Distributed Monte-Carlo Tree Search: A Novel Technique and its Application to Computer Go

Lars Schaefers, and Marco Platzner, *Senior Member, IEEE*

***Abstract*—Monte-Carlo Tree Search (MCTS) has brought about great success regarding the evaluation of stochastic and deterministic games in recent years. We present and empirically analyze a data-driven parallelization approach for MCTS targeting large HPC clusters with Infiniband interconnect. Our implementation is based on OpenMPI and makes extensive use of its RDMA based asynchronous tiny message communication capabilities for effectively overlapping communication and computation. We integrate our parallel MCTS approach termed UCT-Treesplit in our state-of-the-art Go engine Gomorra and measure its strengths and limitations in a real-world setting. Our extensive experiments show that we can scale up to 128 compute nodes and 2048 cores in self-play experiments and, furthermore, give promising directions for additional improvement. The generality of our parallelization approach advocates its use to significantly improve the search quality of a huge number of current MCTS applications.**

***Index Terms*—UCT, HPC, Monte-Carlo Tree Search, Go, distributed memory.**

I. Introduction

A large number of typical HPC applications fall into fields such as financial simulations, data mining or numerically solving differential equations. These applications are especially amenable to parallelization due to their regular memory access patterns, exploitable data locality and a high degree of data independent operations that can be performed concurrently. In contrast, there also exist classes of algorithms with random or highly irregular memory access patterns that pose a challenge for parallelization. One such class is graph and thereby also tree search algorithms [1]. Tree traversal generally requires inhomogeneous and sparse access to memory that is often difficult to predict, especially when the traversal policy depends on volatile data stored with tree nodes or edges.

In this paper, we present results on our parallelization of so called Monte-Carlo Tree Search (MCTS) algorithms, that combine tree search with the Monte-Carlo approach. MCTS is a simulation based search method that brought about great success in the past few years regarding the evaluation of stochastic and deterministic two-player games. Especially in the field of Computer Go, a popular Asian two-player board game, MCTS highly dominates over traditional methods such as $\alpha\beta$ game tree search [2]. In the wake of the enormous success of MCTS in Computer Go, numerous researchers adopted to its use in subsequent years. In a recent survey of MCTS methods, Browne et al. [3] listed almost 250 MCTS related publications originating only from the last seven years, which demonstrates the popularity and importance of MCTS. MCTS is currently emerging as a powerful tree search algorithm, yielding promising results in many search domains requiring only little or no domain knowledge at all. MCTS also performs remarkably well for games other than Go, such as the connection games Hex [4] and Havannah [5], the combinatorial games Othello [6] and Amazons [7] as well as General Game Playing and real-time games. Apart from games, MCTS finds applications in combinatorial optimization [8], constraint satisfaction [9], scheduling problems [10], sample-based planning [11] and procedural content generation [12].

L. Schaefers and M. Platzner are with the Department of Computer Science, University of Paderborn, Paderborn, NRW, 33098 Germany e-mail: slars@upb.de, platzner@upb.de.

MCTS learns a value function for game states by consecutive simulation of complete games of self-play using semi-randomized policies to select moves for either player. MCTS may be classified as a sequential best-first search algorithm [13], where *sequential* indicates that simulations are not independent of each other, as is often the case with Monte-Carlo algorithms. Instead, statistics about past simulation results are used to guide future simulations along the search space's most promising paths in a best-first manner. Looking for parallelization, this dependency could partly be ignored with the aim of increasing the number of simulations for the price of eventually exploring less important parts of the search space. Parallelization of MCTS for distributed memory environments is a highly challenging task since typically we want to adhere to simulation dependencies and, additionally, we need to store and share simulation statistics among all computational entities.

Parallelization of traditional $\alpha\beta$ search is a pretty well solved problem, e.g., see [14][15]. While for $\alpha\beta$ search it is sufficient to map the actual move stack to memory, MCTS requires us to keep a steadily growing search tree representation in memory. Sharing a search tree as the central data structure in a distributed memory environment is rather involved and only a few approaches have been investigated so far [16][17].

In this paper, we present major algorithmic improvements and extend experimental data regarding our parallelization of MCTS for distributed high-performance computing (HPC) systems that we initially presented in [18]. We present an efficient parallel transposition table highly optimized for MCTS applications in time critical use cases. Even further, we propose the use of dedicated compute nodes (CNs) to support necessary broadcast operations, that allow for greatly reducing network traffic by multi-stage message merging.

We evaluate the performance of our parallelization technique on a real-world application, our high-end Go engine Gomorra [18]. All experiments are run on a homogeneous compute cluster with 4xQDR Infiniband interconnect. Our implementation is based on the OpenMPI library and exploits capabilities for low latency Remote-Direct-Memory-Access (RDMA) communication of tiny messages.

Our Go engine Gomorra has proven its strength at the Computer Olympiad 2010 in Kanazawa, Japan, the Computer Olympiad 2011 in Tilburg and several other international Computer Go tournaments. Gomorra is regularly placed among the strongest 6 programs and only recently won a silver medal at the Computer Olympiad 2013 in Yokohama, Japan.

In summary, in this paper we make the following contributions:

- We present an efficient *parallel transposition table*, optimized for time critical parallel MCTS applications that share a single game tree representation in distributed memory.
- We realize the setup and integration of dedicated compute nodes that support necessary broadcast operations. This allows for *greatly reduced network traffic during broadcasts by multi-stage message merging* and more efficient use of RDMA based tiny message communication.
- We integrate the parallel transposition table and the dedicated compute nodes into a strong state-of-the-art Go playing program to be able to conduct *experiments with a real world application*, and show scalability to more than 2000 compute cores.
- We demonstrate that the resulting MCTS parallelization makes efficient use of most compute and memory resources available in the cluster. This is in strong contrast to most formerly investigated parallelization methods that either extensively duplicate data [19][16] or resort to using only a fraction of a cluster's overall memory capacity.

The remainder of the paper is structured as follows: Section II introduces the basic MCTS algorithm and reviews related work in parallelizing MCTS. Our novel parallelization approach for MCTS is presented in Section III. Section IV evaluates our algorithm and details the experimental setup and results achieved. We discuss our results and their relation to the work of others in Section V. Section VI concludes the paper and gives an outlook to future work.

II. Background and Related Work

In this section we provide some background of MCTS techniques and review related work. First, we give a brief introduction to games and basic MCTS and then focus on efforts to parallelize MCTS algorithms.

A. Games

Before presenting the general MCTS framework, we introduce a formalism for games that represent the search domain we use throughout this paper. We write a game as a tuple $G := (S, A, \Gamma, \delta, r)$, with S being the set of all possible states (i.e. game positions) and A being the set of actions (i.e. moves) that lead from one state to the next. A function $\Gamma : S \rightarrow \mathcal{P}(A)$ determines the subset of available actions at each state. For each such available action, the transition function $\delta : S \times A \rightarrow \{S, \emptyset\}$ specifies the follow-up state. Because $\delta(s, a)$ can only return a valid state for actions a that are available at state s, its co-domain contains the empty set, i.e. $\delta(s, a) = \emptyset$ iff $a \notin \Gamma(s)$. A state in which no action is applicable is called *terminal*. Hence, the set of terminal states is denoted by $S_t := \{s \in S | \Gamma(s) = \emptyset\}$. Finally, a reward function $r : S_t \rightarrow (0, 1)$ assigns a reward value to each terminal state. In case of Go, for example, we assigned 1 to each terminal position that yields a win for the black player and 0 to each position that marks a win for white.

B. Basic MCTS

We present the most basic MCTS algorithm used for two-player zero-sum games with complete information. While we concentrate on two-player games and especially the game of Go, for reasons of comparability with the work of others, we want to note that our approach is applicable to the wider class of Markov Decision Processes (MDP), e.g., see [8]. The so-called UCT algorithm (short for *Upper Confidence Bounds applied to trees* [8]) is a modern variant of MCTS and yields the experimentally best results for most of the current Go programs. Algorithm UCT shows a pseudo code representation of UCT.

Algorithm UCT: Basic UCT-Algorithm for two-player zero-sum games

```
Data: A game $G := (S, A, \Gamma, \delta, r)$ as described in the text. A
      set $T \subseteq S$ contains all states that have a memory
      representation. Counters $N_{s,a}$ and $W_{s,a}$ are kept in
      memory for all states $s \in T$ and their corresponding
      actions $a \in \Gamma(s)$. We further set $N_s := \sum_{a \in \Gamma(s)} N_{s,a}$.
input : A state $s_0 \in S$ and a time limit
output: An action $a \in \Gamma(s_0)$
$T \leftarrow s_0$; $d \leftarrow 0$;
while Time available do
    if $s_d \in T$ then
        // in-tree policy:
        $a_d \leftarrow$
        $\text{argmax}_{a \in \Gamma(s)}$ actionValue($W_{s_d,a}, N_{s_d,a}, N_s$);
        $N_{s_d,a_d} \leftarrow N_{s_d,a_d} + 1$;
        $s_{d+1} \leftarrow \delta(s_d, a_d)$;
        $d \leftarrow d + 1$;
    else
        $T \leftarrow T \cup s$; // Expand memory tree
        reward $\leftarrow$ playout ($s_d$);
        update (reward); // Update all $W_{s_i,a_i}$ for
        $0 \leq i \leq d$
        $d \leftarrow 0$; // Start a new simulation
    end
end
return $\text{argmax}_{a \in \Gamma(s_0)}, N_{s_0,a}$;

Function actionValue $(w, v, V)$
    return $\frac{w}{v} + C\sqrt{\frac{\log(V)}{v}}$; // With $C$ being a
    constant
```

UCT takes a state and a time limit as inputs and returns an action. As long as the time limit is not exceeded, UCT computes search tree simulations and builds up statistics about their outcomes for visited states. Being a so-called anytime algorithm, UCT can be interrupted any time and returns the

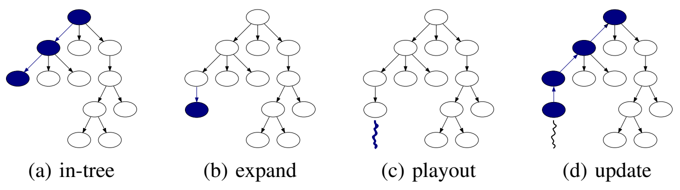

Fig. 1: Building blocks of MCTS

best action found so far. As memory is generally limited, practical implementations build up simulation statistics for near-root tree nodes only. An effective way proposed in [20] is the generation of a memory tree T by starting with the root node and adding the first node not already covered by T during each simulation. This method leads to an efficient and predictable memory usage, as the memory tree likely grows in the most interesting branches and a maximum of one tree node is added with each additional simulation. Accordingly, simulation guidance based on node statistics is only possible for nodes covered by T. Once a simulation leaves T, a semi-randomized heuristic policy is used for action selection until a terminal game position, i.e. a leaf of the real game tree, is reached. We call this randomized policy the *playout policy* and the history-dependent one used for nodes covered by T *in-tree policy*.

Practical MCTS approaches may be divided into the four steps in-tree, expand, playout, and update, that form one simulation and that are repeated in a loop as illustrated in Figure 1. In Section III we will use these steps to form work packages that can be distributed across a cluster.

UCT handles the task of selecting an action as a multi-armed-bandit problem (MAB) and is designed following prior investigations on MAB in [21]. As shown in Algorithm UCT, all data needed to select an action in the in-tree phase are the simulation statistics of all possible actions available at that state. Thus, for practical implementations it makes sense to store the statistics of all those actions together to optimize memory access patterns. This becomes even more important when considering the distribution of the search tree in a cluster, where all data necessary for an action selection step should be stored on a single CN to minimize communication overhead.

The computation of playouts is completely independent of former simulation results and therefore a perfect place to look for work-packages that may be distributed across a cluster. For many search domains, the playout step dominates the simulation runtime, especially in the early stages of a game where the depth of the search tree T remains much smaller than the real game tree depth.

C. Transposition Table

In many games it is possible to reach the same game state through different move sequences. Hence, eventually the resulting search space becomes a directed acyclic graph (DAG) rather than a tree. To be able to use MCTS as explained above and to prevent the creation of multiple tree nodes for identical game states, we need to detect such cases. An efficient way is the use of a hash-table to store the memory representation of the search tree T. In game tree search, this hash-table is commonly called the *transposition table*. In the MCTS context, in 2009, Enzenberger and Müller [22] described a lock-free implementation of a transposition table based on the use of atomic instructions, that allows for very efficient concurrent access in shared memory environments. Already in 2008, Coulom posted his results and implementation details on a lock-free transposition table to the Computer Go mailing list[1]. Moreover, chess programmers have been using lock-free implementations of transposition tables for many years, cf. [23].

D. Parallelization

The most common parallelization methods presented so far are termed Tree-Parallelization, Leaf-Parallelization and Slow-Tree-Parallelization [24][25][19][16]. Tree-Parallelization is most common on shared memory systems, as one search tree representation is shared among several compute cores. Each core performs one simulation at a time and updates the transposition table, which can be implemented in a lock-free manner as described in Section II-C. Leaf-Parallelization and Slow-Tree-Parallelization are both suited for distributed memory machines. Leaf-Parallelization handles the in-tree part on one single CN, and computes multiple playouts on remote CNs once a leaf of the search tree representation T is reached. Slow-Tree-Parallelization performs almost independent searches on all CNs, each of which maintains its own search tree representation. Only statistics of near-root tree nodes become synchronized between the CNs in regular intervals.

Among these methods, Slow-Tree-Parallelization currently excels for distributed memory systems [26]. One drawback of this method is that rather little effort is made to exploit the increased amount of memory available within a cluster.

In 2011, simultaneously and independent to our paper [18], Yoshizoe et al. [17] published an MCTS parallelization that manages the sharing of a single search tree representation on a distributed memory system. Both, Yoshizoe et al. and we proposed a solution based on transposition-table driven scheduling (TDS) [27]. In contrast to us, Yoshizoe et al. developed a depth-first version of the UCT policy (df-UCT) that reduces the update frequency of statistics associated to often visited tree nodes. This is achieved by delaying and merging necessary backpropagations of simulation results. They restrict their scalability experiments to simulation rates achievable with artificial game trees and mention experiments regarding search quality improvements with real games like Go as an important direction for future work. Our own proposed method is detailed in the following section.

III. The UCT-Treesplit Algorithm

We concentrate on homogeneous HPC systems with a fast, low-latency Infiniband interconnect consisting of N compute

[1]Coulom's mail to the Computer Go mailing list about a lock-free transposition table in the MCTS context was sent in 2008 on the 21st of March and is available online at:
`http://www.mail-archive.com/computer-go@computer-go.org/msg07611.html`

nodes (CNs), each having P CPUs that in turn each have $C > 1$ compute cores that share a CN's entire memory. We assume this is a model that fits modern HPC cluster systems. We use MPI for message passing and assign one MPI rank, thus one instance of our program, to each CPU. Hence, we will run $M = N \cdot P$ program instances in parallel. Assigning one MPI rank to each CPU is important to allow for faster and more homogeneous memory access times as the use of e.g. QPI links can be reduced significantly. We devote one core (IO) of each CPU to a thread handling message passing and work package distribution, while the remaining $C - 1$ cores (workers) of each CPU are bound to worker threads. Figure 2 illustrates the setup of one MPI rank on a CPU.

The IO and worker cores communicate using ring-buffers (Transfer Buffer In/Out) that reside in shared memory. The *main loop* running on the IO core reads available messages containing work packages from the network link and stores a reference to the corresponding memory location in a buffer. Afterwards, a work package scheduler distributes the received packages among the workers' ring-buffers, balancing the work load. Workers poll the transfer buffers and start computation once they find a package and, if required, send response messages back to the IO core using the corresponding buffers. In turn, the IO core frequently collects messages from the workers' ring-buffers and forwards them to the network link as appropriate. For practical implementations, care must be taken to pass around references to memory locations rather than copying entire messages whenever possible. We recommend the allocation of a large memory buffer using the MPI function `MPI_Alloc_mem` at program start and handling the memory management within the user application, as frequent allocation and freeing of dynamic memory can easily lead to highly inhomogeneous processing times.

Although ring buffers between dedicated communication partners can be implemented lock free and thus allow for efficient shared memory communication, in our experiments it turned out that allowing workers to look for work packages not only in their own but also in other workers queues, improves the system's overall work load, even though the ring-buffers require explicit locking in this case.

During computation, workers need access to MC-simulation statistics of the tree nodes. As mentioned in Section II-C, it is possible for some search domains that equal states are reached through different paths motivating the use of transposition tables. As transposition table we use a hash table H that is capable of storing up to $|\mathrm{H}|$ nodes of the search tree representation. Section III-D gives a more detailed description of our implementation. We assume the existence of a hash-function $h : S \to \mathbb{N}$ assigning a unique and equally distributed hash value to each single state in the search space. One way for computing an index $I_{\mathrm{H}}(s)$ into H for a search tree node $s \in S$ is given by:

$$I_{\mathrm{H}}(s) = h(s) \bmod |\mathrm{H}|$$

In a manner similar to [28] and [29], we distribute H among M MPI ranks on a cluster by storing for each rank $i \in \{0, \dots, M-1\}$ a partial transposition table H_i with $|\mathrm{H}|/M$ entries. An index to the distributed table is a tuple of a rank i and a local index I_{H_i} to H_i. Both are computed as follows:

$$i(s) = (h(s)/|\mathrm{H_i}|) \bmod M \tag{1}$$

$$I_{\mathrm{H_i}}(s) = h(s) \bmod |\mathrm{H_i}| \tag{2}$$

The key technique of our approach is based on the transposition table driven scheduling (TDS) that was used with, e.g., controlled conspiracy number search [30] and $\alpha\beta$-search, or more precisely its variant MTD(f) [31] before. We spread a single search tree representation among the local memories of all CNs, moving computational tasks to the CNs that own the required data. Therefore, we break simulations into work packages that can be computed on different cores. Moreover, cores computing work packages of one simulation do not need to be on the same CN. Message passing is used to guide simulations over CN boundaries. We overlap necessary communication by computing more simultaneous simulations than there are actual worker cores. This is a standard technique that is generally known as *latency hiding*.

Figure 3 illustrates our distributed simulation process as a finite state machine (FSM). The states are the work packages that make up the computational load and that were derived from the building blocks of MCTS simulations as depicted in Figure 1. Dotted arrows represent state transitions that always happen at a single MPI rank. Solid arrows indicate a possible movement to other MPI ranks and are annotated with the corresponding messages. During the in-tree part of a simulation, several action selection steps take place. Each of those steps can be computed without the need to communicate with other CNs by storing the statistics about all actions available in one state together in memory. Between two consecutive action selection steps, a simulation may move to another CN through a message, denoted as MOVE message in Figure 3. Another message, denoted as UPDATE message in the figure, is sent to all ranks visited by the simulation as statistics need to be updated on all these ranks.

During search, a number of simulations are computed in parallel. We denote this number with S_{par}. Each of S_{par} simulations running in parallel suffers loss of information represented by the results of the $S_{\mathrm{par}}-1$ other simulations that would be available in a sequential UCT version. Obviously this impairs the search quality [32] and urges us to keep S_{par} as small as possible. Furthermore, we duplicate and occasionally synchronize frequently visited tree nodes on all ranks to reduce the communication overhead, hence speed up the overall average time required to complete a single simulation and finally to prevent high congestion of single CNs that are visited by an above average number of simulations. Therefore, besides the partial transposition table Figure 2 also shows a cache for entries of remote transposition tables that is used exactly for those duplicated entries. In total the algorithm requires us to determine

- An overload factor O to compute the number of simulations that run in parallel on M MPI ranks using $C - 1$ worker cores each: $S_{\mathrm{par}} := (C - 1)MO$.
- A policy for duplication and synchronization of frequently visited tree nodes.

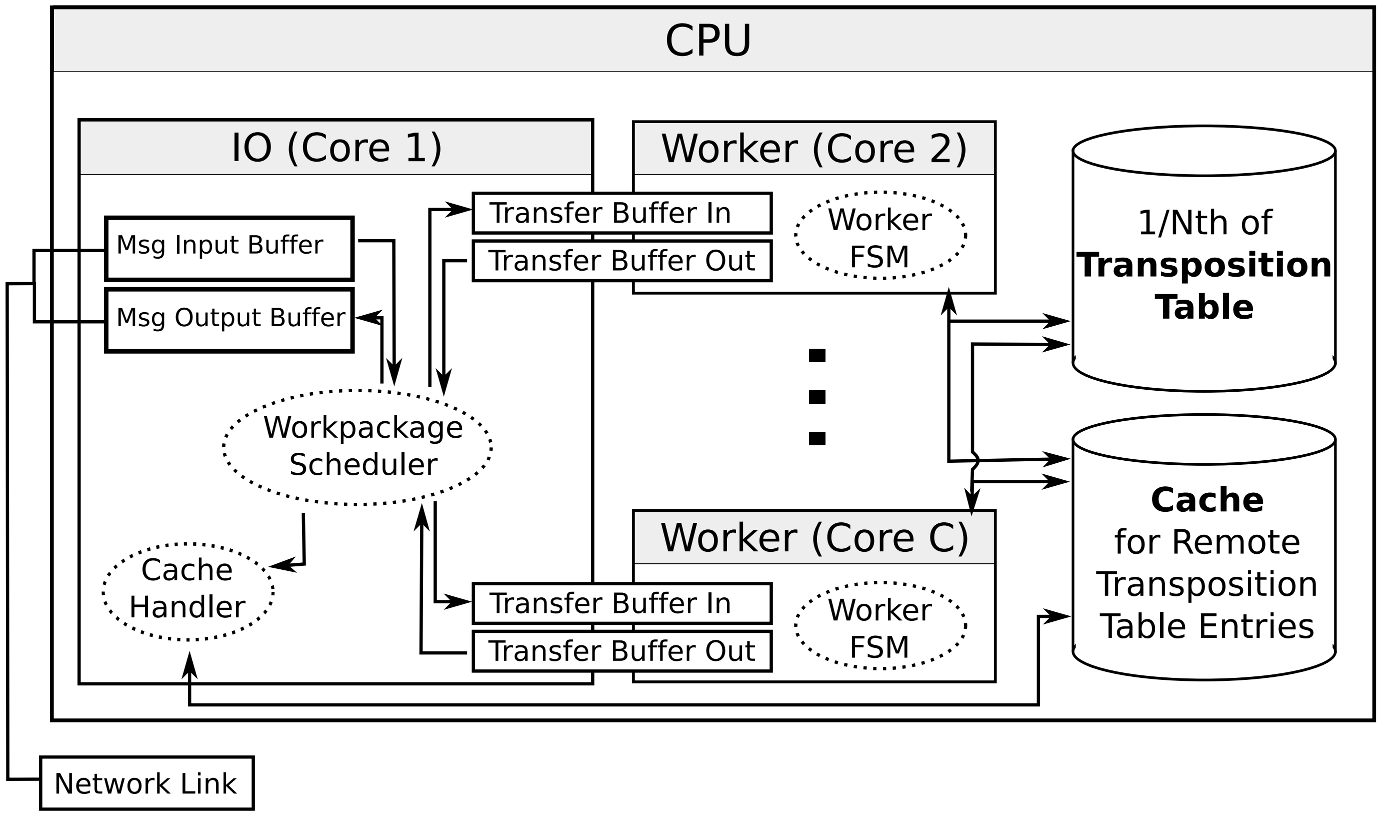

Fig. 2: Setup of an MPI rank on a CPU.

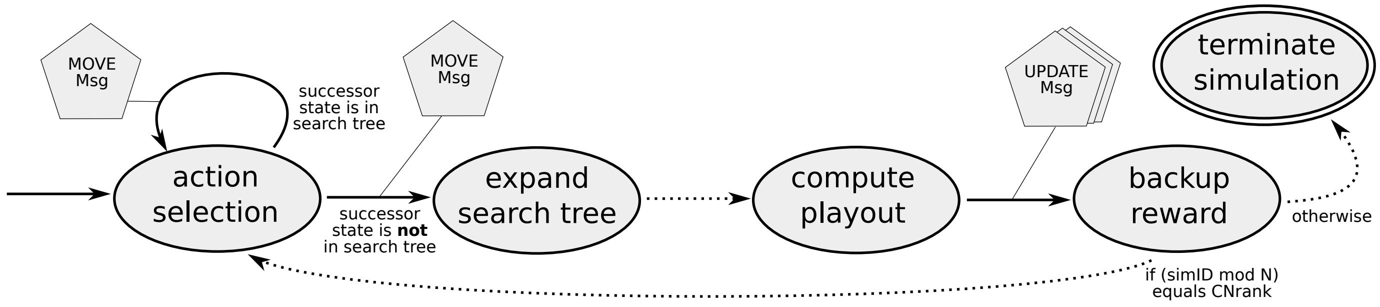

Fig. 3: Finite state machine for distributed simulations (Worker FSM).

While the value of the overload factor O can be determined empirically, we discuss and propose a policy for tree node duplication and synchronization in Section III-C. Note that UCT-Treesplit may be configured to behave comparably to Slow-Tree-Parallelization by sharing near-root nodes immediately. On the other extreme, UCT-Treesplit behaves like Tree-Parallelization if tree nodes are never duplicated.

The search process begins by sharing the root node among all MPI ranks. Then, each rank starts S_{par}/M simulations. The data structure representing a simulation consists in principle only of the state-action stack containing all states visited and actions taken during simulation. Together with each state, we store the rank where the action selection took place. The rank information allows for determining the destination ranks of necessary UPDATE messages. For UPDATE messages we additionally include the playout's reward and in our Go specific implementation also some information describing the actual playout made to support some domain specific state-of-the-art heuristics.

A. Load Balancing

As stated above, we assume the playout work packages to make up by far the largest portion of the overall computational load that comes along with each single simulation. By design of the algorithm, these work packages are distributed randomly across all worker MPI ranks. This, however, does not ensure that work is well balanced at any time but only on average. Even further, the computation of a playout is not strictly bound to a fixed MPI rank as the transposition table does not need to be accessed. We make use of this absence of data dependency by forwarding playout computations to other ranks in case we notice *unusual high workload* on the rank that originally had to compute it. To perceive unusual high workload at a specific rank, we look at the number of MOVE messages that reside in the rank's message buffers. Being aware that in total at most S_{par} simulations are running in parallel, we expect to find an average number of (S_{par}/M) MOVE messages. We define a rank's workload to be unusual high in case we find more than $(S_{\text{par}}/M) \cdot K$ with $K > 1$ MOVE messages in the rank's message buffers. Playout work packages are then forwarded to other, randomly chosen CNs that might of course themselves forward the work again until CNs with appropriate workload were found. Choosing $K > 1$ ensures that such CNs exist. In our experiments we achieved satisfying results with $K = 1.2$.

B. Broadcast Ranks

One obvious bottleneck in this design is the need for regular all-to-all communication to synchronize the statistics for shared near-root nodes. We propose to add additional CNs to help spreading synchronization messages. In the remainder of this paper we denote the MPI ranks that handle the tree search *search ranks* and those helping with shared tree node synchronization *broadcast ranks*. The design choice of using broadcast ranks even allows for merging synchronization messages originating from different search ranks that affect the same tree node and hence, greatly helps to reduce network traffic and takes over additional work load from the search ranks. We empirically determined a good ratio is to be one broadcast rank for every four search ranks. Hence, each search rank has a dedicated broadcast rank to send all synchronization messages to. In turn a search rank will also receive the synchronization messages of other search nodes only via this dedicated broadcast partner. The broadcast ranks themselves synchronize by all-to-all communication. To reduce the network traffic and work overhead coming along with synchronization, messages are artificially delayed by the broadcast ranks to increase the probability that messages containing informations for equal tree nodes can be merged. This delay and merge technique considerably reduces the number of synchronization messages that have to be received and processed by each single search rank. Otherwise, the number of those messages would steadily increase with the number of search ranks and thereby limit the scalability of the overall algorithm.

We determined the broadcast ranks to search ranks ratio (1:4) empirically for 16 search ranks. For other numbers of search ranks, different ratios might yield better performance. Furthermore, our broadcast rank's implementation is single threaded. Hence, further performance improvements might be obtainable by handling work packages like synchronization message duplication and merging on additional compute cores. In summary, our implementation of broadcast ranks is rather a proof of concept than being optimal.

The use of Infiniband allows for so called RDMA (Remote-Direct-Memory-Access) communication, that helps shifting communication related work almost entirely to the corresponding Infiniband hardware. This technique not only provides very low latencies (less than 1 microsecond) but simplifies overlapping of communication with computation remarkably. A practical drawback of RDMA, however, is the need of preallocated receive buffers on each CN for each of the CN's potential communication partners (peers). Even further, these buffers need to be polled in order to recognize the completion of asynchronous message receive operations. Hence, in terms of memory requirements and polling time (and thus CPU time), the use of RDMA in combination with all-to-all communication does not scale to arbitrarily sized clusters. Actually, OpenMPI limits the number of RDMA peers per default to only 16 per MPI rank.

To address this, we introduced a communication pattern for the broadcast ranks that makes each broadcast rank communicate with a limited number of L other broadcast ranks in addition to its associated search ranks. The pattern minimizes the number of hops and equally distributes the communication and work load among the broadcast ranks. Recalling that the number of search ranks is denoted by M, and having $B = M/4$ broadcast ranks, we thus search for positive numbers k, n and $m \in \mathbb{N}$ that fulfill the following two conditions:

$$k^n \cdot m = B \tag{3}$$

$$n + m - 1 \leq L. \tag{4}$$

In case more than one realization of k, n and m can be found, we are interested in the one minimizing $k + n$. We can then define our logical broadcast network as m connected k-ary n-cube networks, where each node belongs to one cube and has links to its counterparts in the $m - 1$ remaining cubes in addition to n links to the neighboring cube nodes (one in each dimension). We label the broadcast ranks by tuples (a, b) with $a \in \{0, \dots, m-1\}$ and $b \in \{0, \dots, k-1\}^n$. Here, a determines the k-ary n-cube the rank is part of while $(b_0, \dots, b_{n-1})$ represents the index of the rank in the respective cube. Inside the cube we allow rank (a, b) the forwarding of messages to its successors $(b_0, \dots, (b_i + 1) \bmod k, \dots, b_{n-1})$ in each dimension $i \in \{0, \dots, n-1\}$. In addition, each rank (a, b) can forward messages to its respective counterparts in the other cubes, i.e. to all ranks (x, b) with $x \neq a$, resulting in a total of $n + m - 1$ outgoing links for each broadcast rank. The number of incoming links is equal, although the cube related ingoing and outgoing links differ for $k \geq 3$. Procedure Broadcast shows in pseudo-code how a broadcast is performed in the network.

Figure 4 depicts an example of the logical interconnect for 16 compute nodes and a peer limit of $L = 6$. Figs. 4b to 4d show the 3-hop broadcast operation that can logically be split into a 2-hop binary 2-cube (i.e. hypercube) broadcast operation and 2^2 associated clique communications between the single 2-cube nodes and their respective counterparts. The maximum number of hops needed for each message is equal to $n(k-1)+1$ while each broadcast rank receives messages from at most L other broadcast ranks.

C. Node Duplication/Synchronization Policy

As stated above, we duplicate frequently visited search tree nodes at all search ranks to reduce the network traffic and speedup the time needed for the computation of each single simulation. This entails the need for regular synchronization of statistics stored with duplicated nodes. We therefore maintain

Procedure Broadcast((a^*, b^*), dimension)

```
Data: $k, n, m \in \mathbb{N}$ define $m$ connected $k$-ary $n$-cubes.
      Nodes are indexed by tuples
      $(a, b_0 b_1 \dots b_{n-1}) \in \{0, \dots, m-1\} \times \{0, \dots, k-1\}^n$.
      dimension $\in \{0, \dots, n-1\}$. $(a, b)$ is the current node
      index and $(a^*, b^*)$ denotes the index of the node that
      initiated the broadcast.
input : Index $(a^*, b^*)$ of the node that initiated the
        broadcast and dimension.
1  if $(a, b) = (a^*, b^*)$ then
2      dimension $\leftarrow n$;
3  end
4  for $0 \le x < m$ and $x \neq a$ do
5      forward message to node $(x, b)$;
6  end
7  for $0 \le i <$ dimension do
8      if $((b_i + 1) \mod k) \neq b_i^*$ then
9          new_dimension $\leftarrow i$;
10         if $((b_i + 2) \mod k) \neq b_i^*$ then
11             new_dimension $\leftarrow i + 1$;
12         end
13         forward message to $(a, b_0 \dots ((b_i + 1)$
           $\mod k) \dots b_{n-1})$ and
14         call Broadcast ($(a^*, b^*)$,new_dimension) there on
           message arrival;
15     end
16 end
```

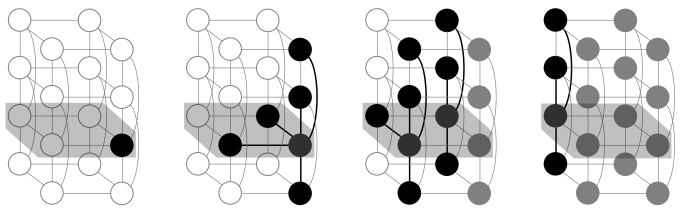

Fig. 4: Example 3-hop broadcast operation with 16 broadcast ranks and a maximum of 6 peers per rank, giving $(k, n, m) = (2, 2, 4)$. The gray box marks the 4 ranks that are used in the binary 2-cube (i.e. hypercube) communication.

additional counters $N^{\Delta}_{s,a}$ and $W^{\Delta}_{s,a}$ next to the $N_{s,a}$ and $W_{s,a}$ that were introduced in Algorithm UCT. $N^{\Delta}_{s,a}$ and $W^{\Delta}_{s,a}$ are used for accounting simulation visits and rewards that still have to be communicated to remote MPI ranks. After communicating them, the Δ-values are incorporated into the corresponding non-Δ variables. Hence, at any time the actual real visit count for a state-action pair (s, a) is obtained by $N^{\Delta}_{s,a} + N_{s,a}$. As we assume that all variables corresponding to children of the same tree node are stored together in memory, all those variables are synchronized at the same time with a single MPI message in order to minimize communication overhead.

We introduce the following four parameters to control the duplication and synchronization of frequently visited search tree nodes:

- N_{dup}: The minimum number of simulations that must have passed a tree node before it is eligible to be shared (i.e. duplicated).
- α: A factor to determine $N_{\text{sync}}(s, a) = \alpha(N^{\Delta}_{s,a} + N_{s,a})$ as the minimum value required for at least one $N^{\Delta}_{s,a}$ to start the synchronization process for node s.
- $N^{\min}_{\text{sync}}$: A lower bound for N_{sync}.
- $N^{\max}_{\text{sync}}$: An upper bound for N_{sync}. Hence, we actually have $N_{\text{sync}}(s, a) = \min(N^{\max}_{\text{sync}}, \max(N^{\min}_{\text{sync}}, \alpha(N^{\Delta}_{s,a} + N_{s,a})))$.

α can be selected with the objective to ensure a uniform reduction of the standard error of the tree node's mean reward per synchronization and allows for less frequent synchronization of more settled node statistics. Let us consider a tuple $(s, a) \in S \times A$ of a node and an action that can be taken in node s with its corresponding values of the number of simulations $N_{s,a}$ that performed action a in node s and the number of those simulations $W_{s,a} \le N_{s,a}$ that lead to a win. We can then compute the mean reward for simulations starting with the transition (s, a) and the corresponding standard error by Equation 5 and Equation 6 respectively.

$$\mu(s, a) = \frac{W_{s,a}}{N_{s,a}} \tag{5}$$

$$SE_{\mu} = \frac{\sigma}{\sqrt{N_{s,a}}} \ , \tag{6}$$

with σ denoting the standard deviation. We can then select a desired rate $p \in (0, 1)$ of reduction of SE_{μ} for each update that yields $SE^{t+1}_{\mu} \leftarrow p \cdot SE^{t}_{\mu}$. If we assume σ to remain unchanged, the following implications yield the maximum amount of simulations δ that can be performed without reducing SE_{μ} by a ratio larger than p:

$$\begin{aligned} \frac{p}{\sqrt{N_{s,a}}} &\le \frac{1}{\sqrt{N_{s,a} + \delta}} \\ \delta &\le N_{s,a}(\frac{1}{p^2} - 1) \\ &\le \alpha N_{s,a} \end{aligned} \tag{7}$$

Hence, given a desired reduction rate $p \in (0, 1)$ we have $\alpha = (1/p^2) - 1$ and synchronization of a node s will be triggered as soon as $N^{\Delta}_{s,a} \ge N_{\text{sync}}(s, a)$ for some $a \in \Gamma(s)$.

D. Distributed Transposition Table

Typical time settings for actual game play limit the time for each player to make all of his moves in a game. Among the obstacles towards actual game play with such time limits is the efficient deletion of parts of the transposition table elements, once a move was made and a new search run is to be started. In such cases, only a single subtree below a direct root-child, the one corresponding to the move that was actually made, must remain in the transposition table, while the other elements are likely not needed any longer. An active deletion or marking of such unneeded elements could easily end up in a lot of communication and might require a substantial amount of time. Saving time for this task allows for using more time with subsequent search runs.

We developed a method for efficient table lookup and insertion that does not require an explicit element deletion phase between subsequent search runs and efficiently implements an inherent locking mechanism for fast replacement of *old* elements. Therefore, we use a time-stamp with every table element to track the element's last access times and override the least recently updated element among the K upcoming indices in a linear probing manner, in case no free table entry was found.

Procedure TableLookup shows the table lookup operation that is used to find the actual table index, given a hash value. The procedure consists of two parts: the first part (line 1–19) looks for the best fitting index and the second part (line 20–39) updates the element's data and ensures that each element is only associated to a single hash value. In case no matching element was found in the first part, an insert takes place at the first free or the least recently updated table index, in case no free one is found. In order to be overridden, an old element must not have been accessed during the current search run. This is ensured by line 8 and in turn guarantees that no data of elements can be deleted that might be accessed by some thread in the current search run. In case we override an old element, we may need exclusive access to this element in order to clear/delete the element's data until it provides valid information again. We declare the element's time-stamp value 0 to be a special lock-value. Line 28 shows, how to use an atomic compare-and-swap operation (atomicCAS) with the lock-value to acquire exclusive access to a table element. The lock is released in line 30 implicitly, when the current value of the lookup counter, that is always greater than 1, is stored in the element's time-stamp variable. Note that in practical implementations, depending on the actual compiler and processor used, it might be necessary to add a compiler and/or CPU store memory barrier, also denoted as fence, right before line 30 to ensure all data was written before the implicit lock is released.

Procedure TableLookup

Data: An array T with $\mathrm{size}(T)$ elements that makes up the transposition table. Each element $T[i]$ is a 2-tuple (h, a) consisting of a hash value h and a volatile integer a tracking the elements last access time. For all i, $T[i].a$ is initialized with 1, indicating an empty table element. A global integer variable lookupCnt is incremented at each table lookup and thereby serves as a clock. startTime holds the value of lookupCnt at the time the search run started.
input : A hash value h
output: A table-index i^* and a return state $r \in \{$FOUND, FULL, BUSY$\}$

```
1  $i^* \leftarrow 0$;
2  $i_{\mathrm{tmp}} \leftarrow 0$; $a_{\mathrm{tmp}} \leftarrow$ MAXINT;
3  $i \leftarrow h \mod \mathrm{size}(T)$;
4  for 10 times do
5      $i \leftarrow \max(i, 1)$;          // index 0 remains unused
6      if $T[i].a = 1$ and $a_{\mathrm{tmp}} > 0$ then
7          $i_{\mathrm{tmp}} \leftarrow i$; $a_{\mathrm{tmp}} \leftarrow 0$;
8      else if $T[i].a <$ startTime and $T[i].a < a_{\mathrm{tmp}}$ then
9          $i_{\mathrm{tmp}} \leftarrow i$; $a_{\mathrm{tmp}} \leftarrow T[i].a$;       // free element
10     end
11     if $T[i].h = h$ and $T[i].a > 1$ then
12         $i^* \leftarrow i$;                // existing element
13         break;
14     end
15     $i \leftarrow (i+1) \mod \mathrm{size}(T)$;
16 end
17 $i \leftarrow i^*$;
18 if $i = 0$ then $i \leftarrow i_{\mathrm{tmp}}$;
19 if $i = 0$ then return FULL;
20 atomicIncrement(lookupCnt);
21 $a \leftarrow T[i].a$;
22 while $a \leq$ startTime do
23     if $a = 0$ then
24         $a \leftarrow T[i].a$; continue;
25     end
26     if $i^* \neq 0$ then
27         atomicCAS ($T[i].a$, lookupCnt, $a$);
28     else if atomicCAS ($T[i].a, 0, a$) then
       // atomicCAS(mem,newval,oldval)
29         clear($T[i]$); $T[i].h \leftarrow h$;  // clear old element
30         $T[i].a \leftarrow$ lookupCnt;          // release lock
31     end
32     $a \leftarrow T[i].a$;
33 end
34 if $T[i].a >$ startTime and $T[i].h = h$ then
35     $T[i].a \leftarrow$ lookupCnt;
36     $i^* \leftarrow i$;
37     return FOUND;
38 end
39 return BUSY;        // another thread updated $T[i]$
```

IV. Experiments

In this section, we present the experimental setup and the results achieved with our Go engine Gomorra that incorporates the UCT-Treesplit algorithm as described in Section III. We start with experimental results concerning the scalability of our parallelization in terms of search quality that were produced by having Gomorra playing against itself and the open-source Go engines FUEGO[2] version 1.1 and Pachi[3] version 10.00 (Satsugen). Then we present more detailed insights by showing the development of achieved simulation rates, measured workloads and network bandwidth usage for varying numbers of MPI ranks. Whenever provided, confidence intervals are given with a 95% confidence level.

A. Setup

Our computer Go engine Gomorra implements several state of the art enhancements over basic UCT and proved its playing strength previously in several games against the currently strongest computer Go programs. In our experiments different instances of Gomorra play against each other on a 19x19 board size, giving each player 10 minutes to make all moves in a game. The distribution of the time used per move computation varies among different phases of the game. Gomorra's time management is loosely based on [33]. We choose the following values for the UCT-Treesplit parameters: $N_{\mathrm{dup}} := 8192$, $N_{\mathrm{sync}}^{\mathrm{min}} := 8$, $N_{\mathrm{sync}}^{\mathrm{max}} := 16$, $\alpha := 0.02$ and $O := 5$. Note that the optimal values will depend on parameters of the compute resources such as network latency and bandwidth as well as on the ratio of processor speed to work package

[2]Available online: `http://fuego.sourceforge.net`
[3]Available online: `http://pachi.or.cz/`

TABLE I: The actual numbers m, n and k used for the m-connected k-ary n-cube logical broadcast network in the experiments.

bc-ranks	m	k	n
1	1	1	1
2	2	1	1
4	4	1	1
8	8	1	1
16	8	2	1
32	8	2	2
64	8	2	3

size. Although reducing N_{dup} decreases the communication overhead for single simulations because less CN hops take place, the overhead for synchronizing shared nodes statistics increases because more nodes are shared. However, few hops per simulation allow for keeping O small. Furthermore, lower values of $N_{\text{sync}}^{\text{min}}$ and $N_{\text{sync}}^{\text{max}}$ lead to increased network traffic as more synchronization messages are sent. For the broadcast operations performed between the broadcast nodes, Table I shows the configuration parameters of the logical communication network for all numbers of broadcast ranks (bc-ranks) used during our experiments. We chose this values with the objective to keep the number of communicating peers for each broadcast rank below 16 while minimizing the number of required hops for each broadcast operation[4]. As each broadcast rank communicates with up to 4 search ranks as stated in Section III-B, 12 peers remain for inter-broadcast nodes communication. Hence, it must always hold that $n + m - 1 \leq 12$.

For our experiments we use up to 160 CNs of the OCuLUS cluster[5], each one equipped with 2 Intel Xeon E5-2670 CPUs (16 cores in total) running at 2.6 GHz and 64 GByte of main memory. The CNs are connected by a 4xQDR Infiniband network. We use OpenMPI version 1.6.4 for message passing.

B. Overhead Distributions

For the development of UCT-Treesplit, one of the assumptions was that the computational overhead for playout work packages excels the time requirement of the selection and update work packages. Figure 5 shows the average absolute time requirements in seconds for the work packages when running Gomorra in parallel on 16(+4) compute nodes, i.e. 16 worker and 4 associated broadcast nodes. As each CN is equipped with 2 CPUs we actually run 32(+8) MPI ranks. In the diagram, the time requirements of the different work packages are stacked. Hence, the total amount of time spent on average for the first move is about 2 seconds and about 1.5 seconds for move 300. Gomorra used a maximum of 10 minutes per game and player to play all its moves. As a relict of Gomorra's time management (based on [33]) we assign more time to the mid-game phase, which results in bell-shaped stacked curves shown in Figure 5. In addition

[4]OpenMPI version 1.6 recommends 16 as the maximum number of peers to use for eager RDMA communication.

[5]The system specification is available online: `http://pc2.de/`

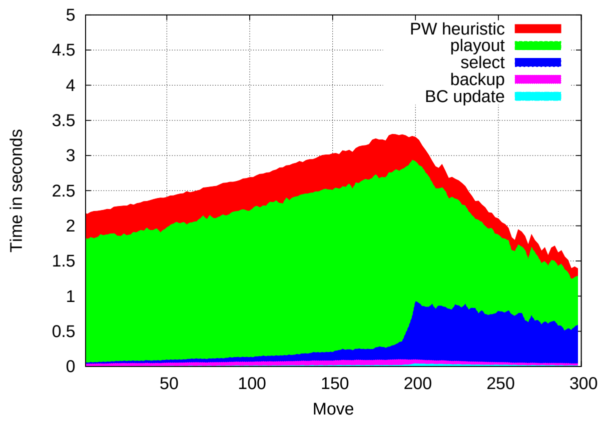

Fig. 5: Distribution of computation time overheads in distributed version

to the work packages we introduced in Section III, we also explicitly plotted the time requirement for the computation of some heuristics that are used to assign prior values to statistics of recently discovered search tree nodes. The curve labeled with *BC update* shows the overhead generated by the need of incorporating incoming synchronization messages received from the associated broadcast rank. As can be seen, these additional costs are negligible when using 16(+4) CNs.

C. Performance and Scalability

In this section we examine our system's overall performance characteristics by having Gomorra play games against itself as well as against the open-source Go programs FUEGO and Pachi with varying numbers of compute nodes. Like Gomorra, FUEGO and Pachi are MCTS based Go programs. We configured FUEGO to compute a constant amount of 250.000 simulations per move decision in order to leverage its playing strength to be comparable with Gomorra. Pachi was configured to play on a single CN using 16 cores with identical time restrictions as Gomorra. With this configuration, Pachi computed about 30.000 simulations per second on the empty 19x19 board.

The most important measure for the performance of a parallel MCTS algorithm and its scalability is the development of playing strength with an increasing number of CNs. We measured the playing strength in ELO[34]. A win rate of p of one program instance over the other, translates to a relative ELO value of $\log_{10}(p/(1-p)) \cdot 400$. Figure 6 shows this development of playing strength for the sequential version of Gomorra when playing with varying number of simulations per move. Figure 7 shows the according graph for the parallel version when using varying numbers of CNs. All games played for these figures were games against Gomorra itself. The reference configuration for the sequential version computed a fixed number of 10240 simulations per move. The reference configuration for the parallel version used 8(+2) CNs. Here, we write 8(+2) for 8 search CNs (i.e. 16 search ranks) and

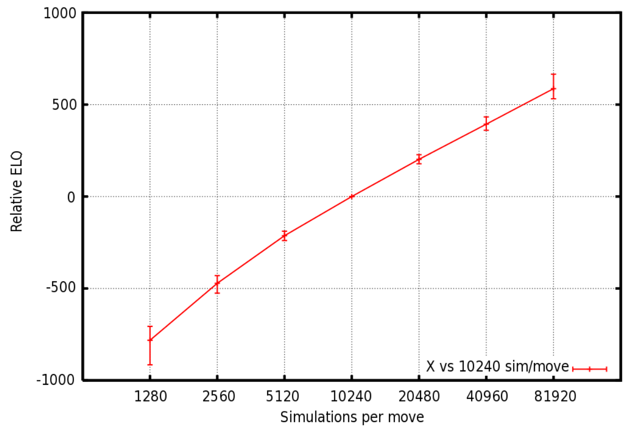

Fig. 6: Evaluation of Gomorra's playing strength when playing against itself on a single compute node.

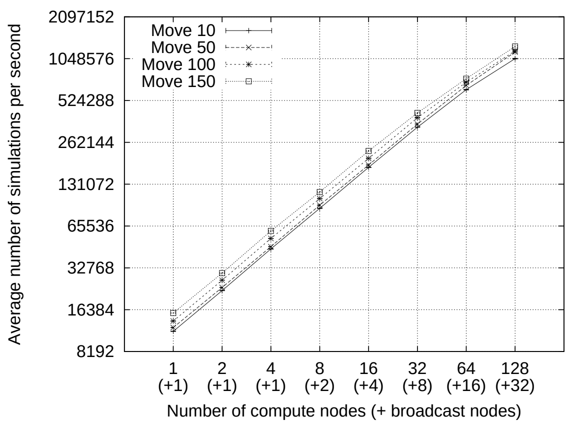

Fig. 8: Scalability of simulation rate with increasing number of CNs at different game phases.

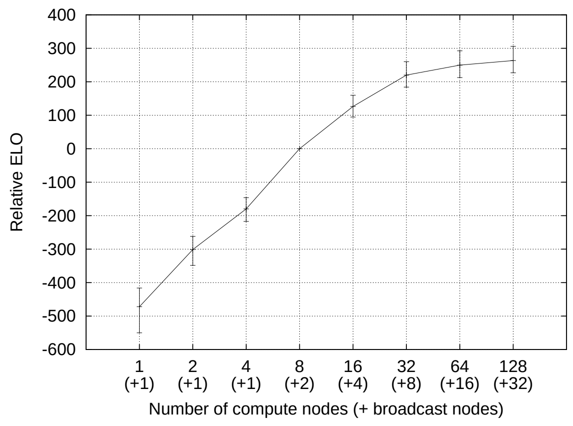

Fig. 7: Evaluation of Gomorra's playing strength when playing against itself using parallel UCT-Treesplit for varying numbers of compute nodes.

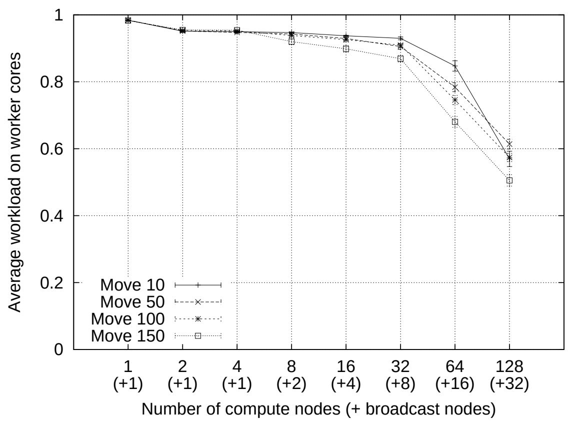

Fig. 9: Average workload at worker cores with increasing number of CNs at different game phases.

2 broadcast CNs (i.e. 4 broadcast ranks). As we can see, the sequential version can improve by more than 1000 ELO from 6 doublings of the number of simulations performed per move whereas the parallel version improves by about 700 ELO from 6 doublings of the number of CNs. In order to relate both curves and to understand why the parallel version does not also improve by more than 1000 ELO, we first want to know if 6 doublings of CNs also results in 6 doublings of the number of simulations that are computed for each move decision. Therefore, we look at the simulation rate, i.e. the number of simulations that can be computed per time unit for varying numbers of CNs.

Figure 8 shows the scalability of the simulation rate and Figure 9 the development of the average workload measured at the worker cores. Both diagrams show 4 curves for measurements at different phases of the game: At moves number 10, 50, 100 and 150. We plot curves for different game phases, because the playouts become shorter in later game phases and this impacts the distribution of computational load among the different work packages of each single simulation. We can see that the number of simulations scales uniformly for up to 32(+8) CNs and observe a slightly degraded scalability for more nodes. Figure 9 in contrast shows a more remarkable drop of measured workloads when using more than 32(+8) CNs. This observation also matches the reduction of playing strength scalability observed in Figure 7.

Observing decreasing workloads leads to the assumption that either the distribution of work packages or the synchronization of shared tree nodes or both becomes more challenging with increasing number of CNs. In order to explain the observed numbers, we examine the actual communication related requirements in terms of used bandwidth and computational overhead. Figure 10 shows the average bandwidth

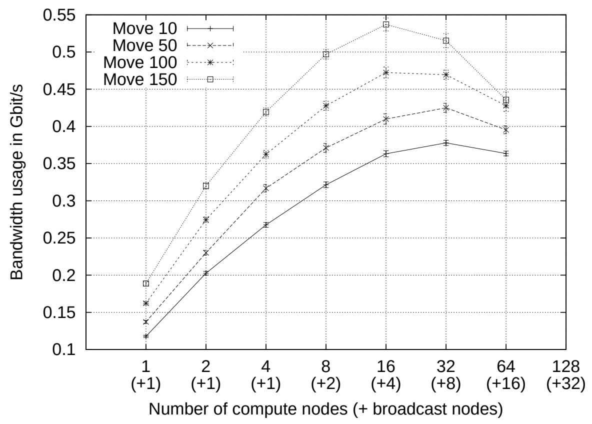

Fig. 10: Average outgoing bandwidth usage at search ranks with varying numbers of computer nodes.

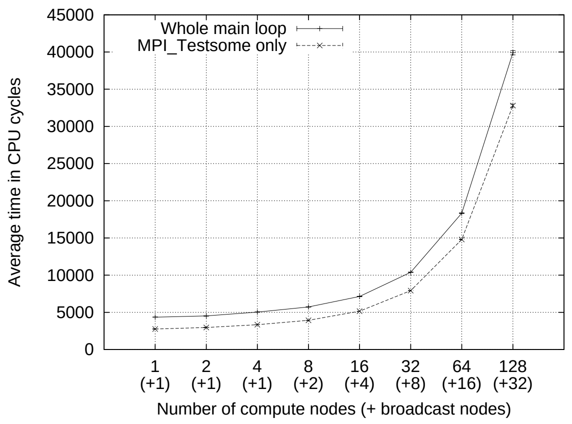

Fig. 12: Time requirement of MPI_Testsome in comparison with the entire message main loop on search ranks.

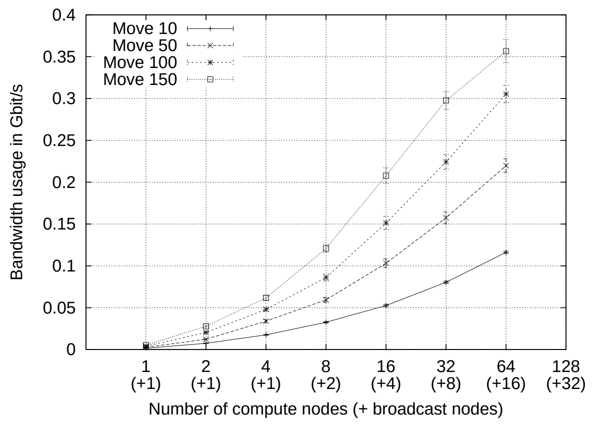

Fig. 11: Average outgoing bandwidth usage at broadcast ranks with varying numbers of computer nodes.

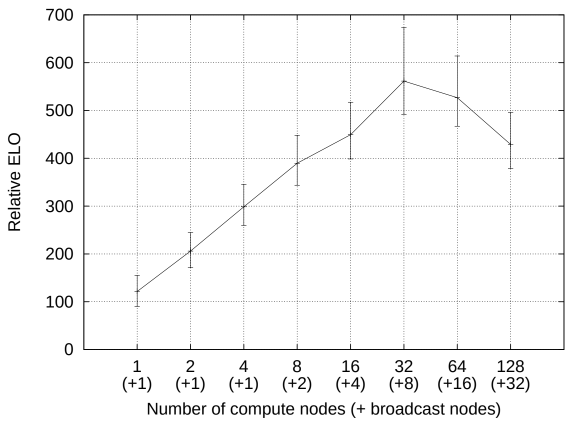

Fig. 13: Evaluation of Gomorra's playing strength when playing against the open-source Go program FUEGO.

usage measured on a worker MPI rank for outgoing messages. The corresponding measures for broadcast ranks are shown in Figure 11. In both cases, the required bandwidth remains well below the available bandwidth for any configuration. Using 4xQDR Infiniband as in our case, the theoretically available bandwidth per CN is about 32 Gbit/s. Recall that each CN is used by two MPI ranks, hence the total amount of bandwidth available per MPI rank is about 16 Gbit/s. Accordingly, the bandwidth requirements stay well below their limits.

In order to estimate the computational overhead related to communication on worker and broadcast CNs, we measure the average main loop iteration time, as we assume that most of the communication related computation time is located in this loop. In fact, we assigned a whole core solely for running this loop. Also computations performed by the MPI library are bound to this core and hence, will inherently be included in our measurements. Some of the communication related work, however, is not included as some parts, such as the composition of messages, were shifted to the worker cores.

Figure 12 shows the aforementioned measurements of the main loop's average turnaround time on the worker CNs with varying number of CNs. During each iteration, the main loop calls the MPI function `MPI_Testsome` once in order to get informed about newly received messages and about the completion of pending asynchronous send operations. In addition it collects all messages supplied through the worker cores' transfer buffers (cf. Figure 2) and asynchronously sends them to the corresponding destination ranks. Furthermore all received simulation related messages are forwarded to the corresponding workers' transfer buffers. Additionally, incoming duplication and synchronization messages for shared tree nodes are immediately processed and corresponding synchronization messages are asynchronously sent to the MPI rank's associated broadcast rank, represented by the cache handler

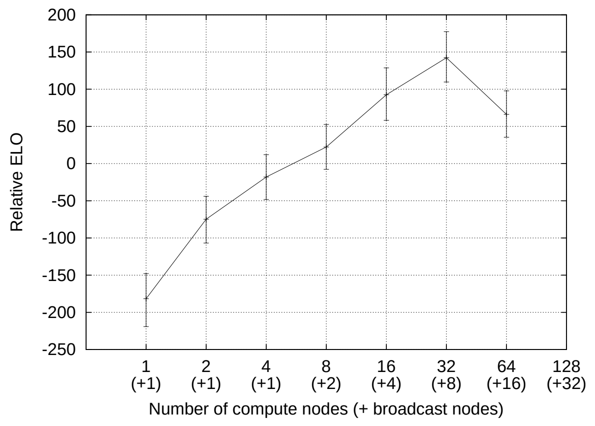

Fig. 14: Evaluation of Gomorra's playing strength when playing against the open-source Go program Pachi.

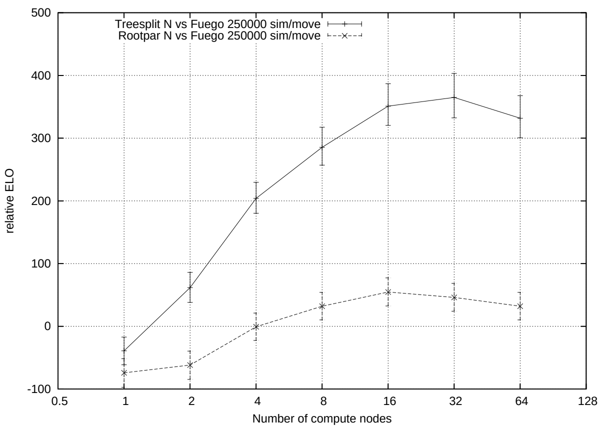

Fig. 15: Evaluation of Gomorra's playing strength when playing against the open-source Go program FUEGO using UCT-Treeplit and Root-Parallelization.

in Figure 2. The figure contains an additional curve for the average time spend inside the `MPI_Testsome` routine, that consumes by far most of the time. This stems from the fact that OpenMPI uses the call to `MPI_Testsome` not only to poll message buffers to recognize completed RDMA communications but also for other library internal work. We can see that `MPI_Testsome` is responsible for steadily increasing turnaround times of the main communication loop with increasing numbers of CNs. This is likely the case because RDMA communication requires the MPI library on each rank to provide a number of receive buffers for each peer that regularly sends messages to this rank, and to poll some memory locations associated to those buffers in order to recognize the completion of incoming messages. OpenMPI allows for restricting the use of RDMA communication to a limited number of peers to bound the polling time and the memory consumption for the receive buffers. However, limiting this number comes with a significant increase of the communication latency and the protocol for non-RDMA communications might require a CPU involvement a number of times in the course of each single message transfer. This in turn impairs the overlapping of communication with computation, as communication can only progress when we explicitly assign execution time to the MPI library (e.g. by a call to `MPI_Testsome`). Our experiments showed that the aforementioned drawbacks of partly non-RDMA communication heavily outweigh the increasing polling time and memory consumptions when using RDMA for communicating with all peers. Obviously, for even larger systems the use of RDMA communication has to be restricted.

Finally, in order to make sure that the scalability of our parallelization in terms of playing strength is not restricted to self play experiments, we measured the playing strength development with varying numbers of CNs against the open-source Go engines FUEGO and Pachi. Figure 13 shows the results for FUEGO. Also against FUEGO we measured a significant improvement in playing strength when using more CNs for up to 32(+8) CNs. However, gaining about 400 Elo, the improvement is considerably less than in self-play where we achieved more than 700 Elo (see Figure 7). In general, improvements over modified versions of the same Go program are known to be better than improvements over other Go programs. Effects comparable to our measurements were also observed and published by e.g. [26]. Even further, we see that the drop of workload and the degrading simulation rate scalability leads to a more severe impairing for 64(+16) CNs and more. We assume this impairing that actually leads to a drop of playing strength for large systems, originates in higher ramp-up times for non-self-play games. In self-play games, it is likely that both program instances are searching most of the time in the same portion of the search space as both instances use, e.g., the same heuristics and playout policies. This increases the chance of being able to reuse parts of the transposition table content of previous search runs and thereby leads to reduced ramp-up times. This is especially true in case the root node was already duplicated as all search ranks can then immediately start simulations. Figure 14 shows the corresponding results for Pachi. Also here, Gomorra gains playing strength for up to 32(+8) CNs and shows a slight drop when using more CNs.

A slight modification to UCT-Treesplit makes it behave similarly to Root-Parallelization. This is achieved by changing function $i(s)$, originally defined in Formula 1 on page 4 to always return the MPI rank at which i was called. This modification keeps simulations from moving to other ranks. As a result, frequently visited nodes still become duplicated and synchronized while each CN grows its own search tree. Figure 15 shows the development of playing strength for varying numbers of CNs when having Gomorra play against FUEGO using UCT-Treesplit and a form of Root-Parallelization as described above. Recall that two MPI ranks are running on each CN, giving rise to the fact that the playing strength obtained by both parallelizations differ already when using

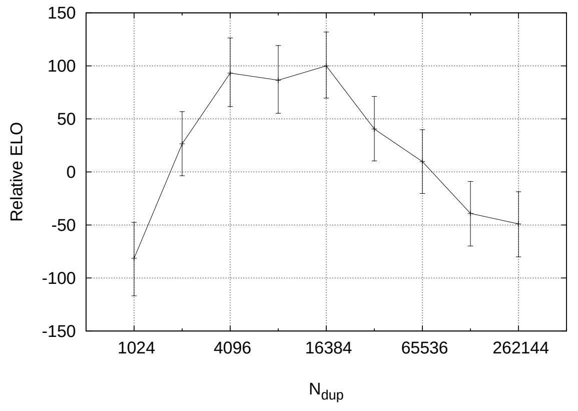

Fig. 16: Evaluation of Gomorra's playing strength when playing against Pachi using UCT-Treeplit and varying values for parameter N_{dup}.

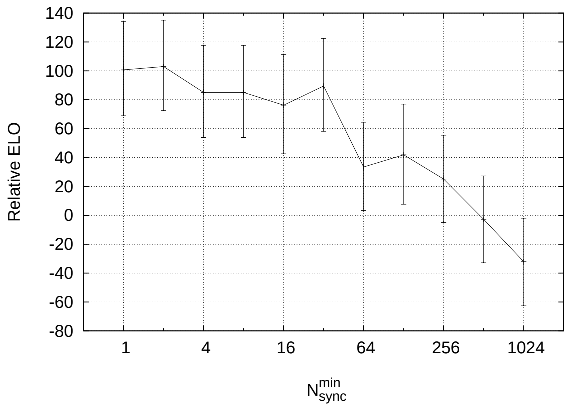

Fig. 17: Evaluation of Gomorra's playing strength when playing against Pachi using UCT-Treeplit and varying values for parameters $N_{\mathrm{sync}}^{\mathrm{min}}$ and $N_{\mathrm{sync}}^{\mathrm{max}} = 2N_{\mathrm{sync}}^{\mathrm{min}}$.

a single CN only.

D. Effect of Parameters

We conducted further experiments to analyze the impact of UCT-Treesplit's parameters N_{dup}, $N_{\mathrm{sync}}^{\mathrm{min}}$ and $N_{\mathrm{sync}}^{\mathrm{max}}$ on the playing strength. We took the parameter settings used for the experiments presented above ($N_{\mathrm{dup}} := 8192$, $N_{\mathrm{sync}}^{\mathrm{min}} := 8$, $N_{\mathrm{sync}}^{\mathrm{max}} := 16$, $\alpha := 0.02$ and $O := 5$) as the reference point and empirically analyzed the effect of changing a single parameter. All experimental results presented below were obtained from games between Gomorra using 32(+8) CNs and Pachi. As in the experiments presented above, each program had 10 minutes to make all its moves in a game.

Figure 16 shows the development of playing strength for varying values of N_{dup}. We observe a bell-shaped curve. Low values of N_{dup} result in larger portions of the tree that get duplicated. This results in more statistics that are less accurate due to delayed synchronization. For very high values of N_{dup} it happens that frequently visited tree nodes that are not yet duplicated lead to high contention on single CNs, leading in turn to severe load imbalances and thereby to reduced simulation rates.

The development of playing strength for varying values of the parameters $N_{\mathrm{sync}}^{\mathrm{min}}$ and $N_{\mathrm{sync}}^{\mathrm{max}}$ is depicted in Figure 17. While changing the value of $N_{\mathrm{sync}}^{\mathrm{min}}$ as shown on the x-axis, we set $N_{\mathrm{sync}}^{\mathrm{max}} = 2N_{\mathrm{sync}}^{\mathrm{min}}$. We see that a reduction of the synchronization rate by increasing the value of $N_{\mathrm{sync}}^{\mathrm{max}}$ leads to reduced playing strength. This makes sense as statistics become less accurate when increasing synchronization times. Mind that network traffic increases when synchronizing more frequently, i.e. for low values of $N_{\mathrm{sync}}^{\mathrm{min}}$ and $N_{\mathrm{sync}}^{\mathrm{max}}$.

V. Discussion

A. Enhancements to Previous Work

Compared to our previous publication on UCT-Treesplit [18], we present significantly improved experimental results. This is due to a number of reasons that can be summarized as follows:

- We reduced the number of duplicated nodes and synchronize more often. This is possible by the use of additional hardware (i.e. broadcast nodes) for assisting the distribution of synchronization data.
- The load balancing was improved by redistributing computation intensive work packages on the observation of load imbalances.
- An improved policy for reuse and lazy deletion of existing transposition table entries allows for greatly saving time between distinct searches during game play.
- In our former publication [18] we limited the number of CPUs used per CN to only one. We are now able to use more CPUs per CN by assigning a proper MPI rank to each CPU.
- The cluster used for experimentation is slightly more powerful. Each CN contains 2 Intel Xeon E5-2670 CPUs (16 cores in total vs. 12 cores in our former experiments) and nodes are connected by a 4xQDR Infiniband network (was 4xSDR before).

B. Comparison of MCTS Parallelizations

A fair comparison of different MCTS parallelizations is challenging, if possible at all. The performance of different parallelization approaches depends on the actual hardware platform used, the specific search domain and of course the many specific implementation details that are often not presented in detail in publications.

Looking at message passing approaches, solely using a different MPI version of the same distributor might already have a severe impact on the performance of a single parallelization approach. We therefore restricted our experiments to a comparison with one form of Root-Parallelization in order to get an idea of the gain from using a single large game tree instead of growing a number of tiny trees. While this allows us to derive a reasonable and expressive comparison of approaches, it also leaves room for more involved comparisons between UCT-Treesplit and Root-Parallelization in general.

VI. Conclusion and Future Work

In this paper, we present a number of algorithmic and methodical enhancements to parallelize MCTS on distributed memory HPC systems. In addition to previous results with our MCTS parallelization UCT-Treesplit [18], we develop and integrate an efficient distributed transposition table and advocate the use of additional compute nodes to support necessary all-to-all communications and thereby allow for multi-stage message merging. In conjunction with a sound policy that handles data sharing and synchronization frequencies we were able to greatly increase the scalability of UCT-Treesplit. To our knowledge, UCT-Treesplit is currently the best scaling distributed MCTS implementation. We evaluated the behavior of our parallelization in a high-end Go engine and show that, for the game of Go, UCT-Treesplit scales up to 128(+32) compute nodes in self-play experiments.

A closer look at performance measurements in terms of bandwidth usage and communication related computation overheads for varying system configurations points to future improvements. Presently, it appears that we reached the limits of RDMA based tiny-message communication. As one future direction, we therefore focus on limiting the number of communication peers per search rank as already done for our broadcast ranks. Even further we need to fully understand and prove the source of observed workload drops when using 128 and more MPI ranks.

Acknowledgment

We want to thank the staff of the two compute centers *Paderborn Center for Parallel Computing* (PC^2) and *Regionales Rechenzentrum der Universität Köln* (RRZK), namely Axel Keller (PC^2), Jens Simon (PC^2), Stefan Borowski (RRZK) and Viktor Achter (RRZK) for their kind support and numerous helpful comments on our work. A huge number of preliminary experiments were performed on the CHEOPS HPC cluster [6]. We also thank the unkown reviewers for extensive comments that helped to further improve this article.

References

[1] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry, “Challenges in Parallel Graph Processing,” *Parallel Processing Letters*, vol. 17, 2007.

[2] D. E. Knuth and R. W. Moore, “An Analysis of Alpha-Beta Pruning,” in *Artificial Intelligence*, vol. 6, no. 1. North-Holland Publishing Company, 1975, pp. 293–327.

[3] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A Survey of Monte Carlo Tree Search Methods,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012.

[4] B. Arneson, R. B. Hayward, and P. Henderson, “Monte Carlo Tree Search in Hex,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 251–258, Dec. 2010.

[5] F. Teytaud and O. Teytaud, “On the Huge Benefit of Decisive Moves in Monte-Carlo Tree Search Algorithms,” in *IEEE Conference on Computational Intelligence and Games*, no. 1, 2010, pp. 359–364.

[6] See `http://rrzk.uni-koeln.de/cheops.html`

[6] P. Hingston and M. Masek, “Experiments with Monte Carlo Othello,” in *Proc. IEEE Congr. Evol. Comput.*, 2007, pp. 4059–4064.

[7] R. J. Lorentz, “Amazons Discover Monte-Carlo,” in *Int. Conf. on Computers and Games*, ser. LNCS, vol. 5131, 2008, pp. 13–24.

[8] L. Kocsis and C. Szepesvári, “Bandit based Monte-Carlo Planning,” in *ECML*, ser. LNCS/LNAI, vol. 4212, 2006, pp. 282–293.

[9] S. Baba, Y. Joe, A. Iwasaki, and M. Yokoo, “Real-Time Solving of Quantified CSPs Based on Monte-Carlo Game Tree Search,” in *Proc. 22nd Int. Joint Conf. Artif. Intell.*, 2011, pp. 655–662.

[10] H. Nakhost and M. Müller, “Monte-Carlo Exploration for Deterministic Planning,” in *Proc. 21st Int. Joint Conf. Artif. Intell.*, 2009, pp. 1766–1771.

[11] R. Gaudel and M. Sebag, “Feature Selection as a One-Player Game,” in *Proc. 27th Int. Conf. Mach. Learn.*, 2010, pp. 359–366.

[12] T. Mahlmann, J. Togelius, and G. N. Yannakakis, “Towards Procedural Strategy Game Generation: Evolving Complementary Unit Types,” in *Proc. Applicat. Evol. Comput.* , ser. LNCS, vol. 6624, 2011, pp. 93–102.

[13] D. Silver, “Reinforcement Learning and Simulation-Based Search in Computer Go,” Ph.D. dissertation, University of Alberta, 2009.

[14] C. Donninger, A. Kure, and U. Lorenz, “Parallel Brutus: The First Distributed, FPGA Accelerated Chess Program,” in *18th International Parallel and Distributed Processing Symposium.* IEEE Computer Society, Apr. 2004.

[15] K. Himstedt, U. Lorenz, and D. P. F. Möller, “A Twofold Distributed Game-Tree Search Approach Using Interconnected Clusters,” in *Euro-Par*, ser. LNCS, vol. 5168. Springer, 2008, pp. 587–598.

[16] A. Bourki, G. M. J.-B. Chaslot, M. Coulm, V. Danjean, H. Doghmen, J.-B. Hoock, T. Hérault, A. Rimmel, F. Teytaud, O. Teytaud, P. Vayssiére, and Z. Yu, “Scalability and Parallelization of Monte-Carlo Tree Search,” in *International Conference on Computers and Games*, ser. LNCS, vol. 6515, 2011, pp. 48–58.

[17] K. Yoshizoe, A. Kishimoto, T. Kaneko, H. Yoshimoto, and Y. Ishikawa, “Scalable Distributed Monte-Carlo Tree Search,” in *SOCS*, D. Borrajo, M. Likhachev, and C. L. López, Eds. AAAI Press, 2011, pp. 180–187.

[18] T. Graf, U. Lorenz, M. Platzner, and L. Schaefers, “Parallel Monte-Carlo Tree Search for HPC Systems,” in *Proceedings of the 17th International Conference, Euro-Par*, ser. LNCS, vol. 6853. Springer, Heidelberg, Aug. 2011, pp. 365–376.

[19] G. M. J.-B. Chaslot, M. H. Winands, and H. J. van den Herik, “Parallel Monte-Carlo Tree Search,” in *Conference on Computers and Games*, 2008, pp. 60–71.

[20] R. Coulom, “Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search,” in *Int. Conf. on Computers and Games*, ser. LNCS, vol. 4630. Springer, 2007, pp. 72–83.

[21] P. Auer, N. Cesa-Bianchi, and P. Fischer, “Finite-Time Analysis of the Multiarmed Bandit Problem,” in *Machine Learning*, vol. 47. Kluwer Academic, 2002, pp. 235–256.

[22] M. Enzenberger and M. Müller, “A Lock-free Multithreaded Monte-Carlo Tree Search,” in *12th International Conference on Advances in Computer Games*, ser. LNCS, vol. 6048. Springer-Verlag, 2010, pp. 14–20.

[23] R. M. Hyatt and T. Mann, “A lockless transposition table implementation for parallel search chess engines,” *ICGA Journal*, vol. 25, no. 1, pp. 36–39, 2002.

[24] T. Cazenave and N. Jouandeau, “On the Parallelization of UCT,” in *ICGA Computer Games Workshop*, Jun. 2007, pp. 93–101.

[25] ——, “A Parallel Monte-Carlo Tree Search Algorithm,” in *International Conference on Computer and Games*, ser. LNCS, vol. 5131, 2008, pp. 72–80.

[26] P. Baudis and J.-L. Gailly, “PACHI: State of the Art Open Source Go Program,” in *Advances in Computer Games 13*, ser. LNCS, H. J. van den Herik and A. Plaat, Eds., vol. 7168. Springer, 2012, pp. 24–38.

[27] J. W. Romein, A. Plaat, H. E. Bal, and J. Schaeffer, “Transposition Table Driven Work Scheduling in Distributed Search,” in *National Conference on Artificial Intelligence*, 1999, pp. 725–731.

[28] R. Feldmann, P. Mysliwietz, and B. Monien, “Distributed Game Tree Search on a Massively Parallel System,” in *Data Structures and Efficient Algorithms*, ser. LNCS, vol. 594. Springer-Verlag, 1992, pp. 270–288.

[29] U. Lorenz, “Parallel Controlled Conspiracy Number Search,” in *Proc. of Int. Euro-Par Conf. (Euro-Par)*, ser. LNCS, vol. 2400, 2002, pp. 420–430.

[30] ——, “Controlled Conspiracy Number Search,” Ph.D. dissertation, University of Paderborn, Dec. 2000.

[31] A. Kishimoto and J. Schaeffer, “Transposition Table Driven Work Scheduling in Distributed Game-Tree Search,” in *Canadian Confer-*

ence on Artificial Intelligence, ser. LNCS, vol. 2338. Springer Berlin/Heidelberg, 2002, pp. 56–68.

[32] R. B. Segal, "On the Scalability of Parallel UCT," in *International Conference on Computer and Games*, 2010, pp. 36–47.

[33] S.-C. Huang, R. Coulom, and S.-S. Lin, "Time Management for Monte-Carlo Tree Search Applied to the Game of Go," in *International Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, ser. LNCS, vol. 6515, 2011, pp. 462–466.

[34] A. E. Elo, *The Rating of Chessplayers, Past and Present*. New York: Arco Publishing, 1986.

Lars Schaefers holds a research position at the Computer Engineering Group at the University of Paderborn. He holds bachelor, diploma and PhD degrees in Computer Science (University of Paderborn, 2006, 2008 and 2014). His research interests include computer-Go, monte-carlo search methods, machine learning, and parallel computing. He is the lead programmer of the world class Go engine Gomorra and a member of the board of the Paderborn Center for Parallel Computing.

Marco Platzner is Professor for Computer Engineering at the University of Paderborn. Previously, he held research positions at the Computer Engineering and Networks Lab at ETH Zurich, Switzerland, the Computer Systems Lab at Stanford University, USA, the GMD - Research Center for Information Technology in Sankt Augustin, Germany, and the Graz University of Technology, Austria. Marco Platzner holds diploma and PhD degrees in Telematics (Graz University of Technology, 1991 and 1996), and a "Habilitation" degree for the area hardware-software codesign (ETH Zurich, 2002). His research interests include reconfigurable computing, hardware-software codesign, and parallel architectures. He is a senior member of the IEEE, a member of the ACM, a member of the boards of the Paderborn Center for Parallel Computing and the Paderborn Institute of Advanced Studies in Computer Science and Engineering, a faculty member of the International Graduate School Dynamic Intelligent Systems of the University of Paderborn, and of the Advanced Learning and Research Institute at Universita' della Svizzera Italiana (USI), in Lugano.