UCT-Treesplit - Parallel MCTS on Distributed Memory

Lars Schaefers and **Marco Platzner**
University of Paderborn
{slars,platzner}@uni-paderborn.de

Ulf Lorenz
TU Darmstadt
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. 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.

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 (Knuth and Moore 1975). In this paper, we present a novel approach for the parallelization of MCTS for distributed high-performance computing (HPC) systems. We evaluate the performance of our parallelization technique on a real-world application, our high-end Go engine Gomorra. Gomorra has proven its strength in several computer Go tournaments last year, including the Computer Olympiad 2010 in Kanazawa, Japan.

MCTS: Background and Related Work

MCTS may be classified as a sequential best-first search algorithm (Silver 2009), 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 all computation entities makes parallelization of MCTS for distributed memory environments a highly challenging task. The so-called UCT algorithm (short for *Upper Confidence Bounds applied to trees* (Kocsis and Szepesvári 2006)) is a modern variant of MCTS and yields the experimentally best results for most of the current Go programs. For a detailed description of UCT we refer to (Kocsis and Szepesvári 2006). For an overview of other parallelization approaches for UCT see e.g. (Chaslot, Winands, and van den Herik 2008).

Copyright © 2011, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

UCT-Treesplit Algorithm for parallel MCTS

The key technique of our approach is the spreading of a single search tree among all compute nodes (CNs) and guidance of simulations across CN boundaries using message passing. Computing more simulations in parallel than cores are available allows us to overlap communication times with additional simulations. 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 (Lorenz 2002)(Feldmann, Mysliwietz, and Monien 1992). A comparable approach used with traditional $\alpha\beta$ search was termed transposition driven scheduling (TDS) (Kishimoto and Schaeffer 2002). As a result we would obtain a single distributed representation of the partial value function learned by UCT. However, to reduce the communication overhead, we store the statistics of frequently visited search tree nodes on all CNs and synchronize regularly.

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. Fig. 1 illustrates our distributed simulation process as a finite state machine (FSM). During the in-tree part of a simulation (i.e. the part where the partial value function learned so far influences the simulation policy) 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. 1 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

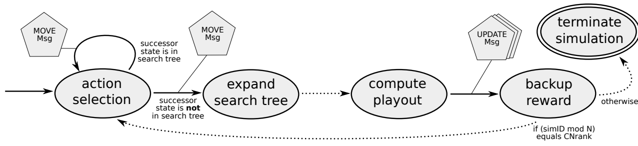

Figure 1: Finite state machine for our distributed simulation (Worker FSM)

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.

In total the algorithm requires us to determine the values of three parameters:

- N_{dup}: The number of simulations that must have passed a state before it gets duplicated on all CNs.
- N_{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_{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 where each node has C cores.

Experiments

In our experiments different instances of our Go Engine 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 $O := 3$. For our experiments we use a cluster consisting of 60 CNs, each one equipped with 6 cores 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.

As the playing strength depends on the number of simulations computable in given time, we are interested in the achievable simulation rates, measured in simulations per second. Fig. 2 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.

The most important metric, however, is the gain in playing strength achievable with increasing compute resources. Table 1a presents the achieved winning percentages of the single node version of Gomorra playing against a copy of itself that can rely on a doubled amount of simulations computed per move decision. 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.

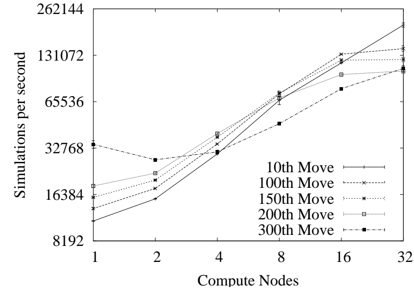

Figure 2: Scalability of simulations with increasing number of CNs at different game phases

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

(a) Running on a single compute node.

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

(b) Using parallel UCT-Treesplit.

Table 1: Evaluation of Gomorra's playing strength

References

Chaslot, G. M.-B.; Winands, M. H.; and van den Herik, H. J. 2008. Parallel Monte-Carlo Tree Search. In *Conference on Computers and Games*, 60–71.

Feldmann, R.; Mysliwietz, P.; and Monien, B. 1992. Distributed game tree search on a massively parallel system. In *Data Structures and Efficient Algorithms*, volume 594 of *LNCS*, 270–288. Springer-Verlag.

Kishimoto, A., and Schaeffer, J. 2002. Transposition Table Driven Work Scheduling in Distributed Game-Tree Search. In *Canadian Conference on Artificial Intelligence*, volume 2338 of *LNCS*, 56–68. Springer Berlin/Heidelberg.

Knuth, E. D., and Moore, R. W. 1975. An Analysis of Alpha-Beta Pruning. In *Artificial Intelligence*, volume 6, 293–327. North-Holland Publishing Company.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo Planning. In *ECML*, volume 4212 of *LNCS/LNAI*, 282–293.

Lorenz, U. 2002. Parallel Controlled Conspiracy Number Search. In *Euro-Par*, 420–430.

Silver, D. 2009. *Reinforcement Learning and Simulation-Based Search in Computer Go*. Ph.D. Dissertation, University of Alberta.