UNIVERSIDADE DO VALE DO RIO DOS SINOS
UNIDADE ACADÊMICA DE GRADUAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUIS GUSTAVO SIMIONI CENTELEGHE

PARALLEL MONTE CARLO TREE SEARCH IN GENERAL VIDEO GAME PLAYING

São Leopoldo
2019

Luis Gustavo Simioni Centeleghe

PARALLEL MONTE CARLO TREE SEARCH IN GENERAL VIDEO GAME PLAYING

Artigo apresentado como requisito parcial para obtenção do título de Bacharel em Ciência da Computação, pelo Curso de Ciência da Computação da Universidade do Vale do Rio dos Sinos (UNISINOS)

Orientador(a): Dr. Sandro José Rigo

São Leopoldo
2019

PARALLEL MONTE CARLO TREE SEARCH IN GENERAL VIDEO GAME PLAYING

Luis Gustavo Simioni Centeleghe[1]
Sandro José Rigo[2]

Abstract:

Monte Carlo Tree Search (MCTS) parallelization is one of the many possible enhancements for MCTS algorithms. Since MCTS parallelization methods were first proposed in 2008 by Cazenave and Jouandeau (2008), many researchers have been evaluating them using a variety of testing methodologies and games. However, no work has been done on evaluating these methods in the rather new area of *General Video Game Playing* (GVGP), an area that challenges the creation of agents that are able to play any video game even without prior knowledge about the video game they are going to play. To address this gap, this paper proposes the implementation and evaluation of the three main MCTS parallelization methods (*Leaf*, *Root*, and *Tree Parallelization*) as agents of the *General Video Game AI* framework, a popular framework for GVGP agents evaluation. It is important to notice that this paper is not focused on comparing the parallel MCTS agents to other existing GVGP agents, but rather on exploring how the MCTS parallelization methods compare between themselves. This paper also presents a testing methodology for evaluating these agents, which is based on a set of three experiments focused on different aspects of the parallel MCTS algorithms. These experiments were executed using 32 hyper-threads of a computer equipped with two Intel *Xeon* E5-2620v4 processors. In these experiments, the overall best results were achieved by the *root parallelization* method using the *sum* merging technique and the UCT's *sigma* value of $\sqrt{2}$. However, it is also discussed in the paper some scenarios where other configurations performed better.

Keywords: Monte Carlo Tree Search, Parallel Monte Carlo Tree Search, General Video Game Playing, General Video Game AI

1 INTRODUCTION

Monte Carlo Tree Search (MCTS) is considered the state-of-the-art algorithm for game tree searching in scenarios where no proper evaluation function exists for intermediate game states, making it very suitable for games such as *Hex*, *Go* or games where the domain is unknown for the playing agents, such as the ones used for *General Game Playing* and *General Video Game Playing*. (BROWNE et al., 2012).

Many enhancements for MCTS have been proposed since it was first introduced by Coulom (2006) in 2006. Among these enhancements, we have the MCTS parallelization, which was proposed in 2008 by Cazenave and Jouandeau (2008) through two different approaches, called *Leaf Parallelization* and *Root Parallelization*. In the same year, Chaslot, Winands and Herik (2008) introduced a third approach called *Tree Parallelization*. Together, these three approaches are considered the main methods for MCTS parallelization. (BROWNE et al., 2012).

[1]Computer Science student at Universidade do Vale do Rio dos Sinos. E-mail: lcenteleghe@edu.unisinos.br
[2]Advisor, Researcher at Unisinos' PPGCA, Post-Ph.D fellow at Friedrich-Alexander Universitat, Ph.D in Computer Science at UFRGS (2008). E-mail: rigo@unisinos.br

Since these parallel approaches were first presented, many researchers have been evaluating them using a variety of games, such as *Reversi* (ROCKI; SUDA, 2011), *Hex* (MIRSOLEIMANI et al., 2015), *Mango* (CHASLOT; WINANDS; HERIK, 2008), and also General Game Playing (ŚWIECHOWSKI; MAŃDZIUK, 2016).

However, no work has been done on evaluating how these approaches perform in the rather new area of General Video Game Playing (GVGP), an area that challenges the creation of agents that are able to play any video game even without prior knowledge about the video game they are going to play, having to infer at runtime how to play it. Thus making it a very suitable candidate for parallel MCTS algorithms.

Due to this lack of research on parallel MCTS for GVGP, we present in this paper three parallel GVGP agents and an evaluation of their performance. Each of the agents implements one of the three main parallel MCTS approaches: *Leaf Parallelization*, *Root Parallelization* and *Tree Parallelization*.

To evaluate these agents we created them to be compatible with the *Single Player Planing Track* of the *General Video Game AI* framework (GVG-AI), a framework that provides an environment for creation and evaluation of GVGP agents, and is one of the most important projects created for GVGP research. The Single Player Planing Track provides more than 115 different games, what allowed us to observe how the agents behave when performing in many different environments. (PEREZ-LIEBANA et al., 2018).

The evaluation of the agents was performed using a set of three different experiments: the first one is a general performance analysis, which is focused on evaluating how the agents compare to each other in terms of performance, and how they compare to a synchronous MCTS agent. The second experiment is focused on comparing merging techniques for root parallelization. And the third experiment is focused on analyzing the impact of the UCT's sigma constant in root parallelization. All these experiments were executed using a computer equipped with two Intel *Xeon* E5-2620v4 processors, which allowed us to run tests using up to 32 hyper-threads.

The results of these experiments are the main contribution of this paper, since they are the first results of parallel MCTS applied to General Video Game Playing. In these experiments, the *root parallelization* method using the *sum* merging technique and the UCT's *sigma* value of $\sqrt{2}$ achieved the overall best results. However, there were scenarios where other methods and parameters performed better. This paper describes and discusses these scenarios and the reasons we believe some agents performed better in them.

One important thing to emphasize is that the focus of this paper was not on comparing the parallel MCTS agents to other existing General Video Game Playing agents, but instead on analyzing and comparing the parallel MCTS agents between themselves.

The remainder of this paper is structured in the following way: Section 2 presents the required background information for reading this paper. Section 3 analyses and discusses the related work. After this discussion, Section 4 describes our approach for parallel MCTS in GVGP

and the parallel agents we implemented. Section 5 then describes the experiments we used to evaluate these agents, and Section 6 presents the results of these experiments, which are later discussed in Section 7. Finally, Section 8 concludes the paper.

2 BACKGROUND

In this section, we present the four main concepts pertinent to this paper. Firstly, we describe the Monte Carlo Tree Search technique, followed by one of its many enhanced versions, the Parallel MCTS. After that, we have an introduction to General Video Game Playing and the General Video Game AI framework, a framework for the evaluation of GVGP agents.

2.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a technique widely used for game tree searching. It is very suitable for games where no proper evaluation function exists for intermediate game states, such as *GO*, *Hex* and games where the domain is previously unknown for the playing agent, as the ones used for General Game Playing and General Video Game Playing. (BROWNE et al., 2012).

Instead of using domain-specific evaluation functions, MCTS algorithms use Monte Carlo Simulations (hence the algorithm's name) as a replacement for such functions. A typical example of state evaluation using Monte Carlo Simulations is to select a state and advance it by taking random actions until it reaches a final state, then, this final state is evaluated; if it represents a victory, it yields a high reward, and if it represents a defeat, it yields a low or negative reward. (BROWNE et al., 2012).

2.2 Structure of MCTS

The base structure of a MCTS algorithm is comprised of the following four phases (CHASLOT et al., 2008):

1. Selection: from the root node a child selection policy is applied until an expandable, non-terminal node is reached.

2. Expansion: from the selected node, a new node is added to the tree based on the available actions.

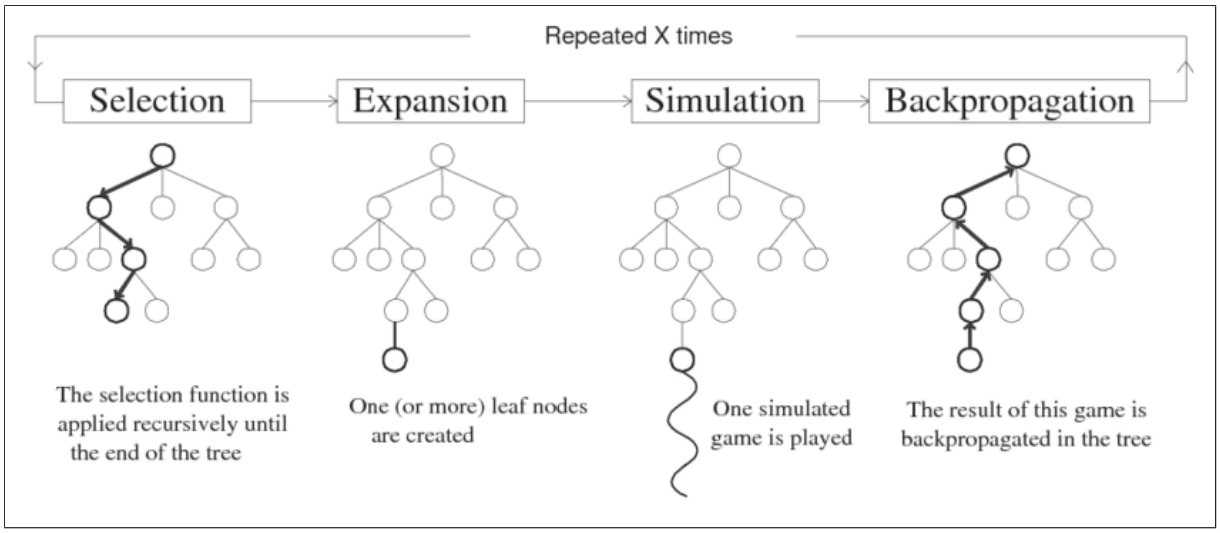
3. Simulation: starting from the state of the newly expanded node a simulation is run by taking random actions until a final state is reached. This final state is then evaluated by a default policy which produces an outcome. This phase is sometimes called *playout phase*.

4. Backpropagation: the simulation results are backpropagated from the newly expanded

node to the root, updating the statistics (number of visits and total reward) of all the nodes along the way.

These four phases are repeated for as long as there is any computational budget (such as time) left. An illustration of this structure is shown in Figure 1.

Figure 1: Monte Carlo Tree Search Scheme



Source: Chaslot et al. (2008)

2.2.1 Upper Confidence Bound Applied for Trees (UCT)

The Upper Confidence Bound Applied for Trees (UCT) is the most popular implementation of the MCTS algorithms family. This algorithm defines all the four phases of the general MCTS structure. However, its most important aspect is the formula it uses in the *selection phase* to select the next node to be expanded once the last selected node has been fully expanded. (BROWNE et al., 2012).

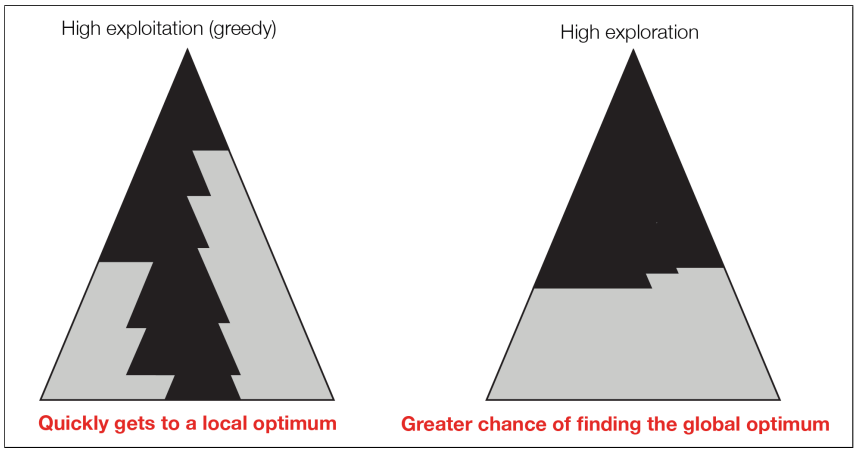
The main objective of the UCT formula is to maintain the balance between exploration and exploitation during the selection phase of the algorithm, avoiding greedily exploration of only the nodes with high simulation reward in the early stages of the search. The UCT formula is defined as follows:

$$v* = \underset{v' \in children\ of\ v}{arg\,max} \left\{ \frac{Q(v')}{N(v')} + C\sqrt{\frac{2\,ln\,N(v)}{N(v')}} \right\}$$

where *v** is the selected node; *v* is the fully-expanded root node; *Q(v)* is the total simulation reward of a node *v*; *N(v)* is the total number of visits to the node *v*; and *C* (also called *sigma*) is a constant that defines to which degree the second component of the formula (called exploration component) is considered over the first component (called exploitation component). So increasing the value of *C* will increase the exploration of the tree, while decreasing it will increase

the exploitation of subtrees with high rewarding root nodes. Figure 2 illustrates the difference between exploitation and exploration. (BROWNE et al., 2012).

Figure 2: Exploitation vs Exploration



Source: Rocki and Suda (2011)

2.3 Parallel Monte Carlo Tree Search

Since each simulation of the MCTS algorithm can be run independently, it is considered a good target for parallelization. (BROWNE et al., 2012). Based on this idea, three major parallelization methods were proposed, as described by Chaslot, Winands and Herik (2008) they are:

A. Leaf Parallelization: Leaf parallelization is the simplest method for MCTS parallelization. The idea here is to run multiple parallel simulations when a new leaf is added to the tree, instead of just one. The results of these multiples runs are them aggregated by the main thread and backpropagated through the tree.

The main advantages of this method are its simplicity (it doesn't even require any mutexes) and the fact that multiple simulations can increase the confidence of the node's statistics. However, this approach suffers from two problems. First, using this approach the tree does not grow faster than using a single-threaded approach, it might even grow slower, since running n simulations using n different threads might take longer on average than running a single simulation. The second problem is that no information is shared between the parallel simulations, and it may cause some adverse effects, for instance: if a simulation with 20 threads is run and 14 of these threads finish faster than the remaining 4 threads with a loss, the other 4 threads will probably also lead to a loss, however, the main thread will have no way to know it, and will still have to wait for them to finish, what might slow down the whole process. (CHASLOT; WINANDS; HERIK, 2008).

B. Root Parallelization: For this method, multiple independent game trees are run using

parallel threads. After the computing budget is over, the main thread merges all the root's children of the parallel trees, usually by summing the nodes statistics (besides of summing, some other merging techniques might be used, as described by Świechowski and Mańdziuk (2016)). These merged statistics are then used to make the final decision and select the best move. The main advantage of this method is that it requires a minimal amount of communication between threads. (CHASLOT; WINANDS; HERIK, 2008).

C. Tree Parallelization: Tree parallelization uses a single tree that is accessed and modified by multiple threads, enabling the threads to run parallel simulations in different leaf nodes. It differs from leaf parallelization, where multiple simulations are run for the same leaf node.

Since a single shared tree is used for this method, mutexes are needed to avoid data corruption. Two forms of such mutexes were proposed by Chaslot, Winands and Herik (2008): the first one is a *global mutex* that locks the access to the whole tree so only one thread can read and modify it at a time, while the other threads can only run simulations on leaf nodes. This method works well if the simulation phase is long enough that the tree access does not become a bottleneck; however, this is not always the case. To avoid such bottleneck *local mutexes* can be used, theses mutexes lock the access to a single tree node whenever a thread is visiting it. When using this method the nodes are locked and unlocked with a high frequency during the search, so this solution requires the use of a fast-access mutex, such as spinlocks, to produce better results. (CHASLOT; WINANDS; HERIK, 2008).

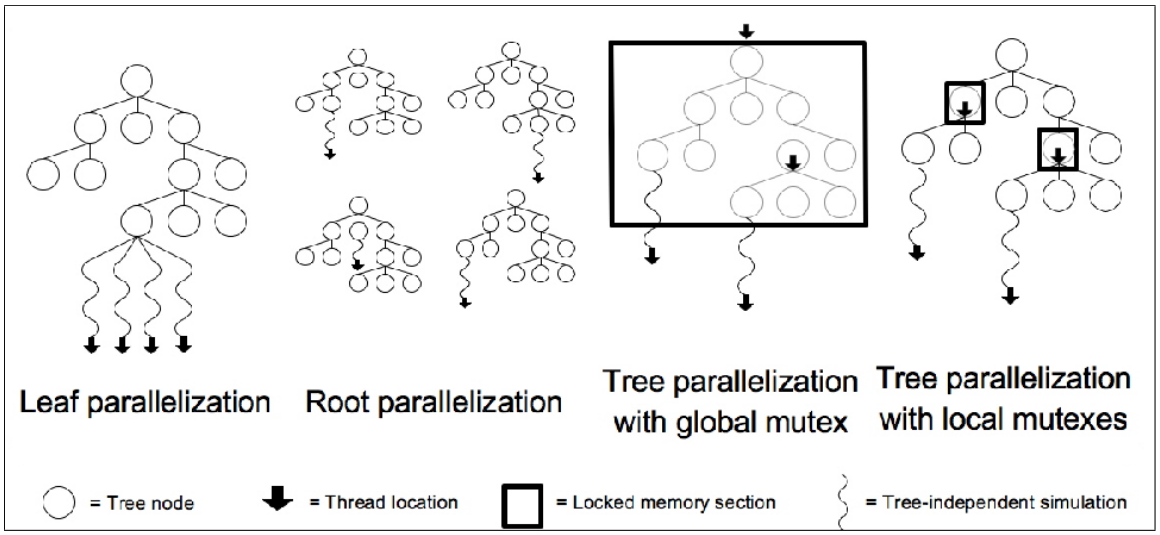
One problem that occurs with tree parallelization is that since all the threads start the search from the same root node, they might all traverse the tree in a very similar way, reducing the exploration of the tree to a small subset of itself. To mitigate this problem, Coulom (2006) suggests assigning a 'virtual loss' to a node each time a thread visits it, making it less valuable for the next threads, and thus making these next threads less prone to explore the same subtree, creating a better balance between the exploration and exploitation of the tree. This virtual loss is removed from the node once the thread that generated it starts to backpropagate the results of the simulation it ran at the leaf node. These parallelization methods are illustrated in Figure 3.

2.4 General Game Playing and General Video Game Playing

The use of board games as a benchmark for Artificial Intelligence algorithms has been of interest to researchers for a long time, especially after 1997 when the IBM supercomputer *Deep Blue* was victorious against the then world chess champion Garry Kasparov. Such games are interesting for AI research because they are a kind of problem that requires high cognitive ability and at the same time offer a reliable way to evaluate the quality of algorithms created to solve them. (CAMPBELL; HOANE JR.; HSU, 2002).

Although the use of specific board games such as Chess and Go has proved to be a great challenge, researchers began to notice that focusing on specific games could limit the development of algorithms that are able to find generic solutions, a characteristic that would make these

Figure 3: Parallel Monte Carlo Tree Search Methods



Source: Chaslot, Winands and Herik (2008)

algorithms better able to extrapolate the world of games to be also used for real problems.

To address this issue the concept of *General Game Playing* (GGP) was proposed, which consists of challenges where the player algorithm does not know the rules of the game, being forced to infer by itself how the game works. Such a challenge also transforms the General Game Playing into a great benchmark for algorithms focused on creating a *Strong Artificial Intelligence* (i.e., a hypothetical Artificial Intelligence so advanced that is able to achieve 'self-awareness'). (GENESERETH; LOVE; PELL, 2005).

In addition to the use of board games as a benchmark for AI, in recent years the idea of using video games for this purpose has emerged, notably appearing through benchmarks and competitions involving the classic games Pac-Man (ROHLFSHAGEN et al., 2018) and Mario (KARAKOVSKIY; TOGELIUS, 2012). However, as with board games benchmarks, there was also a need for a benchmark able to evaluate how algorithms would behave in scenarios where they did not know the rules of the video game they are playing, and for that reason the same concept of General Game Playing was created for video games, which is called *General Video Game Playing*. (LEVINE et al., 2014).

2.5 The General Video Game AI Framework

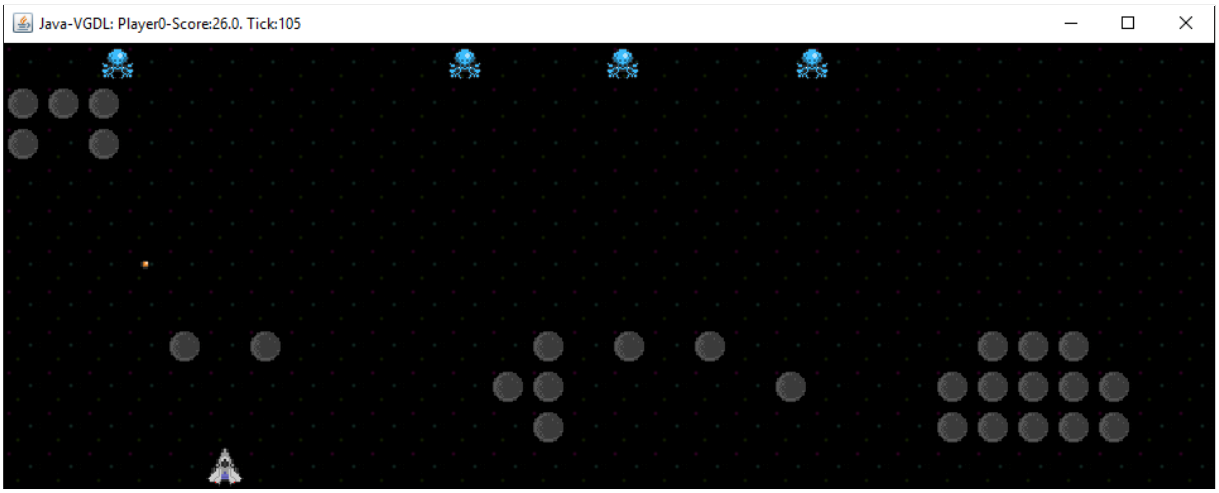
In order to grow the idea of General Video Game Playing, in 2014 the *General Video Game AI (GVGAI)* competition framework was created, which is a framework able to evaluate the quality of agents designed to play any generic video game, even without prior knowledge on the video game they will play. (PEREZ-LIEBANA et al., 2016). When it was first proposed in 2014 the framework included only the *single player planning track*, but since 2017 it now

counts on five different competition tracks, which are described below:

A. Single Player Planning: For this track, the agents receive a forward model that allows them to do run simulations on it and gather feedback on how these simulations affected the model. Based on this model the agents have 40ms to decide the next action they will take. The agents also have a limited boot time of 1s for this track. (PEREZ-LIEBANA et al., 2018).

This track counts with more than 115 different games, being one example the game *Aliens* (shown in Figure 4), a game similar to the traditional game *Space Invaders*. In this game the player wins if it kills all the aliens, it loses if any alien touches it, it scores 1 point for each alien or protective structure destroyed, and it loses 1 point each time an alien's projectile hits it. (PEREZ-LIEBANA et al., 2016).

Figure 4: *Aliens* game of the *GVGAI*, a game similar to the traditional *Space Invaders*



Source: Generated by the author using the GVGAI framework.

B. Two-Player Planning: This track is similar to the Single Player Planning track, except that for this track another intelligent agent also participates in the game, being considered as a major challenge of this track the fact that the agent does not know if the game in question is cooperative or competitive. (PEREZ-LIEBANA et al., 2018).

C. Learning Track: This track is focused on machine learning. For this track, the agents do not receive a simulation model. Instead, they receive a time of 5 min for training in a game before having to play it competitively. (PEREZ-LIEBANA et al., 2018).

D. *Level Generation*: The challenge of this track is to create a general level generation algorithm for a game based only on the knowledge of how the sprites of the game interact with each other. The last version of this track, which took place at the IEEE CIG 2017, was canceled because only one algorithm was submitted. (PEREZ-LIEBANA et al., 2018).

E. *Rule Generation*: The goal of this track is to create an algorithm that generates interaction rules sets for game sprites and game termination rules. Its first version was scheduled to happen during the IEEE CIG 2017, but no one submitted algorithms for this track. (PEREZ-LIEBANA

et al., 2018).

Except for the learning track (which accepts agents in Python), all agents must be developed in Java and follow specific framework interfaces. Currently, agents can be submitted for evaluation through the website *www.gvgai.net*. After being evaluated, these agents are placed in a general ranking which is displayed on the same website. Additionally, the competition has legs that occur during major events such as the IEEE Conference on Computational Intelligence and Games (IEEE CIG) and the IEEE Congress on Evolutionary Computation (CEC). It's important to notice that this competition do not allow the use of parallelism. So even thought we used the GVGAI framework to evaluate the agents we developed for this paper, we did not create them with the intention of competing against another agents created for the competition.(PEREZ-LIEBANA et al., 2018).

3 RELATED WORK

In this section we present five works that focused on evaluating different parallel MCTS methods. We present here which methods they evaluated, which games were used, how they evaluated the methods, and the conclusion they have reached after their evaluation.

To select these papers we first merged the results returned by the scientific citation indexing services Scopus, Web of Science and Google Scholar when searched by the following query: ("PARALLEL"OR "CONCURRENT"OR "THREAD") AND ("MONTE-CARLO TREE SEARCH"OR "MONTE CARLO TREE SEARCH"). From the results of this search we tried to select any work where a parallel MCTS method was applied to General Video Game Playing, but none was found. Then we selected papers that applied parallel MCTS to General Game Playing; two were found. Lastly, we selected some papers that evaluated parallel MCTS approaches in any way, giving priority to the ones who used more distinct ways and parameters to evaluate the algorithms.

3.1 Parallel Monte Carlo Tree Search from Multi-core to Many-core Processors (MIRSOLEIMANI et al., 2015)

Mirsoleimani et al. (2015) used 24 cores of two Intel *Xeon* E5-2596v2 and 61 cores of a Intel *Xeon* Phi 7120P co-processor to evaluate two of the main parallel MCTS methods: root parallelization and tree parallelization. They evaluated these algorithms based on a custom implementation of the game *Hex*, and using two different speedup-measures: *playout speedup* and *strength speedup*. Playout speedup is defined by the increase on the number of simulations per second as the number of threads are increased, and strength speedup is defined by the improvement on the quality of the game playing as more threads are used, which is measured by the win percentage of the algorithm.

The authors conclude their work with two significant results. First, they conclude that the

use of locks for tree parallelization is not a limiting factor for up to 16 threads on a Xeon CPU and 64 threads on a Xeon Phi. And second, they conclude that the Xeon Phi has its performance limited by the sequential part of the MCTS, based on this they suggest that more promising results might be found if a game with a vectorizable simulation phase is used.

3.2 A Parallel General Game Player (MÉHAT; CAZENAVE, 2011)

The work done by Méhat and Cazenave (2011) is of great interest for this work because they evaluated an algorithm developed for General Game Playing, which makes it closely related to the General *Video* Game Player presented in this paper.

For their experiments, Méhat and Cazenave (2011) used eight different board games provided by the General Game Playing framework and a parallel MCTS implementation based on root parallelization. Even though they tested only with root parallelization, they tested it with four different merging techniques: *Sum*, *Sum10*, *Best* and *Raw*.

The *Sum* function sums the statistics of all the root's children of the parallel trees weighted by the total number of simulations executed by the tree. The *Sum10* function is similar to the *Sum*, but instead of summing all the nodes it sums only the ten best nodes from the parallel trees. The *Best* function does not actually merge any node; it simply selects the move that leads to the best overall node between all the parallel trees' root's children. And different from the other functions, the *Raw* function does not merge nor considers the number of visits of each node; it only merges the average score of the nodes.

Their experiments result in 50% of the games having better results when using the parallel version of the MCTS, and 50% of them having the same or worst results when compared to the sequential version. They also conclude that the best merging functions between the ones they evaluate are *Sum* and *Sum10*, which have similar results when compared to each other.

3.3 Parallel Monte Carlo Tree Search Scalability Discussion (ROCKI; SUDA, 2011)

Rocki and Suda (2011) discuss in their work which factors affect the scalability of the MCTS algorithm when running on multiple CPUs/GPUs using root parallelization.

For their tests, they used the games Reversi and SameGame running in the Japanese supercomputer *TSUBAME*, what allowed them to execute experiments with as many as 1024 CPUs and 256 GPUs (each one able to run 1344 threads). Besides from only testing with different numbers of threads, the authors also investigated on how changing the constant C (sigma) of the UCT formula (which defines the exploitation/exploration ratio) and the problem size affects the performance and scalability of the algorithm.

By experimenting with different C values, they observed that a higher exploitation ratio results in better scaling. They believe this result is due to the fact that a higher exploration ratio creates parallel trees that are too similar to each other, while a higher exploitation ratio

promotes more diversified tree structures among the parallel trees. Moreover, by testing with multiple problem sizes, they concluded that the algorithm only scales well if the number of threads is increased as the problem size grows, which shows that the parallel MCTS with root parallelization is a *weak scaling* algorithm.

3.4 Parallel Monte-Carlo Tree Search (CHASLOT; WINANDS; HERIK, 2008)

In this work, Chaslot, Winands and Herik (2008) introduce the third major known parallelization method for MCTS, the tree parallelization method (described in Section 2.3). They then used the games Mango and Go to test the performance of this new method, and how it compares to the two previous major known methods: leaf parallelization and root parallelization, which were introduced by Cazenave and Jouandeau (2008).

Based on the results of their experiments, the authors concluded that the leaf parallelization might be the weakest of the methods and root parallelization the best. In their tests, the leaf parallelization was able to achieve a strength speedup (speedup based on the quality of playing) of 2.6 for 16 threads, while the root parallelization was able to achieve a strength speedup of 14.9 for the same 16 threads. Their new method, tree parallelization, stood between the two others with an 8.5 strength speedup when using 16 threads. However, this speedup was only possible when they used virtual loss and local mutexes instead of global mutexes. The use of local mutexes made it possible for the algorithm to double the number of simulations per second.

The authors believe that one major reason why root parallelization had the best results is that it prevents the MCTS from staying too long in local optima, since the parallelization is able to improve the balance between exploration and exploitation given by UCT formula. However, they conclude the article by saying that further improvements in the balance of the UCT formula might make the tree parallelization the best choice for MCTS parallelization.

3.5 A hybrid approach to parallelization of Monte Carlo Tree Search in General Game Playing (ŚWIECHOWSKI; MAŃDZIUK, 2016)

Świechowski and Mańdziuk (2016) introduced a new MCTS parallelization method called *Limited Root-Tree Parallelization*. This method combines root parallelization with tree parallelization by running multiple distinct parallel trees in different machines (aspect from root parallelization) and using tree parallelization within each machine. Besides from only mixing the two methods, this new method also uses certain techniques to narrow down the search to only specific subsets of action on remote nodes, which proves to be beneficial in games with high branching factor and repetitive movement patterns.

As a mean of testing this new method, the authors also used the General Game Playing framework, testing their algorithm against nine games the framework provides. The results

of these experiments led the authors to believe that their new hybrid method is more suitable for General Game Playing than the tree parallelization alone and the root parallelization alone, except in cases where a small number of computers is used, in such cases the authors still believe that root parallelization alone might be the best option.

3.6 General Analysis

The reviewed related works are summarized in Table 1 according to the four following criteria:

- *Evaluated Parallelization Methods*: which parallelization methods were evaluated.
- *Different Games Used for Testing*: the number of distinct games that were used to evaluate the algorithms.
- *Uses GGP*: whether General Game Playing was used for evaluating the algorithms.
- *Uses GVGP*: whether General Video Game Playing was used for evaluating the algorithms.

Analyzing this table we can notice that only two works evaluated parallel MCTS approaches using General Game Playing (these were the only ones found in the literature), and none used General Video Game Playing.

This lack of parallel MCTS implementations for GVGP can be considered a research gap. Therefore, implementing and testing parallel MCTS approaches using GVGP can contribute both to the General Video Game Playing research and to the Parallel Monte Carlo Tree Search research. The contribution to the GVGP research is due to the fact that it would be the first GVGP player to use a parallel MCTS approach, introducing a new kind of agent to the area. And the main contribution to the parallel MCTS research comes from the fact that the General Video Game Playing framework has more than a hundred different games that can be used for testing parallel MCTS approaches, making it possible to understand even further how these approaches perform in many different environments.

Table 1: Related Work

	Mirsoleimani et al. (2015)	**Méhat and Cazenave (2011)**	**Rocki and Suda (2011)**	**Chaslot, Winands and Herik (2008)**	**Świechowski and Mańdziuk (2016)**
Evaluated Parallelization Methods	Tree Parallelization Root Parallelization	Root Parallelization	Root Parallelization	Leaf Parallelization Root Parallelization Tree Parallelization	Tree Parallelization Limited Root-Tree Parallelization
Different Games Used for Testing	1	8	2	2	9
Uses GGP	No	Yes	No	No	Yes
Uses GVGP	No	No	No	No	No

Source: created by the author.

4 PARALLEL MCTS IN GENERAL VIDEO GAME PLAYING

Considering the lack of research on Parallel Monte Carlo Tree Search approaches for General Video Game Playing described in Section 3, we decided to implement and evaluate four different GVGP agents: three parallel agents, each one using one of the three main parallel MCTS approaches (Leaf, Root, and Tree Parallelization), and one sequential agent used for performance comparison. These agents were created to be compatible with the Single Player Planning Track of the General Video Game AI Framework (described in Section 2.5). The framework served as our main tool for evaluating the agents. The Single Player Planning Track provides more than 115 different games, which allowed us to observe how the agents' performance and behaviour is impacted by many different environments.

In the next sections we present our parallel General Video Game Playing agents implementations, our experimental setup, and the results obtained from these experiments.

4.1 Our Parallel General Video Game Playing Agents

To conform with the GVAI framework, we implemented the agents using the Java language (version 1.8) [3]. Each of the parallel agents uses a different parallel MCTS approach; however, all of them share a core MCTS algorithm, which is slightly modified to accommodate these different parallel MCTS approaches. This core algorithm was based on the algorithm described by Chaslot et al. (2008) and is summarized in pseudocode in Algorithm 1.

In this algorithm, the *selection* and *expansion* phases are mixed in a single function called *TreePolicy*, which uses the UCT formula (described in Section 2.2.1) for balancing exploration and exploitation during the search. The value used for the C (*sigma*) constant of the UCT formula is specified for each of the experiments we performed (described in section 5).

The *simulation* phase is done by the *DefaultPolicy* function, which works by taking random actions until a terminal state or a max depth is reached. If a terminal state is reached, the reward is calculated based on whether it represents a win or a loss, otherwise, the current game score is used as the reward.

The *backpropagation* phase is done in a pretty straightforward way by the *Backup* function, which updates the *number of visits* and *total reward* of all the nodes between the last selected node and the root node.

The sequential agent uses this algorithm as is, while the parallel agents use this algorithm with small modifications, which are described in the next sections.

[3] Implementations available at https://github.com/LCenteleghe/Parallel-MCTS-GVGP-Agents/

Algorithm 1 Core MCTS algorithm used by the agents

function SEARCH(s_0) ▷ s_0: initial state
  $v_0 \leftarrow$ new node with state s_0 ▷ v_0: root node
  while there is computational budget left **do**
    $v_s \leftarrow$ TREEPOLICY(v_0) ▷ v_s: selected node
    $r \leftarrow$ DEFAULTPOLICY($s(v_s)$) ▷ r: reward ▷ $s(v)$: state of v
    BACKUP(v_s, r)
  return a(SELECTCHILDWITHMOSTVISITS(v_0)) ▷ $a(v)$: $action$ that led to v
function TREEPOLICY(v)
  while v is non-terminal **do**
    if v is not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ SELECTBESTCHILD(v)
  return v
function EXPAND(v)
  $a \leftarrow$ next untried action available from v
  $s' \leftarrow f(s(v), a)$ ▷ $f(s, a)$: new state of s when taking action a
  $v' \leftarrow$ new node with $s(v')$ = s' and $a(v')$ = a
  add v' as new child of v
  return v'
function SELECTBESTCHILD(v)
  return $\underset{v' \in children\ of\ v}{arg\,max} \left\{ \frac{Q(v')}{N(v')} + C\sqrt{\frac{2\,ln\,N(v)}{N(v')}} \right\}$ ▷ UCT formula
function DEFAULTPOLICY(s)
  $s' \leftarrow$ RUNRANDOMPLAYOUT(s)
  return EVALUATE(s')
function RUNRANDOMPLAYOUT(s)
  $s' \leftarrow s$
  while s' is non-terminal **and** $max\ depth$ is not reached **do**
    a $\leftarrow$ select available action from s' uniformly at random
    $s' \leftarrow f(s', a)$
  return s'
function EVALUATE(s)
  if s is terminal **and** represents a win **then**
    return $huge\ positive\ value$
  else if s is terminal **and** represents a $loss$ **then**
    return $huge\ negative\ value$
  else
    return $current\ game\ score$
function BACKUP(v, r)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$ ▷ $N(v)$: number of visits to v
    $Q(v) \leftarrow Q(v) + r$ ▷ $Q(v)$: total reward of v
    $v \leftarrow parentOf(v)$

4.1.1 Leaf Parallelization Agent

The difference between this agent's algorithm and the core algorithm (Algorithm 1) resides in the *DefaultPolicy* function (defined in line 23 of Algorithm 1). Instead of running only a single simulation, it runs many parallel simulations and then aggregates the rewards of all these simulations by summing them. This version of the *DefaultPolicy* is shown in Algorithm 2.

Algorithm 2 DefaultPolicy used by the Leaf Parallelization Agent

```
1: function DEFAULTPOLICY(s)
2:     R ← {}                                          ▷ R: set of rewards
3:     for each available thread do
4:         s' ← RUNRANDOMPLAYOUT(s)        } in parallel
5:         r ← EVALUATE(s')                }
6:         R = R ∪ {r}                     }
7: return Σ_{r∈R} r
```

4.1.2 Root Parallelization Agent

This agent's algorithm differs from the core algorithm (Algorithm 1) on the way it implements the *Search* function (defined in line 1 of Algorithm 1). Instead of building only one single tree, this agent builds many independent parallel trees, then after all the computational budget is over it merges all the tree's root's children into one single tree (as described in Algorithm 3).

Algorithm 3 Search method used by the Root Parallelization Agent

```
1: function SEARCH(s_0)                                  ▷ s_0: initial state
2:     T ← {}                                            ▷ T: set of trees' roots
3:     for each available thread do
4:         v_0 ← new node with state s_0                 ▷ v_0: root node
5:         while there is computational budget left do   } in parallel
6:             v_s ← TREEPOLICY(v_0)                     }
7:             r ← DEFAULTPOLICY(s(v_s))                 }
8:             BACKUP(v_s, r)                            }
9:         T = T ∪ {v_0}                                 }
10:    v' ← MERGEALL(T)                                  ▷ v': root of all trees merged
11:    return a(SELECTCHILDWITHMOSTVISITS(v'))           ▷ a(v): action that led to v
```

Many techniques might be used to execute the tree merging. In this work, we experimented with three different techniques: *Sum*, *Best*, and *Raw*. These techniques are based on the ones experimented by Méhat and Cazenave (2011) and Świechowski and Mańdziuk (2016), but for this work they were slightly modified in order to work properly with the GVGAI framework.

The *Best* technique creates a tree with nodes containing the max ("best") value found for the total reward (*Q(v)*) and the total number of visits (*N(v)*) between all parallel trees. This technique consequently makes the search algorithm for root parallelization (Algorithm 3) to select the best node between all nodes of all parallel tree. Algorithm 4 describes this technique.

Algorithm 4 *Best* technique for Tree Merging

function MERGEALL(T) ▷ T: set of trees' roots
  $v'_r \leftarrow$ new node with *no state* ▷ v'_r: root node of the new tree to be created
  $V_c \leftarrow \bigcup_{v \in T} childrenOf(v)$ ▷ V_c: first level children of nodes in T
  for each set V_a of nodes from V_c with same source action a **do**
    $mn \leftarrow \max_{v \in V_a} N(v)$, $mq \leftarrow \max_{v \in V_a} Q(v)$
    $v' \leftarrow$ new node with $N(v') = mn$, $Q(v') = mq$, and $a(v') = a$
    add v' as new child of v'_r
  return v'_r

The *Sum* technique consists of summing the total reward and the total number of visits to a node weighted by the total number of simulations done by the tree in comparison to the total number of simulations done by all the parallel trees, thus raising the significance of the trees which were able to execute more simulations. The pseudocode for this merging technique is defined in Algorithm 5.

Algorithm 5 *Sum* technique for Tree Merging

function MERGEALL(T) ▷ T: set of trees' roots
  $v'_r \leftarrow$ new node with *no state* ▷ v'_r: root node of the new tree to be created
  $t_s \leftarrow \sum_{v \in T} N(v)$ ▷ t_s: total number of simulations
  $V_c \leftarrow \bigcup_{v \in T} childrenOf(v)$ ▷ V_c: first level children of nodes in T
  for each set V_a of nodes from V_c with same source action a **do**
    $wn \leftarrow \sum_{v \in V_a} N(v) \frac{N(parentOf(v))}{t_s}$ ▷ wn: weighted number of visits sum
    $wq \leftarrow \sum_{v \in V_a} Q(v) \frac{N(parentOf(v))}{t_s}$ ▷ wq: weighted total reward sum
    $v' \leftarrow$ new node with $N(v') = wn$, $Q(v') = wq$, and $a(v') = a$
    add v' as new child of v'_r
  return v'_r

In distinction to the *Sum* technique, the *Raw* technique calculates the average total reward and the average number of visits of the nodes without weighting them by the number of simulations done by the tree. This technique is described in Algorithm 6.

For each of the experiments described in section 5 we specify which technique of the ones described above was used.

Algorithm 6 *Raw* technique for Tree Merging

1: **function** MERGEALL(T) ▷ T: set of trees' roots
2: $v'_r \leftarrow$ new node with *no state* ▷ v'_r: root node of the new tree to be created
3: $V_c \leftarrow \bigcup_{v \in T} childrenOf(v)$ ▷ V_c: first level children of nodes in T
4: **for each** set V_a of nodes from V_c with same source action a **do**
5: $an \leftarrow \frac{\sum_{v \in V_a} N(v)}{sizeOf(T)}$, $aq \leftarrow \frac{\sum_{v \in V_a} Q(v)}{sizeOf(T)}$
6: $v' \leftarrow$ new node with $N(v')$ = an, $Q(v')$ = aq, and $a(v')$ = a
7: add v' as new child of v'_r
8: **return** v'_r

4.1.3 Tree Parallelization Agent

The distinction between the algorithm used by this agent and the core algorithm (Algorithm 1) resides in the *Search* function (defined in line 1 of Algorithm 1). The difference here is that instead of having a single main thread running synchronously the *TreePolicy*, the *DefaultPolicy* and the *Backup* function; this agent uses the main thread only for the *TreePolicy*, and then delegates to another thread (from a pool of threads) the task of applying the *DefaultPolicy* and backing up the results of this function, thus enabling many *DefaultPolicy* and *Backup* functions to run in parallel.

To avoid data corruption due to the fact that some nodes might be accessed concurrently by multiple threads, we implemented all the functions that modify any of the nodes' properties as atomic functions by using the Java keyword *synchronized* on them, thus creating effective local mutexes on the nodes.

Another important point to be noticed in this version is that it breaks down the *Backup* function into two separate functions, one to back up the number of visits (*BackupNumberOfVisits*) and another one to backup the total reward (*BackupTotalReward*). The main thread calls the *BackupNumberOfVisits* function before it delegates the *DefaultPolicy* and (*BackupTotalReward*) to another thread. This causes all the nodes between the selected node and the root node to immediately suffer a 'virtual loss' once they are selected by the *TreePolicy*, due to the way the UCT formula works (see 2.2.1).

This 'virtual loss' makes these nodes less prone to be selected again by the *TreePolicy* while their *DefaultPolicy* is not calculated and backed up by the asynchronous worker thread. Once the total reward is backed-up, the temporary increase in the number of visits cannot be considered as a 'virtual loss' anymore.

The idea of a 'virtual loss' is suggested by Coulom (2006) as a way to avoid the algorithm from exploring only a small subset of the search tree. Without it, the *TreePolicy* would probably keep selecting almost the same nodes while the worker threads had not backed up their results. This version of the *Search* function is described in Algorithm 7.

Algorithm 7 Search method used by the Tree Parallelization Agent

```
1: function SEARCH(s_0)                                   ▷ s_0: initial state
2:     v_0 ← new node with state s_0                      ▷ v_0: root node
3:     while there is computational budget left do
4:         v_s ← TREEPOLICY(v_0)
5:         BACKUPNUMBEROFVISITS(v_s)
6:         r ← DEFAULTPOLICY(s(v_s))      } runs asynchronously on next available thread
7:         BACKUPTOTALREWARD(v_s, r)      }
8:     return a(SELECTCHILDWITHMOSTVISITS(v_0))   ▷ a(v): action that led to v
```

5 EXPERIMENTAL SETUP

To evaluate our agents, we created a set of three different experiments using the games available on the *Single Player Planning Track* of the GVGAI framework. Each of these experiments is focused on evaluating a distinct aspect of the agents. The next sections describe the general experimental setup we used for all the experiments, the metrics we collect, and then a definition of each experiment we executed.

5.1 General Experimental Setup

All the experiments were performed on a computer equipped with 2 Intel *Xeon* E5-2620v4 running at 2.10GHz (3.0GHz with Intel's Turbo Boost) and 126GB of physical memory. Each of these two CPUs has 8 cores and 2 hyper-threads per core, summing up to a total of 32 hyper-threads available for processing.

In order to obtain the experimental results in a reasonable amount of time, we imposed a limit of 40ms for the agents to choose their next action for each step of a game (this is the same amount of time used by the GVGAI for competitions (PEREZ-LIEBANA et al., 2018)).

5.2 Evaluation Metrics

For each of the experiments we executed, we collected and computed metrics which were considered important in order to properly evaluate the agents. Each of these metrics is described below:

- *Win Rate*: percentage of victories over the number of games played. This is the primary indicator of quality/performance for an agent.
- *Strength Speedup*: defines the improvement in playing quality of the agent when compared to the sequential agent. It is calculated as the division of the *number of victories* of the agent under evaluation by the *number of victories* of the the sequential agent.
- *Playout Speedup*: measures the improvement in execution time based on the number of simulations per second. It is calculated as the division of the number of *simulations per*

second of the agent under evaluation by the number of *simulations per second* of the sequential agent. This metric is especially useful for analyzing how the increase in the number of simulations reflects into a better agent in terms of strength speedup.

Win Rate is one of the main metrics for measuring MCTS algorithms' strength. It is used by authors such as Rocki and Suda (2011); Mirsoleimani et al. (2015); and Chaslot, Winands and Herik (2008); while *Strength Speedup* and *Playout Speedup* are metrics used by both Mirsoleimani et al. (2015) and Chaslot, Winands and Herik (2008) as a way to compare the improvement of parallel MCTS implementations when compared to sequential implementations, and also to analyze how the increase in the number of simulations reflects into a more powerful agent.

5.3 Experiments and Testing Parameters

In order to evaluate our agents, we carried out three different experiments. For all these experiments we ran the agents against 116 games of the GVGAI framework using 2, 4, 8, 16 and 32 threads. We executed these runs between 15 and 30 times, in order to lower our statistical error rate. We could not run them more times due time restrictions, since each run of 116 game takes around 1:30h to finish. The details and parameters used for each experiment are described bellow:

Experiment A: General Performance Analysis: this is our main experiment, it is focused on analyzing the general performance of the agents, both individually and when compared to each other. For this experiment we used the *sum* merging technique for the root parallel agent, since it is the method which yielded the best results on *Experiment B*; and for the UCT's sigma (C) constant we used the value of $\sqrt{2}$, the same one used by the sample MCTS agent of the GVGAI framework and the one which achieved the best results on *Experiment C*. (PEREZ-LIEBANA et al., 2018).

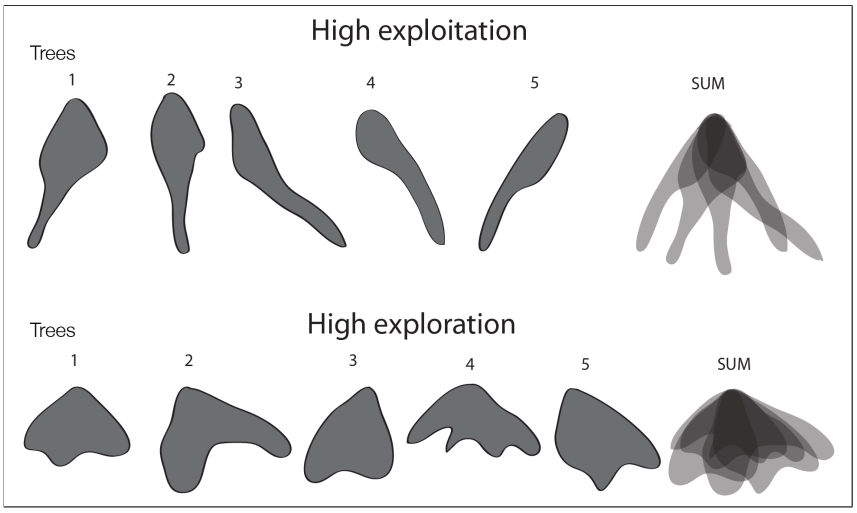
Experiment B: Comparison of Merging Techniques for Root Parallelization: the objective of this experiment is to evaluate and compare the three different merging techniques for root parallelization described in section 4.1.2, namely *Raw*, *Sum*, and *Best*, which are the same ones evaluated by Méhat and Cazenave (2011) and Świechowski and Mańdziuk (2016). For this experiment, we also used the value of $\sqrt{2}$ for the UCT's sigma (C) constant.

Experiment C: Impact UCT's Sigma in Scalability of Root Parallelization: As discussed in section 2.2.1, the sigma (C) constant of the UCT formula defines how much the second component of the formula (exploration component) is considered over the first component (exploitation component). It implies that as the value of C increases, the *exploration* of the tree increases, and as the value of C decreases, the *exploitation* of the tree increases.

When analyzing this change of C in terms of root parallelization, we believe that in most of the cases increasing the value of C will generate a more homogeneous set of parallel trees (due to high exploration), while decreasing it will generate a more diverse set (due to high exploitation). This difference is illustrated in Figure 5.

Therefore, the goal of this experiment is to investigate how the change in the *C* value changes the results obtained by the root parallel agent. To do so, we ran experiments with the root parallel agent using the *sum* merging technique (since it achieved the best results in *Experiment B*) and the values of $\sqrt{2}$, $\frac{\sqrt{2}}{2}$, and $2\sqrt{2}$ for the sigma *C* constant. Our base value $\sqrt{2}$ is based on the value used by the sample MCTS agent of the GVGAI framework. (PEREZ-LIEBANA et al., 2018).

Figure 5: Exploitation vs Exploration in Parallel Trees



Source: Rocki and Suda (2011)

6 EXPERIMENTAL RESULTS

This section solely presents the results we obtained from our experiments. These results are analyzed and discussed in section 7.

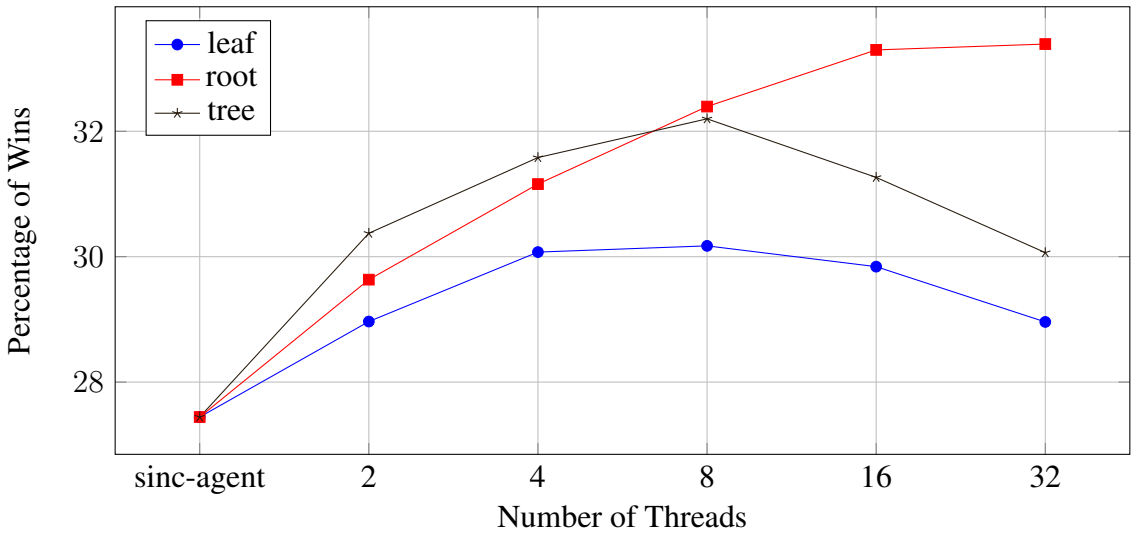
6.1 Experiment A: General Performance Analysis

The objective of this experiment was to analyze and compare the leaf, root, and tree parallel agents, and how they compare to the synchronous agent.

The main results of this experiment are presented in the graph shown in Figure 6, which shows the *win rate* for each of the parallel agents, and the *win rate* for the synchronous agent on the first column.

We can see in these results that all the parallel agents achieved better win rates than the synchronous agents for all number of threads. When compared between themselves, the parallel agents do not show any clear winner. We can see that the leaf parallel agent presented worse results than the other parallel agents in all cases. The root parallel agent achieved the best results for 2 and 4 threads, while the root parallel agent achieved the best results for 8, 16, and

Figure 6: Win Rate for the Agents

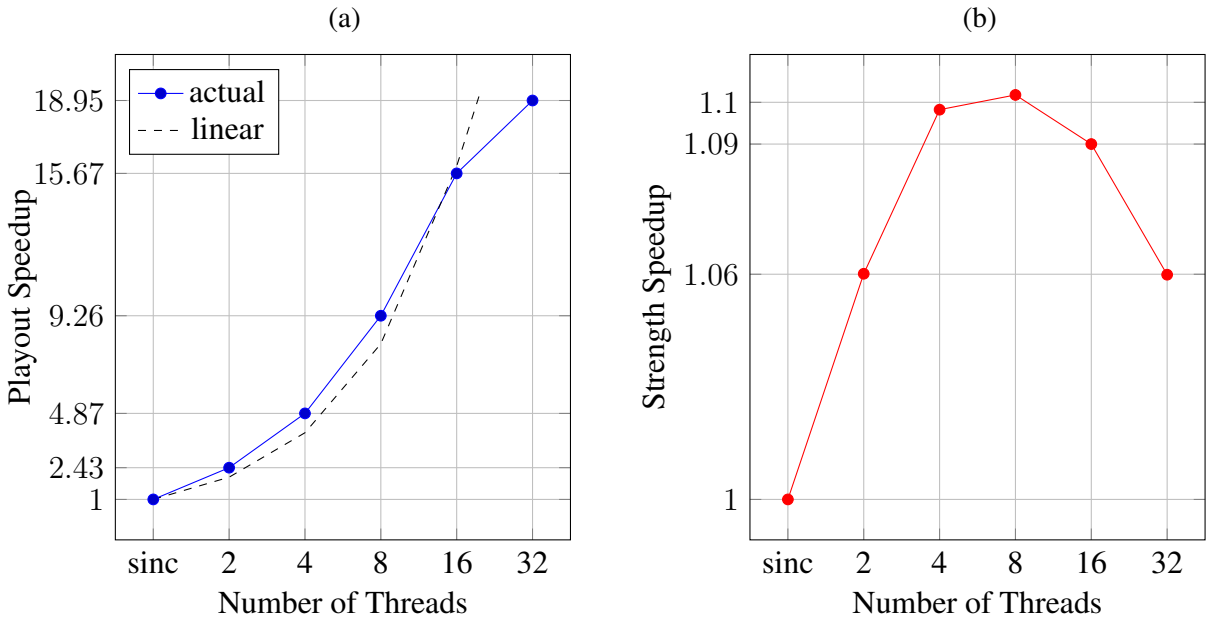


Source: Created by the author.

32 threads, and also the best overall win rate between all agents when using 32 threads.

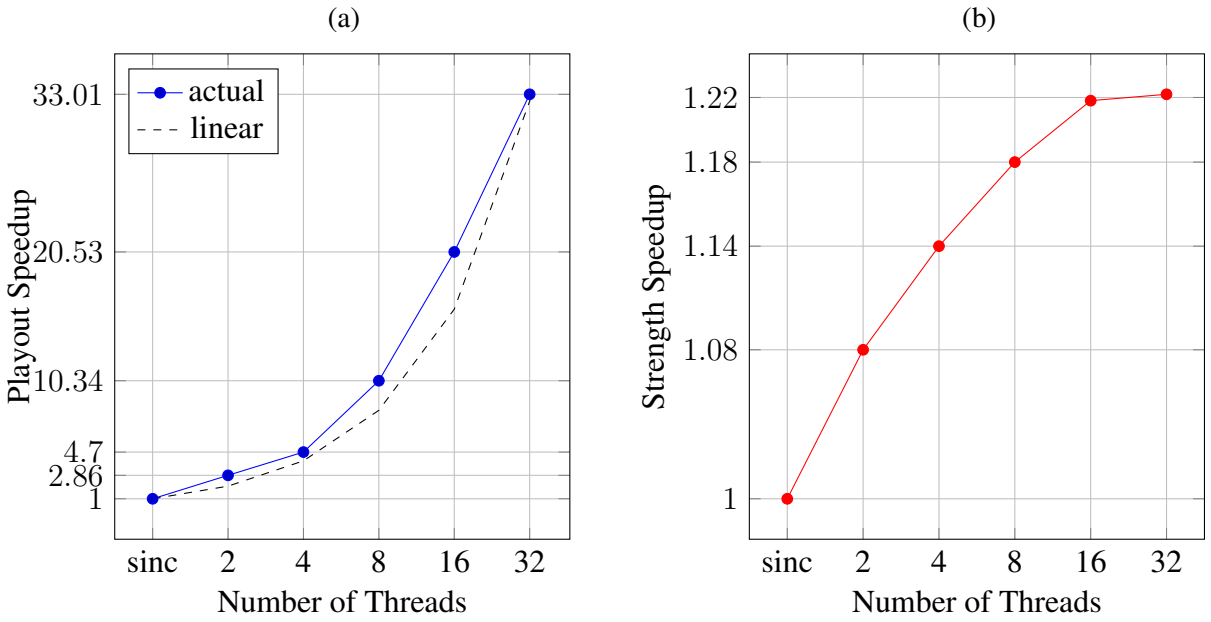
Two other metrics collected in this experiment were the *playout speedup* and *strength speedup* of the agents. These metrics are shown in the graphs *a* and *b* of Figure 7 (leaf parallel agent), Figure 8 (root parallel agent), and Figure 9 (tree parallel agent). The line labeled *linear* in graph *a* indicates the minimum playout speedup any agent has to achieve if it is able to scale perfectly in terms of simulations per second.

Figure 7: Playout Speedup (a) and Strength Speedup (b) for the Leaf Parallel Agent



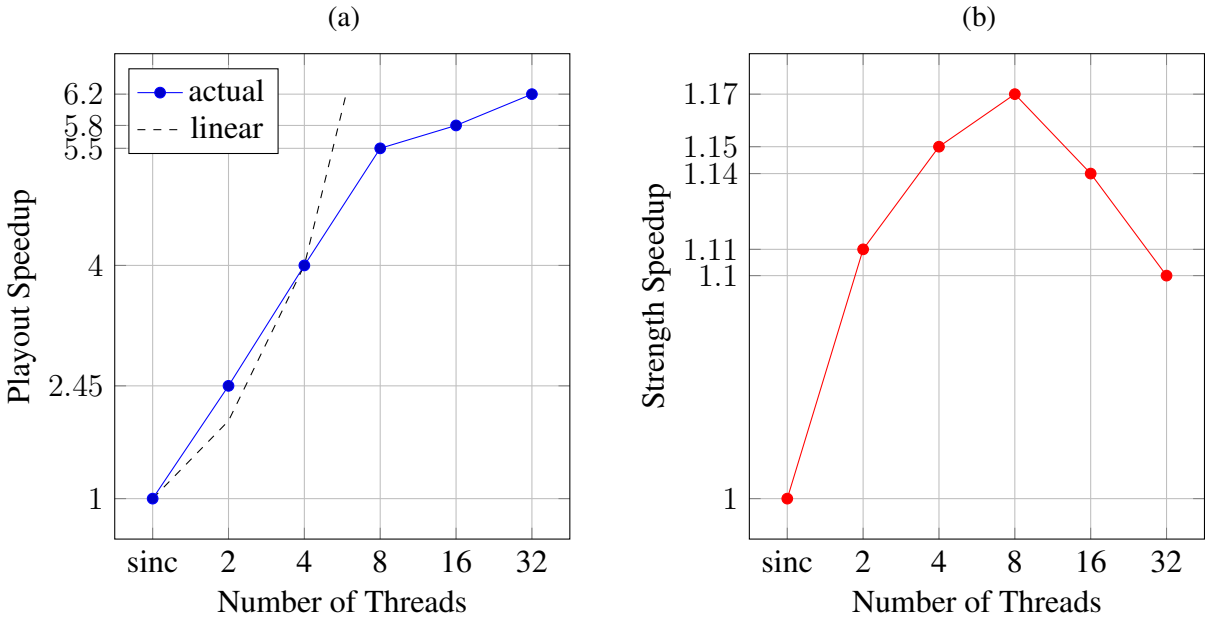
Source: Created by the author.

Figure 8: Playout Speedup (a) and Strength Speedup (b) for the Root Parallel Agent



Source: Created by the author.

Figure 9: Playout Speedup (a) and Strength Speedup (b) for the Tree Parallel Agent



Source: Created by the author.

These results for *playout speedup* show that only the root parallel agent was able to achieve above than linear speedup for all the numbers of threads, while the leaf parallel agent was able to achieve it only for up to 16 threads, and the tree parallel agent only for up to 4 threads. The *strength speedup* graph shown alongside the *playout speedup* graph is presented as a way to demonstrate how changes in *playout speedup* reflect in *strength speedup*.

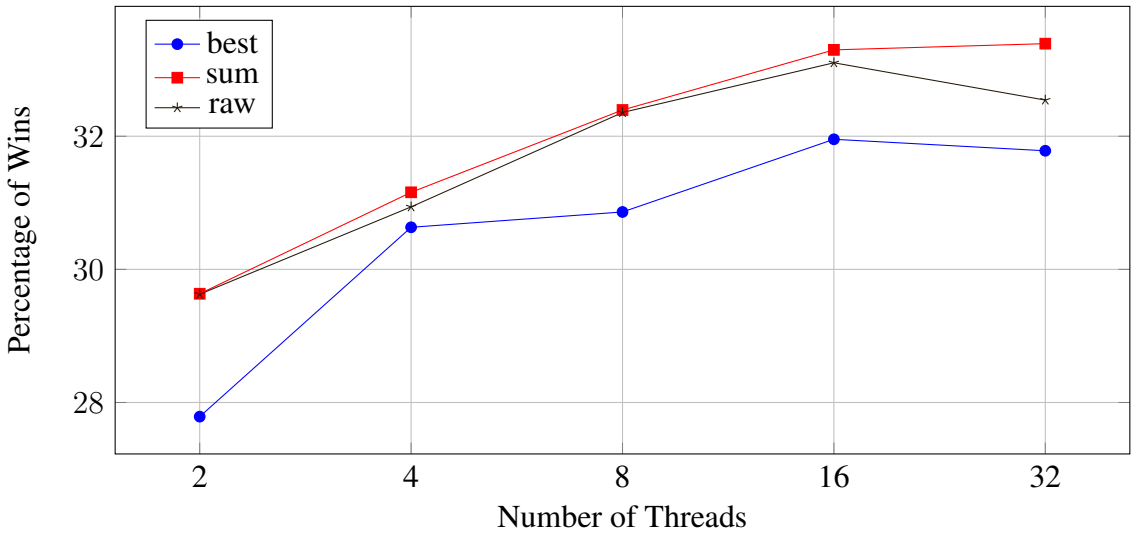
6.2 Experiment B: Comparison of Merging Techniques for Root Parallelization

The goal of this experiment was to compare the *raw*, *sum*, and *best* techniques for root parallelization in terms of *Win Rate* and *Playout Speedup*.

The results for *win rate* are shown in the graph presented in Figure 10, where we can see that the *raw* and *sum* technique achieved almost the same win rates, while the *best* technique achieved the lowest win rates in all scenarios.

We can also see in these results an improvement in win rate for all the techniques for up to 16 threads, with a small decrease for *raw* and *best* when using 32 threads.

Figure 10: Win Rate for each Root Parallelization's Merging Technique

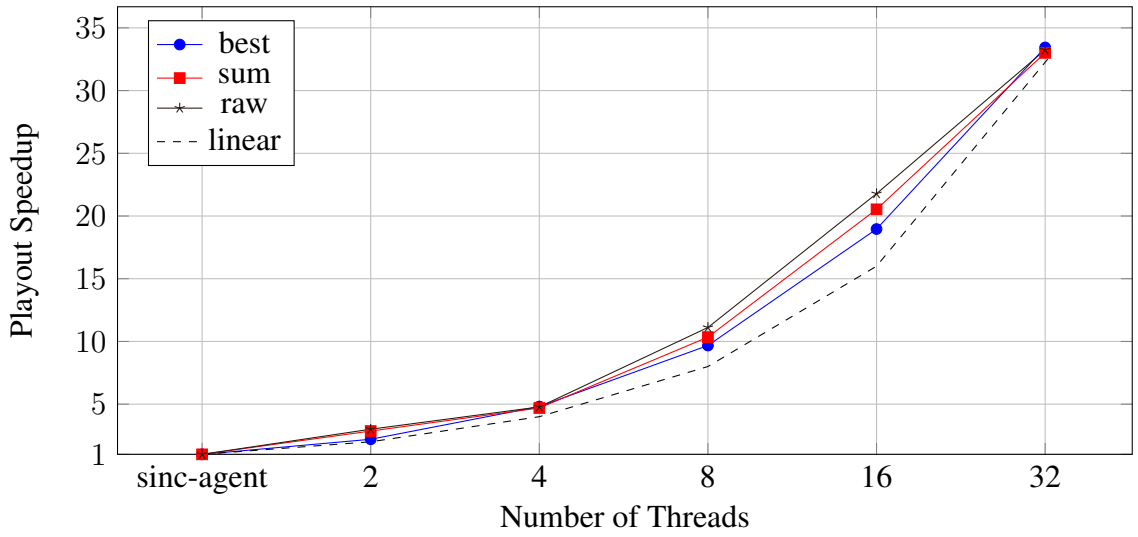


Source: Created by the author.

The results for *Playout Speed* are shown in the graph presented in Figure 11. Theses results show almost no difference for playout speedup between the merging techniques, except a slight difference between them when using 2, 8 and 16 threads.

Based on the graph, we can also notice that the playout speedup for all the agents was no worse than a linear speedup for all numbers of threads.

Figure 11: Playout Speedup for each Root Parallelization's Merging Techniques

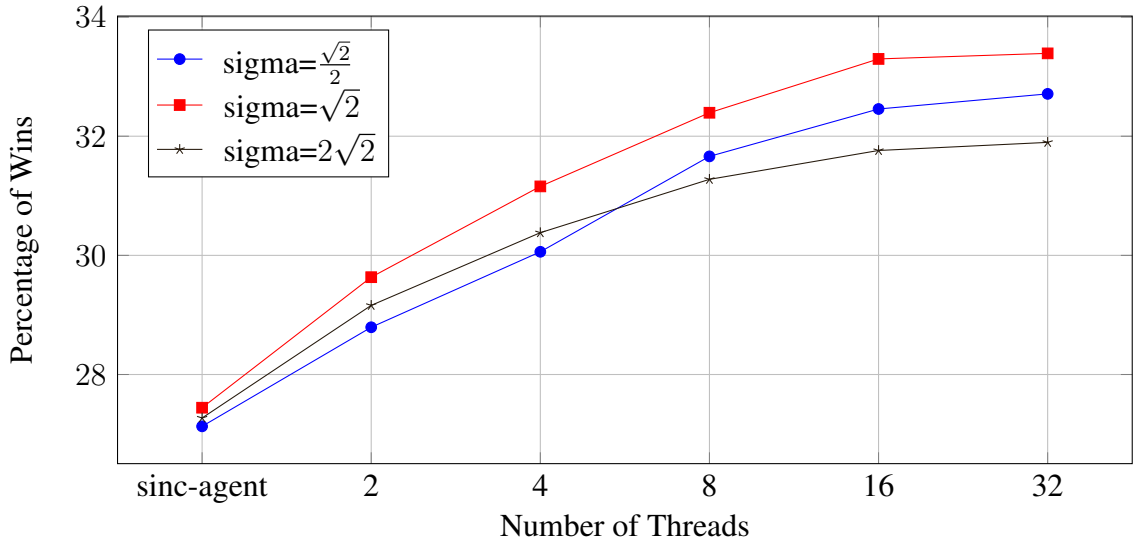


Source: Created by the author.

6.3 Experiment C: Impact of UCT's Sigma in the Scalability of Root Parallelization

The results for the impact of the UCT's sigma constant in root parallelization are shown in terms of win rate in the graph presented in Figure 12.

Figure 12: Win Rate for Root Parallelization with different Sigma Values



Source: Created by the author.

In this graph, we can see that the sigma value of $\sqrt{2}$ achieved the best results in all scenarios, while the value of $2\sqrt{2}$ achieved the second-best results for 2 and 4 threads, and the value of $\frac{\sqrt{2}}{2}$ achieved the second-best results for 8, 16, and 32 threads.

The first column of the graph shows the results obtained by the synchronous agent, so we can see the impact of the UCT's sigma constant in the MCTS algorithm when no parallelization is used.

7 DISCUSSION

In this section, we analyze and discuss the results presented in section 6 for each of the experiments we executed.

7.1 Experiment A: General Performance Analysis

7.1.1 Win Rate Analysis

Win Rate is the primary indicator of performance for an agent, so the results discussed in this section are of great importance for comparing the overall performance of the agents.

As we can see by the results presented in the graph shown in Figure 6, all the agents performed better than the synchronous agents, however we have a tie in terms of general performance between the tree parallel agent (which performed better with 2 and 4 threads), and the root parallel agents (which performed better with 8, 16, and 32 threads). We speculate that this might be due to the fact that root parallelization requires a minimal amount of communication to work (as stated by Chaslot, Winands and Herik (2008)). Thus the tree parallel agent performs better when using a small number of threads, which requires less communication, and the root parallel agent outperforms it when more threads are used, and more communication is required.

Even though the root parallel agent did not perform better than the other agents for all the cases, it was the agent which achieved the overall best win rates and best scalability, since it performed better than the other agents when using higher numbers of threads, so it turned out being the best way of paralleling MCTS given our experimental setup. And the leaf parallel agent turned out being the weakest agents in all scenarios.

This overall result is consistent with the results found by Chaslot, Winands and Herik (2008), who tested the same three parallel methods we tested but using General Game Playing, and they also considered root parallelization as the overall most powerful parallelization method and leaf parallelization the weakest method, given their experimental setup.

However, this overall result is not entirely consistent with the results reported by Mirsoleimani et al. (2015), who tested the root and tree parallelization methods using a custom implementation of the game *Hex*. In their results, they reported better overall win rates for root parallelization when using an Intel *Xeon Phi* 7120P co-processor, but better overall win rates for tree parallelization when using an Intel *Xeon* E5-2596v2 CPU. These results, allied with the fact that the *Xeon* E5-2596v2 CPU has a communication-to-compute ratio 30 times lower the *Xeon Phi* 7120P co-processor reinforces our speculation that tree parallelization performed

worst than root parallelization when working with higher numbers of threads only due communication overhead.

These reported facts lead us to conclude that even though the root parallel agent achieved an overall better performance than the other agents in our experimental setup, the tree parallel agent might surpass it in the future as the communication latency of multiprocessors reduces, making tree parallelization become the best choice for MCTS in General Video Game Playing.

7.1.2 Playout and Strength Speedup Analysis

Our main objective in analyzing playout speedup was too see if there is any direct correlation between playout speedup and strength speedup. However, as we can see by the graphs *a* and *b* of Figures 7, 8, and 9 there is no direct correlation between these two speedup measurements. This lack of correlation is emphasized by the fact that the tree parallel agent achieved better results than the leaf parallel agent in terms of strength speedup while its max playout speedup was three times lower than the one achieved by the leaf parallel agent, leading us to conclude that the way the agents use the extra simulations provided by parallelization is way more important than the raw number of executed simulations.

Another important point to notice is that both the tree and leaf parallel agents presented a decay in terms of strength speedup with 16 and 32 threads. It is unclear for us the reasons for this result for leaf parallelization since it does not present a decrease in playout speed as the strength speedup decreases. However, in the case of tree parallelization, we believe this decrease in strength speedup is due to the local locks used in the tree. This assumption is backed by the fact when using 16 and 32 threads, the playout speedup of the agent starts to stagnate.

7.2 Experiment B: Comparison of Merging Techniques for Root Parallelization

The results for root parallelization merging techniques presented the *sum* merging technique as the overall best technique (Figure 10). This result is directly in line with the findings reported by both Méhat and Cazenave (2011) and Świechowski and Mańdziuk (2016), who also reported the *sum* technique as the best technique in their experiments using General Game Playing, showing that the quality of the *sum* merging technique extends beyond General Game Playing to include also General Video Game Playing.

When analyzing the merging techniques in terms of *playout speedup* (Figure 11), we see almost no difference between the merging techniques. We believe that the slight difference between them when using 2, 4, and 8 threads might be due to the fact that the *raw* technique is computationally simpler than the *sum* and *best* techniques, which reduces its overhead when merging the trees, thus allowing more time for some extra simulations.

7.3 Experiment C: Impact of UCT's Sigma in Scalability of Root Parallelization

By analyzing the graph presented in Figure 12 we can see that the sigma value of $\sqrt{2}$ achieved the best results for all numbers of threads, while the value of $2\sqrt{2}$ achieved the second-best results for 2 and 4 threads, and the value of $\frac{\sqrt{2}}{2}$ achieved the second-best results when using 8, 16, and 32 threads.

If we ignore the results for $\sqrt{2}$, the results for $2\sqrt{2}$ and $\frac{\sqrt{2}}{2}$ already give us some insights on how the sigma value impacts the scalability of root parallelization. These results show that a high value for sigma ($2\sqrt{2}$) worked better only when a low number of threads were used, but as soon as the number of threads was increased, the value of $\frac{\sqrt{2}}{2}$ achieved better results. These results are in line with the findings of Rocki and Suda (2011), who reported their best results for the lowest sigma value they tested when more than 2 threads were used. They believe that lower values lead to better results because a more diverse set of trees is created between the parallel trees when lower sigma values are used.

However, contrary to their findings, the best value for sigma we found ($\sqrt{2}$) is not the lowest value we experimented, but the value in between the lowest and the highest value, what lead us to conclude that there is a lower limit for how low the sigma value can be before it generates a negative impact on performance.

8 CONCLUSION

Monte Carlo Tree Search (MCTS) is one of the most popular algorithms for game tree searching in scenarios where a proper evaluation function for intermediate game states is nonexistent or hard to create. (BROWNE et al., 2012).

Parallelization of MCTS is one of the many enhancements proposed for the algorithm. The three main methods for MCTS parallelization are *Leaf Parallelization*, *Root Parallelization*, and *Tree Parallelization*. These methods have been evaluated by several researchers using many different games, such as *Mango* (CHASLOT; WINANDS; HERIK, 2008), *Hex* (MIRSOLEIMANI et al., 2015), and games of the General Game Playing framework (ŚWIECHOWSKI; MAŃDZIUK, 2016). However, no work had been done on evaluating how these methods perform in the rather new area of General Video Game Playing.

To address this research gap, we implemented and evaluated these three main MCTS parallelization approaches as agents of the *Single Player Planning Track* of the *General Video Game AI* framework. The agents were evaluated using a set of three different experiments, the first one focused on general performance analysis, the second one on comparing merging techniques for root parallelization, and the third experiment was focused on analyzing the impact of the UCT's sigma constant in the scalability of root parallelization.

In these experiments the *root parallelization* method using the *sum* merging technique and the UCT's *sigma* value of $\sqrt{2}$ achieved the overall best results.

The experiments on general performance also allowed us to conclude that *tree parallelization* might surpass *root parallelization* in the future as the communication latency of multiprocessors reduces. And the experiments on the impact of the UCT's sigma constant in root parallelization lead us to conclude that there is a lower limit for the sigma value before it starts to generate a negative impact, which is probably higher than $\frac{\sqrt{2}}{2}$ considering the results we obtained .

As future work, we suggest evaluating how the parallel MCTS agents perform against other existing General Video Game Playing agents. We also suggest the execution of experiments using a broader range of values for the UCT's sigma constant, in order to find with more precision when the sigma value starts to generate a negative impact on the scalability of root parallelization.

REFERENCES

BROWNE, C. B. et al. A Survey of Monte Carlo Tree Search Methods. **IEEE Transactions on Computational Intelligence and AI in Games**, [S.l.], v. 4, n. 1, p. 1–43, 3 2012.

CAMPBELL, M.; HOANE JR., a. J.; HSU, F.-h. Deep Blue. **Artificial Intelligence**, [S.l.], v. 134, n. 1-2, p. 57–83, 2002.

CAZENAVE, T.; JOUANDEAU, N. A parallel Monte-Carlo tree search algorithm. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, [S.l.], v. 5131 LNCS, p. 72–80, 2008.

CHASLOT, G. et al. Monte-carlo tree search: a new framework for game ai1. **Belgian/Netherlands Artificial Intelligence Conference**, [S.l.], p. 389–390, 2008.

CHASLOT, G. M. J. B.; WINANDS, M. H. M.; HERIK, H. J. van den. Parallel Monte-Carlo Tree Search. In: **Frontiers in artificial intelligence and applications**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. v. 227, p. 60–71.

COULOM, R. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In: INTERNATIONAL CONFERENCE ON COMPUTERS AND GAMES, 2006, Berlin, Heidelberg. **Proceedings...** Springer Berlin Heidelberg, 2006. p. 72–89.

GENESERETH, M.; LOVE, N.; PELL, B. General Game Playing: overview of the aaai competition. **AI Magazin**, [S.l.], v. 26, n. 2, p. 62–72, 2005.

KARAKOVSKIY, S.; TOGELIUS, J. The Mario AI Benchmark and Competitions. **IEEE Transactions on Computational Intelligence and AI in Games**, [S.l.], v. 4, n. 1, p. 55–67, 3 2012.

LEVINE, J. et al. General Video Game Playing. **Artificial and Computational Intelligence in Games**, [S.l.], v. 6, p. 77–83, 2014.

MÉHAT, J.; CAZENAVE, T. A Parallel General Game Player. **KI - Künstliche Intelligenz**, [S.l.], v. 25, n. 1, p. 43–47, 2011.

MIRSOLEIMANI, S. A. et al. Parallel Monte Carlo Tree Search from Multi-core to Many-core Processors. **Proceedings - 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2015**, [S.l.], v. 3, p. 77–83, 2015.

PEREZ-LIEBANA, D. et al. The 2014 General Video Game Playing Competition. **IEEE Transactions on Computational Intelligence and AI in Games**, [S.l.], v. 8, n. 3, p. 229–243, 9 2016.

PEREZ-LIEBANA, D. et al. General Video Game AI: a multi-track framework for evaluating agents, games and content generation algorithms. , [S.l.], 2018.

ROCKI, K.; SUDA, R. Parallel Monte Carlo Tree Search Scalability Discussion. In: **Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)**. [S.l.: s.n.], 2011. v. 7106 LNAI, n. Figure 1, p. 452–461.

ROHLFSHAGEN, P. et al. Pac-Man Conquers Academia: two decades of research using a classic arcade game. **IEEE Transactions on Games**, [S.l.], v. 10, n. 3, p. 233–256, 9 2018.

ŚWIECHOWSKI, M.; MAŃDZIUK, J. A hybrid approach to parallelization of Monte Carlo Tree Search in General Game Playing. In: STUDIES IN COMPUTATIONAL INTELLIGENCE, 2016. **Proceedings...** [S.l.: s.n.], 2016. v. 634, p. 199–215.