charlton, and M. Huth. Polynomial-time under-approximation of winning regions in parity games. *ENTCS*, 225:115–139, 2009.

[2] R. Arcucci, L. D'Amore, and L. Carracciuolo. On the problem-decomposition of scalable 4d-var data assimilation models. In *HPCS'15*, pages 589–594. IEEE, 2015.

[3] R. Arcucci, L. D'Amore, L. Carracciuolo, G. Scotti, and G. Laccetti. A decomposition of the tikhonov regularization functional oriented to exploit hybrid multilevel parallelism. *International Journal of Parallel Programming*, pages 1–22, 2016.

[4] R. Arcucci, L. D'Amore, S. Celestino, G. Laccetti, and A.Murli. A scalable numerical algorithm for solving tikhonov regularization problems. In *PPAM'15*, pages 45–54. Springer, 2016.

[5] K. Chatterjee, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Generalized mean-payoff and energy games. In *FSTTCS'10*, LIPIcs 8, pages 505–516, 2010.

[6] K. Chatterjee, T. A. Henzinger, and M. Jurdzinski. Mean-payoff parity games. In *LICS'05*, pages 178–187, 2005.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2002.

[8] D. L. Hill G. Hinton D. A. Koufaty J. A. Miller D. T. Marr, F. Binns and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, Q1:1–12, 2002.

[9] A. Di Stasio, A. Murano, V. Prignano, and L. Sorrentino. Solving parity games in scala. In *FACS'14*, LNCS 8997. Springer, 2014.

[10] E. A. Emerson and C. Jutla. Tree automata, μ-calculus and determinacy. In *FOCS'91*, pages 368–377, 1991.

[11] O. Friedmann. Recursive algorithm for parity games requires exponential time. *RAIRO-Theoretical Informatics and Applications*, 45(04):449–457, 2011.

[12] O. Friedmann and M. Lange. The pgsolver collection of parity game solvers. *University of Munich*, 2009.

[13] O. Friedmann and M. Lange. Solving parity games in practice. In *ATVA'09*, LNCS 5799, pages 182–196. Springer, 2009.

[14] P. Hoffmann and M. Luttenberger. Solving parity games on the gpu. In *ATVA'13*, pages 455–459. Springer, 2013.

[15] M. Jurdzinski. Deciding the winner in parity games is in up $\cap$ co-up. *Inf. Process. Lett.*, 68(3):119–124, 1998.

[16] M. Jurdzinski. Small progress measures for solving parity games. In *STACS'00*, LNCS 1770, pages 290–301. Springer, 2000.

[17] M. Y. Vardi O. Kupferman and P. Wolper. An automata theoretic approach to branching-time model checking. *JACM*, 47(2):312–360, 2000.

[18] S. Schewe. Solving parity games in big steps. In *FSTTCS'07*, LNCS 4855, pages 449–460. Springer, 2007.

[19] A. Murano V. Malvone and L. Sorrentino. Concurrent multi-player parity games. In *AAMAS 2016*, pages 689–697. ACM, 2016.

[20] J. Vöge and M. Jurdzinski. A discrete strategy improvement algorithm for solving parity games. In *CAV'00*, LNCS 1855, pages 202–215. Springer, 2000.

[21] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor. Comput. Sci.*, 200(1-2):135–183, 1998.