A Hybrid Approach to Parallelization of Monte Carlo Tree Search in General Game Playing

Maciej Świechowski[1] and Jacek Mańdziuk[23]

[1] Phd Studies at Systems Research Institute, Polish Academy of Sciences, Warsaw, Poland
m.swiechowski@mini.pw.edu.pl
[2] Faculty of Mathematics and Information Science, Warsaw University of Technology, Warsaw, Poland
[3] School of Computer Engineering, Nanyang Technological University, Singapore
j.mandziuk@mini.pw.edu.pl

Abstract. In this paper, we investigate the concept of a parallelization of Monte Carlo Tree Search applied to games. Specifically, we consider General Game Playing framework, which has originated at Stanford University in 2005 and has become one of the most important realizations of the multi-game playing idea. We introduce a novel parallelization method, called Limited Hybrid Root-Tree Parallelization, based on a combination of two existing ones (Root and Tree Parallelization) additionally equipped with a mechanism of limiting actions available during the search process. The proposed approach is evaluated and compared to the non-limited hybrid version counterpart and to the Tree Parallelization method. The advantages over Root Parallelization are derived on a theoretical basis. In the experiments, the proposed method is more effective than Tree Parallelization and also than non-limited hybrid version in certain games.

Keywords: Monte Carlo Tree Search, Upper Confidence Bounds Applied for Trees, General Game Playing, Parallelization, Parallel Computing

1 Introduction

Monte Carlo Tree Search (MCTS) [5] is renowned for being the state-of-the-art algorithm of searching a game tree in a variety of domains. It is particularly useful in complex games with high branching factors such as Go [11], Hex [2], Havannah [27] or Arimaa [26] where no good evaluation function exists. Since the introduction of MCTS to a domain of General Game Playing (GGP) [13] in 2007, it has also become a backbone of almost all the strongest players [25]. GGP deals with creating autonomous agents capable of playing many games with a high level of competence. The term was proposed by Stanford Logic Group in 2005, together with the introduction of the General Game Playing Competition

as the official world championships. Our player, called MINI-Player [22, 24], has been our annual entry to the competition since 2012.

Parallelization is understood as making the program run simultaneously on many computers and/or processing cores. In this paper, we are concerned with the parallelization of the MCTS algorithm in the GGP framework, specifically. The MCTS is relatively easy to parallelize compared to any classic Depth-First Search algorithm [19] (e.g., alpha-beta, min-max, MTD-f) because it contains fewer synchronization points. The synchronization can be less frequent because there is a simulation phase in MCTS, which is usually very time-consuming and isolated, i.e., the game tree does not need to be accessed in this phase. However, running multiple simulations in parallel steps away from the original idea of MCTS where for each iteration of the algorithm there is one simulation. Therefore, parallelization is not only an implementation issue but also a design choice.

The main motivation behind parallelization is to increase implementation efficiency of the Monte Carlo Tree Search algorithm. Naturally, the more simulations are performed the more accurate statistics of actions are collected. In GGP, each game is written in a universal logic language which is very slow to interpret compared to any game-dedicated representation. Utilizing many CPU cores is especially important in the GGP Competition scenario, where every participant runs the program on their own computational facilities. Parallelization in GGP area brings an additional difficulty stemming from the fact that the approach needs to be universally good, i.e. suitable for a variety of games.

This paper is organized as follows. In the next section, we briefly introduce the GGP competition setting and describe our MINI-Player and the MCTS algorithm in more details. In section, 4 related parallelization methods are presented which are the entry points to our method. The following section contains description of the proposed novel method applied in MINI-Player. The last two sections are devoted to results and conclusions, respectively.

2 Background

2.1 General Game Playing

As already mentioned in the introduction, General Game Playing refers to the area of creating autonomous multi-game playing agents. Each game in the GGP framework is defined in the Game Description Language (GDL) [15], which allows for describing any finite, synchronous, deterministic and perfect-information game. Apart from these constraints, games have no limitation, e.g. they can feature any number of players (including only one), can be of various genres (not only board games) can be cooperative, competitive or partially both.

GDL is a first-order logic language based on Datalog [1] and Prolog [4]. It is relatively easy to convert a GDL description to a Prolog program. A complete game state of any game is defined by a set of facts (predicates) which hold true. When designing methods of parallelization, one must keep in mind that the

GDL description must be sent to all remote units in the system. Moreover, many methods require game states to be sent frequently. Although certain compression of the state is possible (e.g., conversion of strings to numbers), the representation is still less compact, in a general case, than a game-specific one [23]. Therefore, not only simulations are slower but also the communication overhead is higher in GGP.

In GGP, the communication takes place through messages sent via HTTP protocol. Players do not communicate with each other directly, but with a special component called the **Game Manager** (GM). The GM can be started locally or provided by the organizers of the GGP Competition. The defined messages are **START**, **PLAY**, **STOP**, **ABORT**, **PING** and **INFO**.

The **START** message is sent by the GM to each player when a game starts. It may look as follows: *(START match1 white ((role white)(role black)....) 40 10)*. The message contains the keyword "START", an identifier of the played match (to differentiate between matches), a role for the player (which must exactly match one of the roles defined in a game), the complete GDL description of a game and a pair of clocks (in seconds). The *START-clock* defines the initial preparation measured from sending the rules until the game starts. Players are expected to respond with "(ready)" before the *START-clock* expires. The *PLAY-clock* defines time available for players to make a move. After the successive *PLAY-clock*, PLAY messages are sent and the game state is updated by the GM. In the GGP competition, the *START-clock* is usually set to somewhere between 20 and 120 seconds whereas the *PLAY-clock* to 10-30 seconds. These relatively low settings significantly limit the possible approaches to creating and parallelizing a GGP agent.

The START message, which is sent only once, is followed by a number of **PLAY** messages at even intervals equal to the *PLAY-clock*. Before each *PLAY-clock* expires, each player has to send their chosen move. When it expires, the GM sends the PLAY message that contains moves chosen by all players (called a joint move) to all the players so they can update their game state accordingly. The GM is also responsible for checking whether the submitted moves are legal in the game. If not, it chooses random ones instead and/or disqualifies the illegally playing players. A play message may look as follows: *(PLAY match1 (move a 1) noop)*.

Technically, the **STOP** message is the last PLAY message with the additional meaning that the game has reached the conclusion. Players should update their state for the last time and check the outcome of the game. The **ABORT** message means that game has been terminated prematurely - either by manual intervention or a failure at the GM side.

The **PING** and **INFO** messages are used to check whether players are online. They should respond with "(available)" or "(busy)".

2.2 MINI-Player

MINI-Player [22, 24] has been designed and implemented as part of the PhD thesis [20] and with the aim of taking part in the official GGP championships.

All of parallelization methods investigated in this paper were implemented in MINI-Player. The key features of the program are:

1. **Monte Carlo Tree Search.** MCTS, which is the main routine of the player, is described in subsection 2.3.
2. **Simulation strategies.** MINI-Player uses seven policies to bias the search in the simulation phase of the MCTS algorithm. The policies are Random, Approximate Goal Evaluation (greedily trying to fulfill a goal condition), History Heuristic (exploiting past good actions), Mobility (maximizing the relative number of available actions), Exploration (prioritizing novel game states during search), Statistical Symbols Counting [16] (dynamically constructed material-inspired evaluation function) and Score (detection of explicitly defined scoring condition in the GDL rules).
3. **Adaptive mechanism of selection of strategies.** The strategies are evaluated dynamically based on their empirical performance. A strategy is assigned to guide a simulation based on its evaluation and a confidence of the evaluation. The better or the less simulated the strategy the higher the probability of choosing it.
4. **Fast GDL interpreter.** We developed a dedicated GDL interpreter [21] which is faster than known Prolog distributions.
5. **Three-time (2012-2014) participation in GGP competition.** In 2012 and 2014, MINI-Player achieved the 7-8th place.

2.3 Monte Carlo Tree Search

The MCTS algorithm is an iterative simulation-based approach to searching the game tree. Each iteration consists of four phases depicted in Figure 1.
Selection: start the search from the root node. Traverse the tree down until reaching a leaf node. In each node, choose the child node with the highest average score. More sophisticated approaches replace the average score with a specialized formula for determining the node to choose.
Expansion: if a state in the leaf node is not terminal choose a continuation which falls out of the tree and allocate a new child node. Typically, in the basic version of the method, just one new node is added per each expansion step.
Simulation: starting from a state associated to the newly expanded node, perform a full game simulation (i.e. to the terminal state).
Backpropagation: fetch the result of the simulated game. Update statistics (scores, visits) of all nodes on the path of simulation starting from the newly expanded node up to the root.
When the time budget is up, an action leading to the highest average score is chosen.

In the selection phase, MINI-Player uses the Upper Confidence Bounds Applied for Trees (UCT) [14] method. The purpose of the algorithm is to maintain a balance between exploration and exploitation in the selection step. Instead of sampling each action uniformly or choosing always the best action so far, the selection of the best action is made as follows:

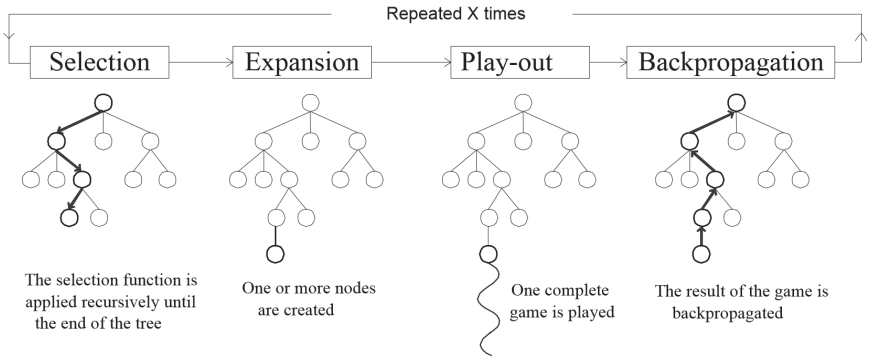

Fig. 1. Four phases of the MCTS algorithm. The figure is reproduced from [8]
.

$$a^* = arg \max_{a \in A(s)} \left\{ Q(s,a) + C\sqrt{\frac{ln\,[N(s)]}{N(s,a)}} \right\} \tag{1}$$

where a - is an action; s - is the current state; $A(s)$ - is a set of actions available in state s; $Q(s,a)$ - is an assessment of performing action a in state s; $N(s)$ - is a number of previous visits to state s; $N(s,a)$ - is a number of times an action a has been sampled in state s; C - is a coefficient defining a degree to which the second component (exploration) is considered.

The first remarkably successful application of MCTS in games referred to Go. A majority of the strongest programs, e.g. MoGo [12] or CrazyStone [9] use variants of MCTS. In contrast to all the variations of min-max alpha-beta search, the MCTS is an aheuristic method. The aheuristic property means that there is no game-specific knowledge (heuristics) required so the method can be applied to a wide selection of problems.

3 Related Methods of Parallelization in GGP

There have been three major methods proposed for the task of parallelizing an MCTS-based player, i.e. Leaf Parallelization (LP) [6], Root Parallelization (RP) [6, 17] and Tree Parallelization (TP) [6, 7, 18]. These methods can also be called At-the-leaves Parallelization, Single-Run Parallelization and Multiple-Runs Parallelization, respectively. Our method is based on two of them, so we decided to dedicate a section for a short review of all the methods.

3.1 Leaf Parallelization

In Leaf Parallelization (LP), there is a single game tree with exclusive access from the master process (a distinguished thread on one of the machines). The

MCTS/UCT algorithm is performed by the master process. When it reaches a leaf node, the state associated to that node is sent to all remote processes. Then, a simulation starting from that state is executed in parallel in each of these processes. The master process waits for the simulations to finish and collects the results. In Leaf Parallelization, only simulation and back-propagation phases differ in comparison to the original (sequential) MCTS formulation. The simulation phase consists of multiple independent simulation starting from the same state, whereas the back-propagation phases updates statistics aggregated from these multiple runs instead of one.

A simple idea and easy implementation are the biggest advantages of the Leaf Parallelization method. However, although many simulations performed from the same node increase the confidence of its statistics, it is not the most effective approach. The original MCTS/UCT algorithm chooses the node to simulate after each simulation, because every time the statistics are updated, a new node might be more suitable for the next simulation. In the LP approach, even the meaningless state has to be simulation at least K times, where K is the chosen number of remote processes. Moreover, the tree does not grow quicker than in the single-threaded MCTS. Figure 2 shows how Leaf Parallelization works.

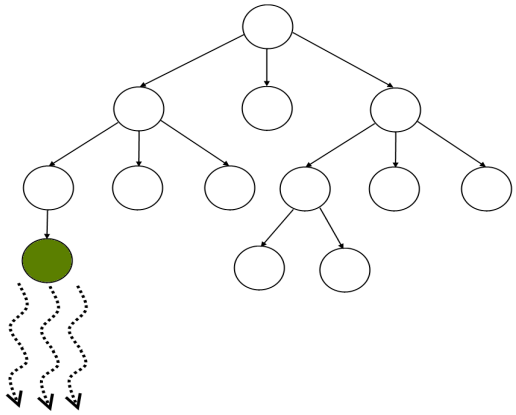

Fig. 2. An illustration of the Leaf Parallelization method with 3 parallel processes. The parallel simulations start from the marked node and the MCTS algorithm waits until all of them are completed.

3.2 Root Parallelization

The main idea of Root Parallelization (RP) is to expand the game trees on each remote process individually and synchronize them occasionally. In the most typical approach, the master process passes raw messages (*START*, *PLAY*, *STOP*, *ABORT*) received from the GM to all connected machines. Each machine acts as

a regular GGP player with the following two exceptions: (1) - they respond to the master node instead of to the GM; (2) - instead of the chosen action, they send MCTS statistics (total score, visits) stored in the top-most level of the game. The top-most level of the game tree contains the nodes related to all possible actions in the current state. The master process performs synchronization by aggregating the gathered statistics to its own game tree to compute the global average scores for actions. The most frequently chosen method of aggregation is a sum but certain other are possible, e.g., a sum of the best K or majority voting.

The biggest advantage of the method is a minimal communication overhead. The statistics gathered from the trees expanded by independent UCT algorithms are more confident but they may also be too overlapping (similar). In overall, the method scales relatively well and was successfully used by a GGP player called Ary [17].

Three variants of aggregating the results were tested: *Best* (select the best evaluated move from a distributed node), *Sum* (sum total scores and total visits and compute a global average scores), *Sum10* (*Sum* performed only for the top ten best evaluated moves) and *Raw* (send only average scores of moves from nodes without weighting by the number of total visits). The best results were reported for *Sum* and *Sum10* with no significant difference between them. Figure 3 shows how Root Parallelization works.

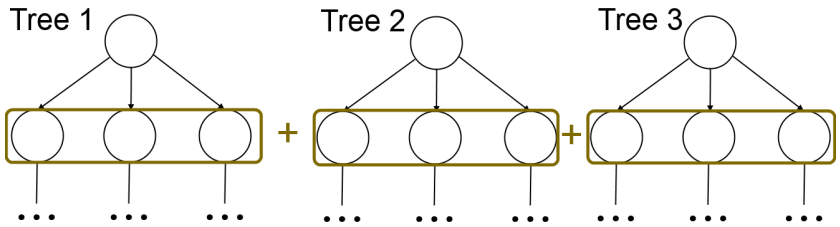

Fig. 3. An illustration of the Root Parallelization method with 3 parallel processes. Each process maintains an independent tree constructed in a sequential fashion. Statistics of the root's children are aggregated during a synchronization point.

3.3 Tree Parallelization

Tree Parallelism (TP) resembles the idea of the sequential MCTS/UCT as closely as possible. There is a single game tree expanded by the master process. Whenever the MCTS/UCT algorithm schedules a new simulation, the master process checks for the first available process which is either a local thread (the preferred case) or a remote process. Once the master process sends the simulation request it proceeds to the next iteration of the MCTS/UCT. The simulation count (visits) of the previously selected node is incremented immediately but the scores

are updated when the simulation is actually completed (therefore, until it happens, it is a virtual loss for all players). If no parallel unit is available, the system will wait or place the simulation request into a waiting queue, depending on the implementation. The advantage of Tree Parallelization over both Root and Leaf Parallelization is that more unique states are visited. The MCTS algorithm converges faster in the Leaf Parallelization approach. However, the communication overhead is significantly higher than in Root Parallelization. Figure 4 shows how Tree Parallelization works.

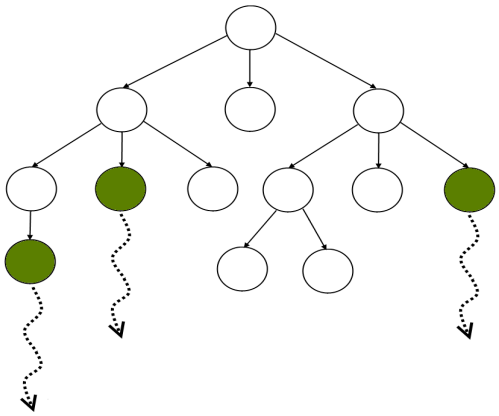

Fig. 4. An illustration of the Tree Parallelization method with 3 parallel processes. When a simulation is finished, statistics are updated in the tree and a new simulation is scheduled based on the currently observed statistics.

4 Our Hybrid Approach: Limited Root-Tree Parallelization

The idea of our method is to combine the best of both words. Within one machine we use Tree Parallelization which is then mixed by Root Parallelization between the machines. TP can be relatively efficiently performed on a single computer, because of the benefits gained from a shared memory. TP on remote machines, however, not only requires network communication but also serialization and deserialization of states which simulations start from. With the shared memory available in one program's instance, all the necessary synchronization can be put into a critical section or even a lock-free fashion can be pursued. The lock-free version [10] assumes that some synchronization errors (often called faulty updates) are possible. The idea is to detect such cases and ignore result from the simulation if a fault was encountered. In our implementation, the safer approach with the lock is used.

Although the shared game tree has to be accessed twice in one iteration of the MCTS algorithm (first during Selection/Expansion and next during Back-Propagation), both phases can reside within the same lock when a smart reordering of the MCTS phases is applied. Figure 5 shows how Hybrid Parallelization works.

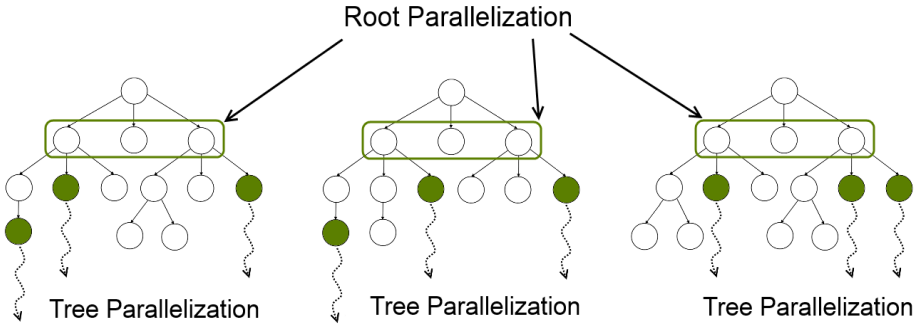

Fig. 5. An illustration of the Hybrid Parallelization method with 3 parallel nodes and 3 parallel processes in each node.

A unique feature of the MINI-Player's parallelization approach is that each remote node operates on a subset of the tree to increase the variety of search. Each remote node (a computer) is assigned a unique identifier $ID \in [1, N]$. The maximum number (N) is globally available to all nodes. On the first possible level in the tree, where MiNI-Player has more than one available action we narrow the search of each remote machine to the portion of the total moves defined by:

$$\left[\frac{ID-1}{N}, \frac{ID+1}{N}\right] \tag{2}$$

These intervals overlap to keep some redundancy in case a machine disconnects. The last interval is wrapped back to the beginning. For instance, if $N = 5$, the intervals are:
$[0, \frac{2}{5}][\frac{1}{5}, \frac{3}{5}][\frac{2}{5}, \frac{4}{5}][\frac{3}{5}, 1]$ and "$[\frac{4}{5}, \frac{6}{5}]$" $= [\frac{4}{5}, 1] \cup [0, \frac{1}{5}]$.
These intervals are used on remote nodes to constrain the available actions to the UCT algorithm in the selection phase. If M is the total number of moves, then only actions with indices from the range as computed by Formula 3 are available to choose.

$$\left[\left\lfloor M * \frac{ID-1}{N} \right\rfloor, \left\lfloor M * \frac{ID+1}{N} \right\rfloor\right] \tag{3}$$

The redundancy caused by overlapping intervals can be further strengthened in two ways:

1. By increasing the overlapping margin i.e. $\left[\frac{ID-k}{N}, \frac{ID+k}{N}\right]$, where $k > 1$.
2. By having multiple machines assigned the same ID. For instance, with 8 machines, N can be set to 4 and each value of ID can be assigned to a pair of machines. In such a case, the same subset of actions will be checked twice due to redundancy. Since the MCTS algorithm is non-deterministic, repeated simulations may increase confidence of the statistics.

Actions of MINI-Player are limited only **in the first node** on each path coming from the **root** in which our player has more than 1 action available. The root is included, so if MINI-Player has more than 1 action in the root then it is the only node with limited actions. There are no limitations of actions chosen for the simulated opponents. There are two sample trees shown in Figure 6 where the mechanism is illustrated. The set of allowed moves is limited only for MINI-Player because in this approach the assessment of actions in distributed programs is fair by means of being evaluated against all possible opponents' actions. If the search was limited to certain subsets of moves for all players, then some of them could have too promising assessment if they happened to be simulated together with only weak actions of the opponents. To solve this issue, each combination of players' actions would have to be simulated by some remote node what vastly increases complexity of the problem.

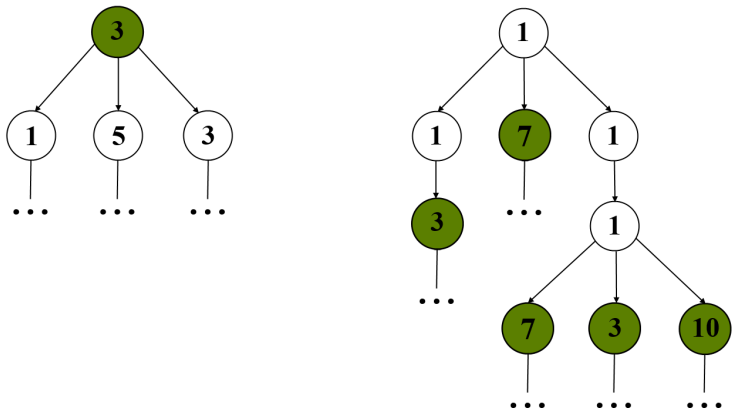

Fig. 6. An illustration of how MINI-Player's actions are limited in the UCT algorithm. The nodes are labeled by the number of legal moves for MINI-Player. Circles marked in green are the first nodes on each path from the root with more than 1 action available, so the limiting takes place there. Branches ending with three dots are not relevant to the algorithm's presentation.

In the proposed parallelization approach, three types of nodes are used. The setup is presented in Figure 7. Each physical computer is running one node at a time which can be the Master, a Hub or a Worker Node.

1. **Master Node** - the main program, which communicates with the *Game Manager*. It passes *START*, *PLAY*, *STOP* and *ABORT* messages without any processing further to other types of nodes. The MCTS/UCT algorithm is used here to construct the game tree. Within a scope of the machine, the algorithm uses Tree Parallelization. Before a *PLAY-clock* elapses, the statistics are gathered and aggregated as in Root Parallelization method.
2. **Hub Node** - an intermediate layer passing messages back and forth between the Master Node and Worker Nodes. Hub Nodes are used only in situations, where there are so many computers available that maintaining connections between the Master Node and each of them would be infeasible. Each Hub Node gathers statistics from a certain group of Worker Nodes and sends them, already aggregated, to the Master Node. A high number of connections to the main machine active at same time (just before a move is made) renders it vulnerable to time-outs, which in turn can lead to game losses. This is important in the GGP Competition scenario. However, with the intermediate layer the responsibility of communicating with Worker Nodes is shared among Hub Nodes and the main machine is not overloaded.
3. **Worker Node** - a procedure of work is similar to the Master Node's one. The difference is only that Workers do not pass messages any further and respond with the statistics of level one nodes (just below the root) instead of the action to play.

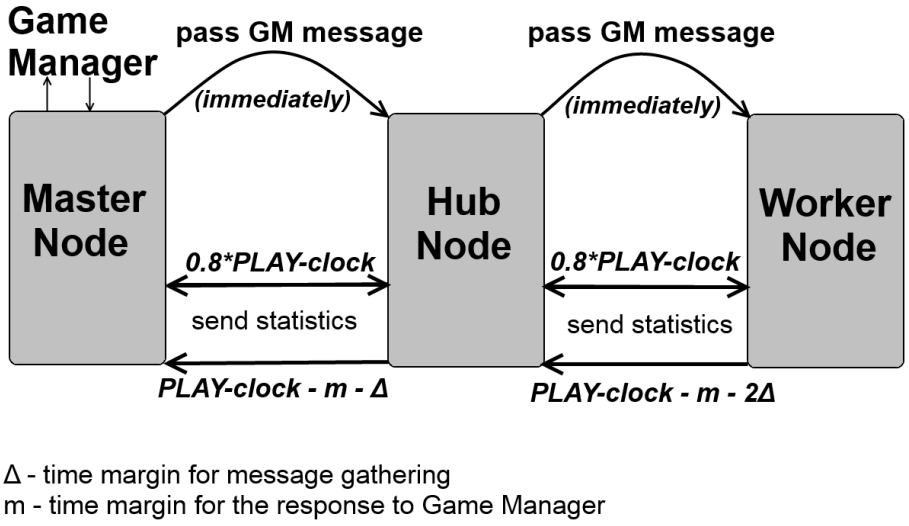

Fig. 7. A data flow in the proposed Hybrid Parallelization algorithm.

4.1 Aggregation of Statistics

Worker Nodes send statistics computed for all currently available actions of all players by local MCTS/UCT algorithms. Because the current state in the game is stored in a tree root, the nodes which contain these statistics are the root's children. The total scores computed for all players and the number of visits are sent twice per each *PLAY-clock* interval. The first synchronization is when 80% of the *PLAY-clock* elapses whereas the second one occurs right before it elapses. The Master Node sums the obtained data for every node which is present in its own game tree. This operation affects the average score which is now updated. Since the nodes are close to the root, the only situation in which they are not present is when a game has an enormously huge branching factor or the *PLAY-clock* is extremely small. Please recall that the UCT algorithm will prioritize actions which were never visited before, so K children of the root are sure to be visited in the first K simulations.

After the first aggregation, the Master Node propagates the aggregated statistics back to Hub Nodes which are further passed to Worker Nodes. Statistics on each machine become synchronized in the first levels of the game trees. From now on, until the *PLAY-clock* expires, the remote nodes are allowed to sample all actions, i.e., the search narrowing algorithm is switched off. In the remaining portion of the *PLAY-clock*, all game trees have the chance to expand in direction of the highest evaluated actions. If the narrowing mechanism was not switched off at any point, the machines which did not have access to the eventually chosen move, would have to build the game tree starting from the root only after every PLAY message. If only there are no more actions in a current state than available machines, Formula 3 guarantees that each action (the played action, in particular) is available to exactly two machines, therefore, all but two would have a degenerated game tree at each step.

4.2 Disadvantages of using intermediate layers

A huge advantage of the Hybrid Parallelization method is that it can utilize a lot of threads and computers relatively easily. As depicted in Figure 5, it has a hierarchical structure, where the Root component is higher in the hierarchy. In comparison to Tree Parallelization method, where a communication cost quickly becomes an issue, it can harvest significantly more power due to less frequent need of sending data. In TP, each request for a simulation generates a message, which can count to thousands per step between a single pair of machines. In HP only two messages are exchanged per step between a pair of machines. The utilization of computers can be even increased by adding more intermediate, i.e., Hub Node, layers. There is no difference whether a Hub Node passes messages to a Worker Node or another Hub Node. A downside of adding intermediate layers is that each such a layer has to include a small time margin - Δ - for communication. To avoid time-outs, the Master Nodes responds with the chosen action with a time margin - m. An i-th layer Hub Node sends the gathered statistics to the Master Node with a margin equal to $m + i * \Delta$. A Worker Node

sends the final statistics with a margin of $m + (K + 1) * \Delta$, where K denotes the number of intermediate layers.

4.3 Advantages over Root Parallelization

In comparison with Root Parallelization, the hybrid method scales better because the aggregation mechanism treats one computer, instead of one thread as in the original RP, as one parallel unit. For instance, if quad-core computers are considered, HP can employ approximately four times more machines with the same communication overhead. Moreover, game trees constructed by the Hybrid Parallel algorithm are of higher quality than in the Root Parallel one.

A predominance of the hybrid method over RP can be derived theoretically. For a given pair (game, parallelization method) there always exists a saturation point beyond which adding more threads does not increase the performance. Eventually, the synchronization and/or communication cost will be even detrimental to the overall player's strength. Let us denote this boundary number of parallel instances for (game1, RP) by B. For a given game, the Root component of Hybrid Parallelization will stop scaling around approximately the same number of units. However, the communication costs imposed by this component are completely unaffected by the synchronization costs from the Tree components running on each remote unit. In the case of a classic RP, the remote units are simply threads. In the case of HP, the remote units are k-core machines parallelized via TP. This concludes, that Hybrid Parallelization scales asymptotically better up to k times.

Root Parallelization has a chance to be more effective only with a small numbers of computers, when it is not yet saturated. In general, it is more valuable to compare HP and TP methods than HP and RP methods. The former are simply more different whereas the latter use the same parallelization idea when analyzed top-down (Figures 3 and 5).

5 Results

We performed two experiments in the following hardware setup: 16 identical machines equipped with *Intel(R) Core i7-2600* processors (4 physical, 8 logical cores), 8GB of RAM and *Windows 7* 64-bit operating system. In both experiments, two players parallelized on 32 threads (8 machines x 4 cores), play a series of 100 matches against each other. The number of matches is limited to 100 due to significant amount of time required to complete the experiment.

In GDL, the obtained scores for players are defined by integer numbers from the $[0, 100]$ interval. These scores, called goals, are valid only in terminal states. In order to make comparison of results from various games easier, we define the concepts of a win, draw and loss. One player wins the match if it achieves a higher score than the opponent it is tested against. In such a case, the latter loses the game. A draw occurs if both tested agents achieve equal scores. In both experiments, the total score for each player is calculated as shown in Eq. 4. The

number of their wins is summed with a half of the number of draws. This method of calculating scores has been widely used in GGP, e.g., in [3], where multiple games are involved in an experiment. After 50 repetitions, the roles in the game are swapped to avoid any bias related to a starting position.

$$Score = (|WINS| + |DRAWS| * 0.5) \tag{4}$$

The first experiment was aimed at testing Limited Root-Tree Parallelization (**Limited-RT**) against the regular Root-Tree Parallelization (**RT**). In such a case, we can measure the impact of the proposed actions limiting mechanism and verify its usefulness.

Nine significantly different games, including chess-like games, connection games, market-inspired games, of various complexity (in terms of the numbers of possible unique states and actions) were chosen. For these selected games, Table 1 presents two game properties which are very important from the parallelization point of view: the average time required to perform one Monte Carlo simulation and the average number of possible moves (branching factor).

Table 1. The average time required to complete one game simulation in our framework (on a single thread) and the approximate average branching factor of each game. The branching factors marked with a * were computed empirically based on massive simulations in large UCT trees.

Game	1-sim time	Branching Factor
Breakthrough	4.6 [ms]	20*
Checkers	20.7 [ms]	2.8
Connect4	0.9 [ms]	4
Farmers	1.7 [ms]	1000
Farming Quandries	1.5 [ms]	8*
Hex 9x9	0.8 [ms]	79*
Othello	298.5 [ms]	10
Pentago	13.9 [ms]	97.3
Pilgrimage	20.3 [ms]	9.2*

Table 2 presents the experimental results. Using 75% confidence intervals, the total result (measured in wins-draws-losses) is 4-1-4 whereas within 95% confidence intervals, the outcome is 3-3-3. The average score is slightly higher for the Limited Root-Tree variant, but it is not significant by margins with at least 75% confidence. We can conclude that both methods are similarly strong but suitable for different games. Limiting actions on a machine seems to work well for Farming Quandries, because in this game usually the last actions computed by the GDL interpreter are the best, so the last two machines can take full advantage of that. Good performance of this method in Farmers and Pentago is probably caused by high average branching factors in that games, around 1000 and 100 respectively. Both games feature some universally good moves such

as the buy/sell actions near the start/end, respectively, in Farmers or placing stones in the middle of four sub-boards in Pentago. Hex, having a relatively high branching factor too, features no universally good moves in the above-mentioned sense. Connect Four has the second lowest branching factor, and there are no move patterns like in Farming Quandries, what is the reason for the plain Hybrid Method being stronger there. The same (low branching factor and no move ordering patterns), is true for Othello. In a game-dependent scenario, both parallelization methods should be initially verified (before choosing the right one), because neither is universally stronger.

Table 2. Evaluation of Root-Tree Parallelization (**Limited-RT**) against the plain Root-Tree Parallelization (**RT**). The results above 50, which are shown in bold, are in favor of the first player appearing in the table, which is Limited-RT in this case. Values in column *Clock* represent the time alloted for *START-clock* (the left value) and *PLAY-clock* (the right value). There are both 75% and 95% confidence intervals put in square brackets. Confident wins by either player are marked with ticks next to confidence intervals.

Game	**Clock [s]**		**Limited-RT vs. RT**	**75% conf. interval**	**95% conf. interval**
Breakthrough	45	8	42.00	[4.16] ✓	[9.67]
Checkers	60	10	**51.00**	[3.99]	[9.30]
Connect4	40	5	37.50	[3.64] ✓	[8.47] ✓
Farmers	45	5	**57.00**	[3.86] ✓	[8.98]
Farming Quandries	60	8	**77.00**	[3.49] ✓	[8.13]✓
Hex 9x9	60	10	40.00	[4.12] ✓	[9.60] ✓
Othello	90	15	39.00	[4.06] ✓	[9.46] ✓
Pentago	45	8	**63.00**	[3.93] ✓	[9.15]✓
Pilgrimage	90	10	**61.00**	[3.79]✓	[8.83]✓
Average			**51.94**		

In the second experiment (see Table 3), the proposed parallelization method (**Limited-RT**) was evaluated against a player using Tree Parallelization (**Tree**). The total result (wins-draws-losses) is 6-1-2 in favour of the former method when the 75% confidence intervals are applied and 5-2-2 with the 95% confidence. Moreover, the average score of **Limited-RT** - 58.60 - is significantly higher than that of **Tree**. The **Tree** approach appears to be generally better in slowly simulated games (with the exception of checkers). Please notice, that **RT** as well as **Limited-RT**, contain **Tree** as a subsystem limited to 4 CPU cores (in our case). The biggest gain of parallelization measured after adding an additional thread is noticed with the lowest number of threads. The more cores (threads), the worse scalability of **Tree**, so the hybrid methods are much more promising in maintaining scalability.

Table 3. Evaluation of Root-Tree Parallelization (**Limited-RT**) against Tree Parallelization (**Tree**). See the description of Table 2 for the interpretation of results.

Game	**Clock [s]**		**Limited-RT vs. Tree**	**75% conf. interval**	**95% conf. interval**
Breakthrough	45	8	**60.00**	[4.12] ✓	[9.60] ✓
Checkers	60	10	**62.50**	[3.88] ✓	[9.02] ✓
Connect4	40	5	**57.00**	[3.47] ✓	[8.08]
Farmers	45	5	**83.50**	[2.67] ✓	[6.21] ✓
Farming Quandries	60	8	**86.00**	[2.80]✓	[6.51]✓
Hex 9x9	60	10	**64.00**	[4.04]✓	[9.41]✓
Othello	90	15	24.00	[3.60] ✓	[8.37] ✓
Pentago	45	8	**52.00**	[4.21]	[9.79]
Pilgrimage	90	10	38.00	[3.91] ✓	[9.10] ✓
Average			**58.60**		

6 Conclusions

In this paper, a new method of parallelization of the MCTS algorithm was presented. The method was applied to a GGP program called MINI-Player. Its underpinning ideas are a combination of Root and Tree Parallelization and narrowing the search to only certain subsets of actions on remote units. We proposed a mechanism of data aggregation coming from the remote machines. The combination of the two mentioned parallelization methods appears to be more suitable for GGP scenario than Tree Parallelization alone (shown empirically) and Root Parallelization alone (discussed theoretically). The mechanism of narrowing actions is beneficial in certain class of games, especially with high branching factor and repetitive move patterns (e.g., when the last moves with respect to specific ordering are stronger in average).

A dynamic detection of the most suitable parallelization method based on game rules analysis in the *START-clock* time is one of the future goals. We also plan to perform further tests with a higher number of games using the proposed methods. In general, parallelization of the MCTS algorithm is a robust and promising way to increase its performance. The methods discussed in this paper in the scope of GGP should be easily adapted to any game or even beyond the game domain where the MCTS algorithm is used.

Acknowledgment

M. Świechowski was supported by the Foundation for Polish Science under International Projects in Intelligent Computing (MPD) and The European Union within the Innovative Economy Operational Programme and European Regional Development Fund.

This research was financed by the National Science Centre in Poland, based on the decision DEC-2012/07/B/ST6/01527.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Arneson, B., Hayward, R.B., Henderson, P.: Monte carlo tree search in hex. Computational Intelligence and AI in Games, IEEE Transactions on **2**(4) (2010) 251–258
3. Björnsson, Y., Finnsson, H.: CadiaPlayer: A Simulation-Based General Game Player. Computational Intelligence and AI in Games, IEEE Transactions on **1**(1) (March 2009) 4–15
4. Bratko, I.: Prolog Programming for Artificial Intelligence. International Computer Science Series. Addison Wesley (2001)
5. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A Survey of Monte Carlo Tree Search Methods. Computational Intelligence and AI in Games, IEEE Transactions on **4**(1) (March 2012) 1–43
6. Cazenave, T., Jouandeau, N.: On the Parallelization of UCT. Proceedings of CGW07 (2007) 93–101
7. Cazenave, T., Jouandeau, N.: A Parallel Monte-Carlo Tree Search Algorithm. In: Computers and Games. Springer (2008) 72–80
8. Chaslot, G., Winands, M.H., Szita, I., van den Herik, H.J.: Cross-Entropy for Monte-Carlo Tree Search. ICGA Journal **31**(3) (2008) 145–156
9. Coulom, R.: Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In: Computers and games. Springer (2007) 72–83
10. Enzenberger, M., Müller, M.: A lock-free multithreaded monte-carlo tree search algorithm. In: Advances in Computer Games. Springer (2010) 14–20
11. Gelly, S., Kocsis, L., Schoenauer, M., Sebag, M., Silver, D., Szepesvári, C., Teytaud, O.: The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions. Communications ACM **55**(3) (March 2012) 106–113
12. Gelly, S., Silver, D.: Achieving Master Level Play in 9 x 9 Computer Go. In: AAAI. Volume 8. (2008) 1537–1540
13. Genesereth, M.R., Love, N., Pell, B.: General Game Playing: Overview of the AAAI Competition. AI Magazine **26**(2) (2005) 62–72
14. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Proceedings of the 17th European conference on Machine Learning. ECML'06, Berlin, Heidelberg, Springer-Verlag (2006) 282–293
15. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General Game Playing: Game Description Language specification. Available at: `htp://games.stanford.edu/readings/gdl_spec.pdf` (2008)
16. Mańdziuk, J., Świechowski, M.: Generic Heuristic Approach to General Game Playing. In Bieliková, M., Friedrich, G., Gottlob, G., Katzenbeisser, S., Turán, G., eds.: SOFSEM. Volume 7147 of Lecture Notes in Computer Science., Springer (2012) 649–660
17. Méhat, J., Cazenave, T.: A Parallel General Game Player. Künstliche Intelligenz **25**(1) (2011) 43–47

18. Méhat, J., Cazenave, T.: Tree Parallelization of Ary on a Cluster. In: Proceedings of the IJCAI-11 Workshop on General Game Playing (GIGA'11). (2011) 39–43
19. Plaat, A., Schaeffer, J., Pijls, W., de Bruin, A.: Best-First Fixed-Depth Minimax Algorithms. Artificial Intelligence **87**(1–2) (1996) 255–293
20. Świechowski, M.: Adaptive Simulation-Based Meta-Heuristic Methods in Synchronous Multiplayer Games. PhD thesis, Systems Research Institute, Polish Academy of Sciences, Warsaw, Poland (2015) in review.
21. Świechowski, M., Mańdziuk, J.: Fast Interpreter for Logical Reasoning in General Game Playing. Journal of Logic and Computation (2014) DOI: 10.1093/logcom/exu058.
22. Świechowski, M., Mańdziuk, J.: Self-Adaptation of Playing Strategies in General Game Playing. IEEE Transactions on Computational Intelligence and AI in Games **6**(4) (Dec 2014) 367–381
23. Świechowski, M., Mańdziuk, J.: Specialized vs. Multi-game Approaches to AI in games. In Angelov, P., Atanassov, K., Doukovska, L., Hadjiski, M., Jotsov, V., Kacprzyk, J., Kasabov, N., Sotirov, S., Szmidt, E., Zadrożny, S., eds.: Intelligent Systems'2014. Volume 322 of Advances in Intelligent Systems and Computing. Springer International Publishing (2015) 243–254
24. Świechowski, M., Mańdziuk, J., Ong, Y.S.: Specialization of a UCT-based General Game Playing Program to Single-Player Games. IEEE Transactions on Computational Intelligence and AI in Games (2015) *(accepted for publication)*
25. Świechowski, M., Park, H., Mańdziuk, J., Kim, K.: Recent Advances in General Game Playing. The Scientific World Journal **2015** (2015)
Available at: `http://dx.doi.org/10.1155/2015/986262`.
26. Syed, O., Syed, A.: Arimaa - A New Game Designed to be Difficult for Computers. ICGA **26** (2003) 138–139
27. Teytaud, F., Teytaud, O.: Creating an Upper-Confidence-Tree Program for Havannah. In: Advances in Computer Games. Springer (2010) 65–74