# Parallel Monte-Carlo Tree Search for HPC Systems

Tobias Graf[1,⋆], Ulf Lorenz[2], Marco Platzner[1], and Lars Schaefers[1,⋆⋆]

[1] University of Paderborn, Paderborn, Germany
{slars@,platzner@,tobiasg@mail.}uni-paderborn.de
[2] TU Darmstadt, Darmstadt, Germany
lorenz@mathematik.tu-darmstadt.de

**Abstract.** Monte-Carlo Tree Search (MCTS) is a simulation-based search method that brought about great success to applications such as Computer-Go in the past few years. The power of MCTS strongly depends on the number of simulations computed per time unit and the amount of memory available to store data gathered during simulation. High-performance computing systems such as large compute clusters provide vast computation and memory resources and thus seem to be natural targets for running MCTS. However, so far only few publications deal with parallelizing MCTS for distributed memory machines. In this paper, we present a novel approach for the parallelization of MCTS which allows for an equally distributed spreading of both the work and memory load among all compute nodes within a distributed memory HPC system. We describe our approach termed UCT-Treesplit and evaluate its performance on the example of a state-of-the-art Go engine.

**Keywords:** UCT, HPC, Monte-Carlo Tree Search, distributed memory.

## 1 Introduction

Monte-Carlo tree search (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. MCTS learns a value function for game states by consecutive simulation of complete games of self-play using randomized policies to select moves for either player. Especially in the field of Computer Go, an Asian two-player board game, MCTS highly dominates over traditional methods such as $\alpha\beta$ search [12]. MCTS may be classified as a sequential best-first search algorithm [17], 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. This dependency and the need to store and share the statistics among

⋆ Authors are listed in alphabetical order.

⋆⋆ This work is supported by Microsoft Research Ltd. through a Phd-Scholarship.

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6853, Part II, pp. 365–376, 2011.

all computation entities makes parallelization of MCTS for distributed memory environments a highly challenging task.

Parallelization of traditional $\alpha\beta$ search is a pretty well solved problem, e.g., see [6][10]. While for $\alpha\beta$ search it is sufficient to map the actual move stack to memory, MCTS requires us to keep a consecutively growing search tree representation in memory. On SMP machines, sharing a single search tree representation in memory is straight-forward and has already been proven to be very effective for MCTS parallelization [3][7]. However, sharing a search tree as the central data structure in a distributed memory environment is rather involved and only few approaches have been investigated so far [2].

In this paper, we present a novel approach for the parallelization of MCTS for distributed high-performance computing (HPC) systems. Our algorithm spreads a single search tree representation among all compute nodes (CNs) and guides simulations across CN boundaries using message passing. We map search tree nodes to randomized hash values, and the hash values to CNs in an equally distributed fashion which makes spreading tree nodes a straight-forward procedure [14][8]. A comparable approach used with traditional $\alpha\beta$ search was termed transposition driven scheduling (TDS) [15]. Computing more simulations in parallel than cores are available allows us to overlap communication times with additional simulations. We evaluate the performance of our parallelization technique on a real-world application, our high-end Go engine Gomorra. Gomorra has proven its strength at the Computer Olympiad 2010 in Kanazawa, Japan. In summary, we make the following contributions:

- Our algorithm makes efficient use of *all memory resources* available in the cluster. This is in strong contrast to formerly investigated parallelization methods that either extensively duplicate data [3][2] or resort to using only a fraction of a cluster's overall memory capacity.
- We provide a *flexible parallelization framework* not only for MCTS but for history-dependent simulation processes in general. Our framework is adjustable to light and heavy-weight simulations as well as for different kinds of networks, targeting an optimal exploitation of available resources.
- Compared to other MCTS parallelizations for distributed memory systems, our approach *reduces the loss of information* that inevitably results from parallelizing simulations.

The reminder of the paper is structured as follows: Section 2 introduces to the basic MCTS algorithm and reviews related work in parallelizing MCTS. Our novel parallelization approach for MCTS is presented in Section 3. Section 4 evaluates our algorithm and details the experimental setup and results achieved. Section 5 concludes the paper and gives an outlook to future work.

## 2 MCTS: 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 basic MCTS and then focus on efforts to parallelize MCTS algorithms.

### 2.1 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 [13]. The so-called UCT algorithm (short for *Upper Confidence Bounds applied to trees* [13]) 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**: $G := (S, A, \Gamma, \delta, r)$, with $S$ being the set of all possible states (i.e. game positions), $A$ the set of actions (i.e. moves) that lead from one state to the next, a function $\Gamma : S \to \mathcal{P}(A)$ determining the subset of available actions at a state, the transition function $\delta : S \times A \to \{S, \emptyset\}$ specifying the follow-up state for a state-action pair where $\delta(s) = \emptyset$ iff $s \notin \Gamma(s)$ and a reward function $r : S_t \to \{0, 1\}$ assigning a binary reward to each terminal state $S_t := \{s \in S | \Gamma(s) = \emptyset\}$. 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 $N_s := \sum_{a \in \Gamma(s)} N_{s,a}$.

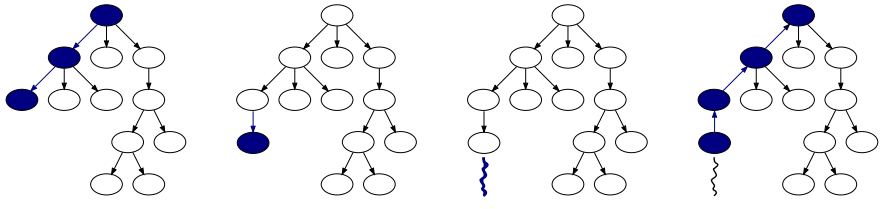
**input** : A state $s_0 \in S$ and a time limit

**output**: An action $a \in \Gamma(s_0)$

$T \leftarrow s_0$; $t \leftarrow 0$;
**while** *Time available* **do**
  **if** $s_t \in T$ **then**
    `// in-tree policy:`
    $a_{t+1} \leftarrow \text{argmax}_{a \in \Gamma(s)}$`actionValue`$(W_{s_t,a}, N_{s_t,a}, N_s)$;
    $N_{s_t,a_{t+1}} \leftarrow N_{s_t,a_{t+1}} + 1$;
    $s_{t+1} \leftarrow \delta(s_t, a_{t+1})$;
    $t \leftarrow t + 1$;
  **else**
    $T \leftarrow T \cup s$; `// Expand memory representation of search tree`
    reward $\leftarrow$ `playout` $(s_t)$;
    `update` (reward); `// Update all` $W_{s_i,a_i}$ `for` $0 \leq i \leq t$ `appropriately`
    $t \leftarrow 0$; `// Start a new simulation`
  **end**
**end**
**return** $\text{argmax}_{a \in \Gamma(s_0)}, N_{s_0,a}$;
**Function** `actionValue` $(w, v, V)$
  **return** $\frac{w}{v} + 2C\sqrt{\frac{2\log(V)}{v}}$; `// 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 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 [4] 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 randomized heuristic policy $\pi_H$ is used for action selection until a terminal game position, i.e., a leaf of the real game tree, is reached. We call the randomized heuristic policy *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 Fig. 1. In Section 3 we will use these steps to form work packages that can be distributed across a cluster.



**(a)** in-tree **(b)** expand **(c)** playout **(d)** update

**Fig. 1.** Building blocks of MCTS

UCT handles the task of selecting an action as a multi-armed-bandit problem (MAB) and is designed following prior investigations on MAB in [1]. As shown in Algorithm UCT, all data needed by UCT to select an action in the in-tree phase are the simulation statistics of all possible actions available at that states. 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.

Researchers investigating the Go playout policy $\pi_H$ follow two principal directions. While some researchers concentrate on handcrafted, computationally light-weight policies [4][9], others advocate more heavy-weight policies learned off-line from records of expert games or games of self-play with many simulations [5][18][11]. In either case, 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 Go, the playout step is typically dominating 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. The runtime dominance of the playout step is even more pronounced when heavy-weight playout policies are being used.

### 2.2 Parallelization of MCTS

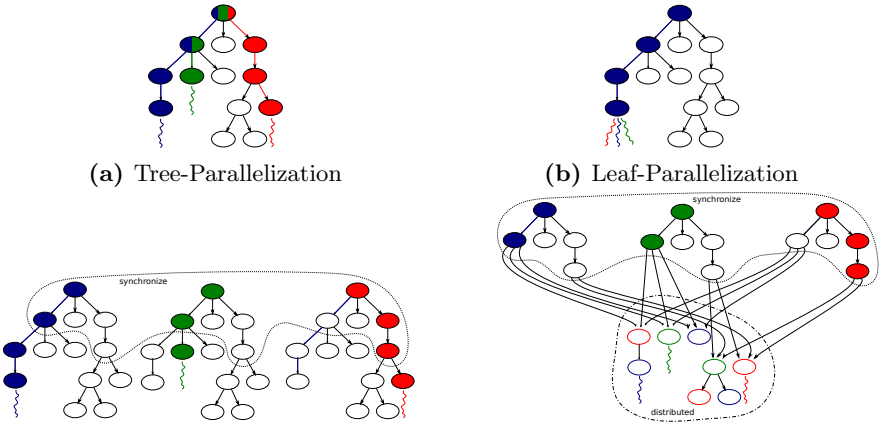
The most common parallelization methods presented so far are termed Tree-Parallelization, Leaf-Parallelization and Slow-Tree-Parallelization [3], [2]. Fig. 2 illustrates these methods together with our algorithm UCT-Treesplit proposed in this paper. Tree-Parallelization is most common on SMP machines as one search tree representation is shared among several compute cores. Each core performs one simulation and updates the shared tree representation using atomic instructions. Leaf-Parallelization and Slow-Tree-Parallelization are suited for distributed memory machines. Leaf-Parallelization handles the in-tree part on only one CN, and computes multiple playouts on remote CNs once a leaf of the search tree representation $T$ is reached. Slow-Tree-Parallelization performs rather independent searches on all CNs but occasionally synchronizes statistics of near-root tree nodes.

Among these methods, Slow-Tree-Parallelization is currently excelling for distributed memory systems. One drawback of this method is that no effort is made at all to exploit the increased amount of memory available within a cluster. Instead, all CNs try to keep a nearly identical copy of the search tree representation. However, simply distributing the search tree across all cluster nodes would result in very costly remote read/write operations, slowing down the simulation dramatically. In the following section we present a novel approach that combines both possibilities stated above resulting in an efficient parallelization of MCTS for distributed memory systems.

## 3 The UCT-Treesplit Algorithm for Parallel MCTS

We concentrate on homogeneous HPC systems with a fast interconnect (e.g., 10GBit Ethernet or Infiniband) consisting of $N$ compute nodes (CNs), each having $C > 1$ compute cores that share the CN's entire memory, a model that fits most common HPC systems. We use MPI for message passing and devote one core (IO) of each CN for a thread handling message passing and work package distribution while the remaining $C - 1$ cores (workers) are bound to worker threads. Fig. 3 illustrates the setup of a CN.

The IO and worker cores communicate using thread-safe ring-buffers (Transfer Buffer In/Out) that reside in shared memory. An infinite loop running on the IO core reads available messages containing work packages from the network link and stores them in a buffer. Afterward, a work package scheduler distributes the received packages among the workers' ring-buffers, balancing the work load.




**(a)** Tree-Parallelization **(b)** Leaf-Parallelization

**(c)** Slow-Tree-Paralellization **(d)** UCT-Treesplit

**Fig. 2.** Overview of MCTS parallelization methods

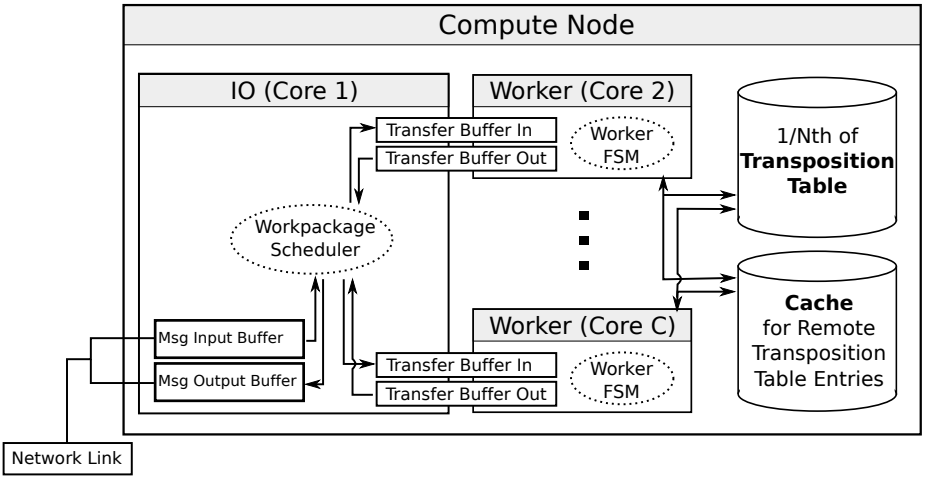



**Fig. 3.** Setup of a compute node

Workers start computation once they receive a package and, if required, send response messages back to the IO core using the corresponding buffer. In turn, the IO core frequently collects messages from the workers' ring-buffers and forwards them to the network link appropriately.

During their computation, workers require access to state-action statistics. As for some search domains it may be possible that equal states are reached through different paths, we use a lock-free but thread-safe transposition table[7] TR capable of storing size(TR) nodes of the search "tree" representation. We assume the existence of a hash-function hash : $S \rightarrow \mathbb{N}$ assigning a unique and equally

distributed hash value to any search tree node. A straight forward method for computing an index $I_{\mathrm{TR}}(s)$ into TR for a search tree node $s$ is given by:

$$I_{\mathrm{TR}}(s) = \mathrm{hash}(s) \bmod \mathrm{size}(\mathrm{TR})$$

Distributing TR on $N$ CNs of a cluster may be done by storing on each CN $i$ for $1 < i < N$ a transposition table $\mathrm{TR}_i$ of size $\mathrm{size}(\mathrm{TR})/N$. An index to this distributed table is a tuple of a CN $i$ and a local index $I_{\mathrm{TR}_i}$ to $\mathrm{TR}_i$. Both can be computed as follows:

$$i(s) = \mathrm{hash}(s) \bmod N \tag{1}$$

$$I_{TR_i}(s) = (\mathrm{hash}(s)/N) \bmod (\mathrm{size}(\mathrm{TR})/N) \tag{2}$$

Beside the partial transposition table, Fig. 3 also shows a cache for entries of remote transposition tables allowing for faster access to statistics of frequently visited states.

The key technique of our approach is the spreading of a single search tree among all CNs while overlapping communication with the computation of additional simulations. We break simulations into small 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.

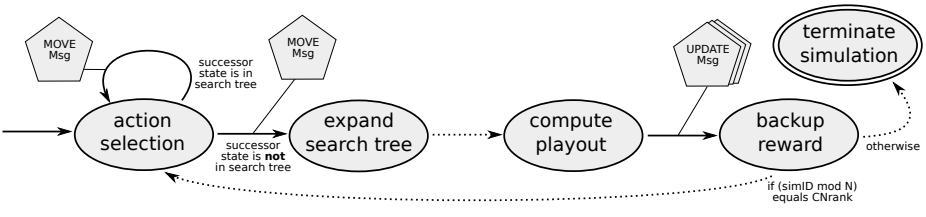



**Fig. 4.** Finite state machine for our distributed simulation (Worker FSM)

Fig. 4 illustrates our distributed simulation process as a finite state machine (FSM). 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. However, between two consecutive action selection steps a simulation may move to another CN through a MOVE message. The dotted arrows in Fig. 4 represent state transitions that always happen on a single CN while solid arrows may cross CN boundaries. Those arrows are annotated with the corresponding messages that are sent in case the CN changes. Note that the UPDATE message is likely sent to more than one CN. In fact, it is send to all CNs visited by the simulation as statistics need to be updated on all these CNs. The states of the FSM are the work packages that make up the computational load.

Each of $S_{\text{par}}$ simulations running in parallel suffers loss of information represented by the results of the $S_{\text{par}} - 1$ other simulations that would be available in a sequential UCT version. Obviously this impairs the search quality [16] and urges us to keep $S_{\text{par}}$ as small as possible. Furthermore, we duplicate and occasionally synchronize frequently visited tree nodes on all CNs to reduce the communication overhead. In total the algorithm requires us to determine the values of three parameters:

- $N_{\text{dup}}$: The number of simulations that must have passed a state before it gets duplicated on all CNs.
- $N_{\text{sync}}$: The number of simulations that lead to the synchronization of a shared state. Each time one action of a shared state $s$ has been visited at least $N_{\text{sync}}$ times after the last synchronization, all values of actions of $s$ are synchronized.
- $O$: An overload factor used to compute the number of simulations $S_{\text{par}} := (C-1)NO$ that run in parallel on a system with $N$ compute nodes.

Note that UCT-Treesplit may be configured to behave comparably to Slow-Tree-Parallelization by sharing all nodes immediately and choosing an appropriate threshold $N_{\text{sync}}$ for triggering synchronization. On the other extreme, UCT-Treesplit behaves like Tree-Parallelization if tree nodes are never shared.

The search process begins by sharing the root node among all CNs. Then, each CN starts $S_{\text{par}}/N$ simulations. A unique identifier simID $\in \{0, S_{\text{par}} - 1\}$ is assigned to each simulation. The data structure describing a simulation consists of:

- The **state stack** containing all states visited and actions taken during simulation. Together with each state, we store the CN where the action selection took place.
- The **simulation identifier** simID.

Procedure WorkerMainLoop gives a pseudo code representation of the main loop running on each worker core. Once a worker receives a MOVE message it searches for the current state's statistics in the cache and transposition table, respectively (line 6). If no entry exists, the worker **expands** the search tree by adding a new entry for the state. Immediately afterwards, a **playout** is computed and an UPDATE message is sent to all CNs visited in the course of the simulation (line 9). In case state statistics are found, an **action is selected** as in the sequential algorithm (line 7) leading to a new state $s'$. If the statistics to $s'$ are already shared or are located in the CN's transposition table, the computation continues on the same CN. Otherwise, a MOVE message is sent to the CN storing the statistics of $s'$. Upon receipt of an UPDATE message, statistics for all states for which an action was selected on this CN get **updated** (line 12). MPI assigns a rank number CNrank $\in \{0, N\}$ to each CN. If the CN's rank equals (simID mod $N$), the worker initiates a new simulation.

**Procedure** WorkerMainLoop

```
   //
 1 while Time available do
 2     if an incoming message is available then
 3         M_out ← ∅;
 4         msg ← inTransferQueue.dequeue();
 5         if msg is a move-message then
 6             if msg.stateStack.top() is already in transposition table then
 7                 M_out ← actionSelection(msg);
 8             else
 9                 M_out ← expandAndPlayout(msg);
10             end
11         else// an update-message arrived
12             M_out ← update(msg);
13         end
14         outTransferQueue.enqueue(M_out);
15     end
16 end
```

## 4 Experiments

In this section, we present the experimental setup and the results achieved with our Go engine Gomorra that incorporates the novel UCT-Treesplit algorithm.

### 4.1 Setup

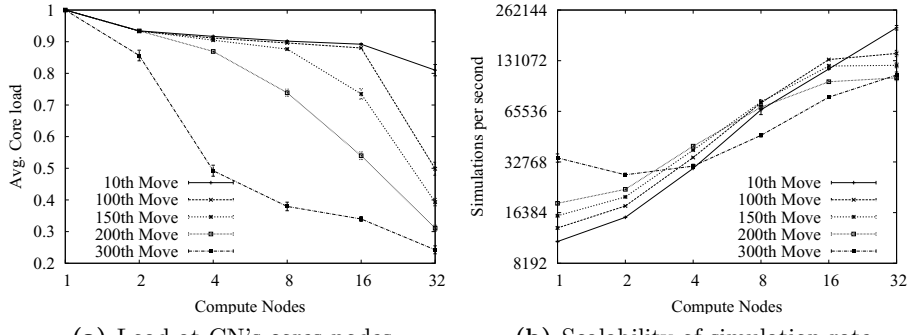
Our computer Go engine Gomorra implements several state of the art enhancements over basic UCT and proved its playing strength previously in several games against the currently strongest computer Go programs. In our experiments different instances of Gomorra play against each other on a 19x19 board size, giving each player 10 minutes to make all its moves in a game. We choose the following values for the three UCT-Treesplit parameters: $N_{\text{dup}} := 16$, $N_{\text{sync}} := 100$ and $O := 3$. 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 size. Although reducing $N_{\text{dup}}$ decreases the communication overhead for single simulations because less CN hops take place, the overhead for synchronizing shared nodes increases because more nodes are shared. However, few hops per simulation allow for keeping $O$ small. Furthermore, lower values of $N_{sync}$ lead to increased network traffic as more synchronization messages are sent.

For our experiments we use a cluster consisting of 60 CNs, each one equipped with 2 Intel Xeon X5650 CPUs (12 cores in total) running at 2.67 GHz and 36 GByte of main memory. The CNs are connected by a 4xSDR Infiniband network. We use OpenMPI for message passing between the CNs.

### 4.2 Results

Since UCT-Treesplit is communication intensive we first measure the average load on the worker cores. A low load indicates that work packages cannot be communicated fast enough and that the network is too slow in relation to the average work package size, number of cores and the core's clock rate. As MOVE and UPDATE messages are rather tiny in our experiments, we identify the network latency as the limiting parameter. With the given settings for $N_{\text{dup}}$, $N_{\text{sync}}$ and $O$, we have to restrict the number of used cores per CN to 6 to achieve reasonable loads for higher numbers of CNs. Fig. 5a shows the measured core loads for different numbers of CNs at different phases of the game. Considering different phases of a game is important since a 19x19 Go game lasts for about 250-350 moves, and playouts at the end of the game require less computation.

Next, we are interested in the achievable simulation rates, measured in simulations per second. The simulation rate directly influences the achievable playing strength. Fig. 5b displays the scalability of the simulation rate over the number of compute nodes. It can be seen that UCT-Treesplit scales well for up to 32 CNs in early game phases while after move 200 the performance decreases considerably with higher numbers of CNs.




**(a)** Load at CN's cores nodes **(b)** Scalability of simulation rate

**Fig. 5.** Scalability of simulations with increasing number of CNs at different game phases

The most important metrics, however, is the gain in playing strength achievable with increasing compute resources. Table 1a presents the achieved winning percentages of Gomorra playing against a copy of itself that can rely on a doubled amount of simulations computed per move decision. To conduct these experiments in reasonable time, we use a 4 core SMP machine except for the results marked with * for which we use a 12 core SMP machine and for the results marked with ** for which we use a 24 core SMP machine equipped with 132 GB of main memory to be able to store the vast amount of statistics. Comparable experiments were done in [2](Table 1) for the Go program MoGo. Although the absolute playing strength of MoGo is superior to Gomorra's, our Go engine seems to scale better with higher simulation numbers.

**Table 1.** Evaluation of Gomorra's playing strength

**(a)** Running on a single compute node.

| $N_s$: Number of sim/move | Winrate $2N_s$ vs $N_s$ | Games played |
|---|---|---|
| 1,000 | 85.8 ±1.6% | 500 |
| 4,000 | 86.0 ±1.6% | 500 |
| 16,000 | 79.6 ±1.8% | 500 |
| $128,400^{*}$ | 82.4 ±1.7% | 500 |
| $256,800^{**}$ | 83.9 ±3.0% | 149 |

**(b)** Using parallel UCT-Treesplit.

| N: Number of CNs | Winrate 2N vs N | Games played |
|---|---|---|
| $1^{*}$ | 53.3 ±2.0% | 600 |
| 2 | 73.0 ±1.8% | 600 |
| 4 | 61.3 ±2.0% | 600 |
| 8 | 53.8 ±2.0% | 600 |
| 16 | 46.1 ±2.3% | 486 |

Table 1b shows the achieved winning rates of Gomorra playing against a copy of itself using the UCT-Treesplit algorithm on a varying number of compute nodes. In the 2 vs 1 node experiment marked with $^{*}$, the single node version has no additional work for building MPI messages, moving simulations between compute cores, etc., explaining the rather small advantage of the double node version.

## 5 Conclusion and Future Work

In this paper, we investigate a novel approach to parallelize MCTS on distributed memory HPC systems. We present a way to share a single game tree representation efficiently among all compute nodes and evaluate the behavior of the new UCT-Treesplit algorithm in a high-end Go engine. We show that, for the game of Go, UCT-Treesplit scales up to 16 nodes.

An explanation for the diminishing gains UCT-Treesplit achieves with increasing compute nodes as shown in Table 1b could be the high number of simulations that are computed in parallel. With an overload factor of 3, we compute 480 simulations in parallel on 32 CNs. Richard B. Segal measured the scaling of the Go program Fuego with increasing numbers of parallel simulations and different time settings in [16]. His experiments showed a major decrease in playing strength if more than 128 threads are used for short time settings. However, giving Fuego more time per move yielded good results for even 512 parallel simulations. Thus, we may expect better scalability of UCT-Treesplit by just increasing the search time.

Another important observation and possible explanation for the scalability limitations is the diminishing increase in simulation rate at the end of games, i.e., when the remaining search tree and thus the work packages become too small, as shown in Fig. 5b. As part of future work we will try to keep the simulation rate high and reduce the communication overhead occurring during each simulation, for example by smoothly decreasing $N_{\text{dup}}$ and increasing $N_{\text{sync}}$ when it comes to the end of a game. Furthermore, we will study the influence and optimal settings for the various parameters involved.

## References


1. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-Time Analysis of the Multiarmed Bandit Problem. In: Machine Learning, vol. 47, pp. 235–256. Kluwer Academic, Dordrecht (2002)
2. Bourki, A., Chaslot, G.M.J.-B., Coulm, M., Danjean, V., Doghmen, H., Hoock, J.-B., Hérault, T., Rimmel, A., Teytaud, F., Teytaud, O., Vayssiére, P., Yu, Z.: Scalability and Parallelization of Monte-Carlo Tree Search. In: International Conference on Computers and Games, pp. 48–58 (2010)
3. Chaslot, G.M.J.-B., Winands, M.H.M., Jaap van den Herik, H.: Parallel Monte-Carlo Tree Search. In: Conference on Computers and Games, pp. 60–71 (2008)
4. Coulom, R.: Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M(J.) (eds.) CG 2006. LNCS, vol. 4630, pp. 72–83. Springer, Heidelberg (2007)
5. Coulom, R.: Computing Elo Ratings of Move Patterns in the Game of Go. ICGA Journal 30(4), 198–208 (2007)
6. Donninger, C., Kure, A., Lorenz, U.: Parallel Brutus: The First Distributed, FPGA Accelerated Chess Program. In: 18th International Parallel and Distributed Processing Symposium. IEEE Computer Society, Los Alamitos (2004)
7. Enzenberger, M., Müller, M.: A Lock-Free Multithreaded Monte-Carlo Tree Search Algorithm. In: van den Herik, H.J., Spronck, P. (eds.) ACG 2009. LNCS, vol. 6048, pp. 14–20. Springer, Heidelberg (2010)
8. Feldmann, R., Mysliwietz, P., Monien, B.: Distributed game tree search on a massively parallel system. In: Monien, B., Ottmann, T. (eds.) Data Structures and Efficient Algorithms. LNCS, vol. 594, pp. 270–288. Springer, Heidelberg (1992)
9. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modifications of UCT with Patterns in Monte-Carlo Go. Technical Report 6062, INRIA (2006)
10. Himstedt, K., Lorenz, U., Möller, D.P.F.: A twofold distributed game-tree search approach using interconnected clusters. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 587–598. Springer, Heidelberg (2008)
11. Huang, S.-C., Coulom, R., Lin, S.-S.: Monte-Carlo Simulation Balancing in Practice. In: Conference on Computers and Games, pp. 81–92 (2010)
12. Donald Knuth, E., Moore, R.W.: An Analysis of Alpha-Beta Pruning. In: Artificial Intelligence, vol. 6, pp. 293–327. North-Holland Publishing Company, Amsterdam (1975)
13. Kocsis, L., Szepesvári, C.: Bandit Based Monte-Carlo Planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
14. Lorenz, U.: Parallel controlled conspiracy number search. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 420–430. Springer, Heidelberg (2002)
15. Romein, J.W., Plaat, A., Bal, H.E., Schaeffer, J.: Transposition table driven work scheduling in distributed search. In: National Conference on Artificial Intelligence, pp. 725–731 (1999)
16. Segal, R.B.: On the Scalability of Parallel UCT. In: International Conference on Computer and Games, pp. 36–47 (2010)
17. Silver, D.: Reinforcement Learning and Simulation-Based Search in Computer Go. PhD thesis, University of Alberta (2009)
18. Silver, D., Tesauro, G.: Monte-Carlo Simulation Balancing. In: International Conference on Machine Learning, pp. 945–952 (2009)