Massively Parallel Monte Carlo Tree Search

Kamil Rocki, Reiji Suda

Department of Computer Science
Graduate School of Information Science and Technology
The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan
{kamil.rocki, reiji}@is.s.u-tokyo.ac.jp

Abstract. Monte Carlo Tree Search is a method of finding near-optimal solutions for large state-space problems. Currently, is it very important to develop algorithms being able to take advantage of great number of processors in such areas. In this paper MCTS parallel implementations for thousand of cores are presented and discussed. The MCTS parallelization method used is the root parallelization. Implementation of the distributed scheme uses MPI. Results presented are based on the Reversi game rules.

1 Introduction

In artificial intelligence game trees have always been important as a structure used for representing states and possible outcomes for each of them, based on the decision (move) made. In order to choose the best possible move being in a particular state, a look-ahead search must be performed. For small trees a thorough search is feasible given algorithms such as the minimax[1] search or just brute force search. However for current complex problems, games or puzzles such approaches fail for several reasons. First is that the time needed for solving such problems might be very long, i.e. the state-space size of game Go is moreover it equals approximately $3.7 * 10^{79}$ for the 13x13 board size. Even more important is that the time needed is unknown until the solution is found, which is a huge problem when time constraints are present. Then, effective parallelization of algorithms like minimax with $\alpha\beta$ pruning, which is strongly sequential, is extremely difficult[2][3].

Therefore methods such as Monte Carlo Tree Search[4] were introduced to search the tree in a semi-random way and providing sub-optimal results. Instead of performing a complete tree search, a number of simulations from a given state is executed to estimate the successors' values. The key advantage of this approach is that the longer the algorithm runs the better the solution and the time limit can be specified allowing to control the quality of the decisions made. This means that this algorithm guarantees a solution (although not necessarily the best one). It provides relatively good results in games like Go or Chess where standard algorithms are likely to fail. MCTS algorithm was introduced recently and because of that, currently some parallelization methods have been proposed. The result

obtained are promising in the sense of utilizing multiple CPUs much more effectively than in the case of standard tree search algorithms. In this paper a parallel implementation of MCTS is presented and discussed among with the results of running it on the TSUBAME supercomputer in Tokyo.

2 Monte Carlo Tree Search overview

A simulation is defined as a series of random moves until the end of a game (until the move is still possible). The result of such a simulation can be successful, when there was a win in the end or unsuccessful otherwise. So, let every node i in the tree store the number of simulations T_i (visits) and the number of successful simulations S_i. First the algorithm starts only with the root node. The general MCTS algorithm comprises 4 steps (Fig. 1) which are repeated until a particular condition is met (i.e. there is no time left):

Selection - a node from the game tree is chosen based on the specified criteria. The value of each node is calculated and the best one is selected. In this paper, the formula used to calculate the node value is the Upper Confidence Bound (UCB). Supposed that some simulations have been performed for a node, first the average node value is taken and then the second term which includes the total number of simulations for that node and its parent. The first one provides the best possible node in the analyzed tree (exploitation), while the second one is responsible for the tree exploration. That means that a node which has been rarely visited is more likely to be chosen, because the value of the second terms is greater. The C parameter adjusts the exploitation/exploration ratio.

$$UCB_i = \frac{S_i}{t_i} + C * \sqrt{\frac{logT_i}{logt_i}}$$

Where:

T_i - total number of simulations for the parent of node i
C - a parameter to be adjusted

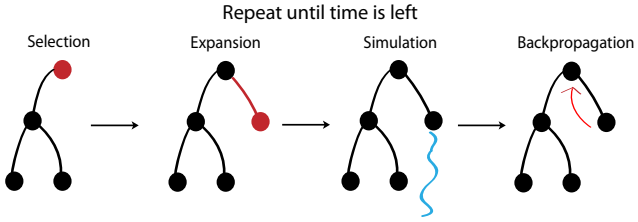

Fig. 1. MCTS algorithm loop

Expansion - one or more successors of the selected node are added to the tree depending on the strategy.

Simulation - for the added node(s) perform simulation(s) and update the node(s) values (successes, total)

Backpropagation - update the parents' values up to the root nodes. The numbers are added, so that the root node has the total number of simulations and successes for all of the nodes and each node contains the sum of values of all of its successors.

2.1 Parallelization

This paper focuses on large-scale MCTS algorithm parallelization. The two main approaches are considered - **leaf parallelization** and **root parallelization** [6]. Other more complicated methods including synchronization are omitted. In the first case, in the *simulation* step of the algorithm, n simulations of a given node are performed in parallel and afterwards the results are added. The latter approach starts with n root nodes (for n threads) and builds n separate MCTS trees in parallel. In the end, the results for the root node and its successors are summed. [6] presents the root parallelization method as the better one and therefore this papers focuses only on this method.

3 Multi-node implementation

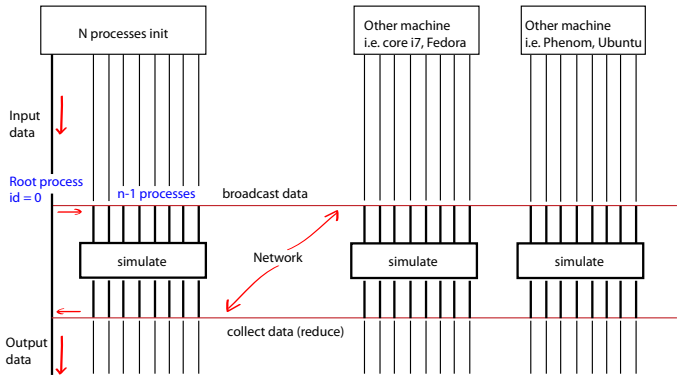

Fig. 2. MPI integration

In order to run the simulations on more machines the application has been modified in the way that communication through MPI is possible. This allows to take advantage of systems such as the TSUBAME supercomputer or smaller heterogenous clusters. The implemented scheme (Fig. 4) defines one master process (with id 0) which controls the game and I/O operations, whereas other processes are active only during the MCTS phases. The master process broadcast the input data (current state/node) to the other processes. Then each process performs an independent Monte Carlo search and stores the result. After this phase the master process collects the data (through the *reduce* MPI operation which sums the results. To ensure that all the processes end in the specified amount of time, the start time and time limit values are included in the input data. None of the processes can exceed the specified time limit. The other important aspect of the multi-node implementation is whether it should be based entirely on MPI even within one node. The possible approaches include MPI, MPI/OpenMP, MPI/pthreads schemes.

4 Results

The results presented are based on the average values of 1000 simulations in each case. The test platform is the TSUBAME supercomputer using up to 64 nodes, where one node comprises 8 2.4 GHz dual-core Opteron CPUs (SunFire X4600) and 32GB of main memory. Each node is connected by the Voltaire ISR9288 Infiniband network. Fig. 3 presents the average number of point difference between a player using the parallel implementation of the MCTS algorithm and a player which makes random moves in respect to the number of cores used and time dedicated for each move (20 ms, 50 ms or 500 ms). Both the number of processors used as the time matter in this case and influence the score. However the parallelism seems to be the more important factor what can be observed later in Fig. 7 which show the ratio $\frac{cores*milliseconds}{points}$. It is clear that the longer the time is spent on the search the worse the performance is (points per cpu ms). Fig. 4 shows the average number of simulations per second for aforementioned runs, here also the best performance is achieved when the time is short. This can be explained by the fact that, when the time is longer the trees become larger and then more time is spent on looking for the best node to expand in such a tree. Worth mentioning is the fact that the speedup is almost linear (around 10k simulations per second for the 20 ms run on 1 core and more than 10M simulations/second for 1024 cores. Fig. 5 is very similar to Fig. 3, except for the opponent is not a randomly playing computer, but a sequential version of the MCTS algorithm, the time used for a move in each case is 100 milliseconds. A win ratio of 99 percent is achieved using only 64 cores in this case. To test how the parallel algorithm performs when 'better' opponents are considered it has been also tested against itself, the 16-core parallel version, the results can be seen in Fig. 6.

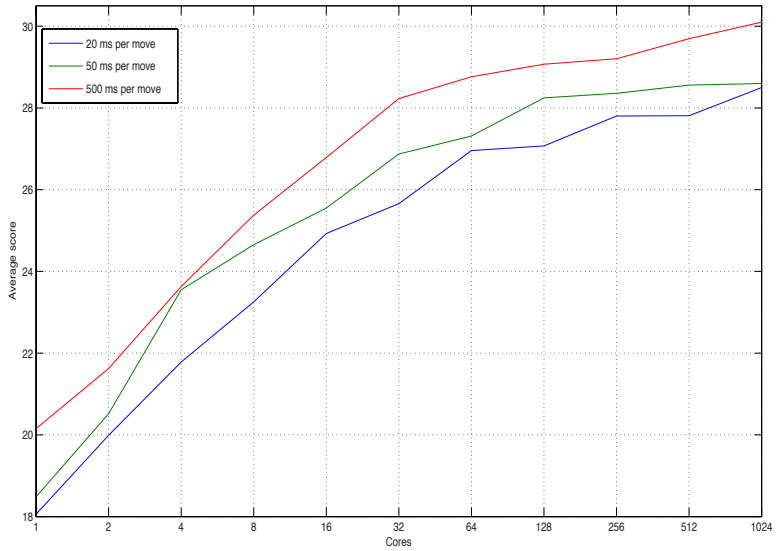

Fig. 3. TSUBAME: Average score of MCTS parallel algorithm playing against randomly acting computer depending on the time spent on the sequential search and number of cpu cores

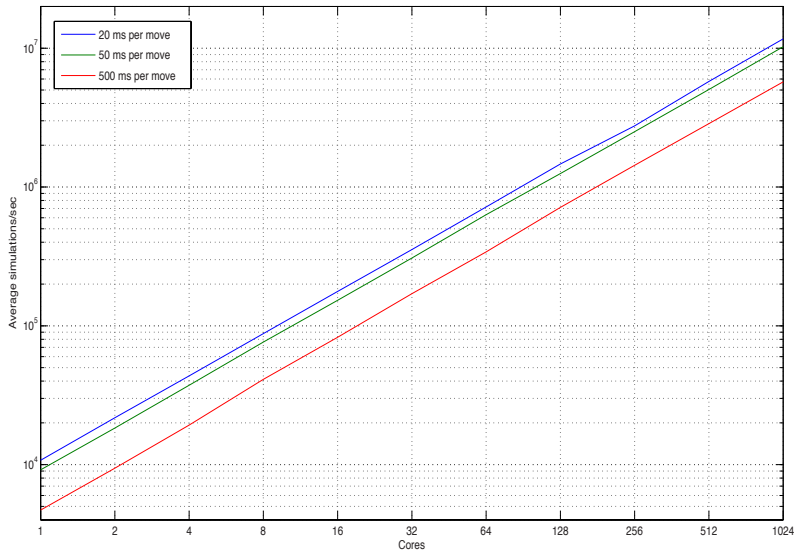

Fig. 4. TSUBAME: Average number of simulations per second in total depending on the time spent on the sequential search and number of cpu cores

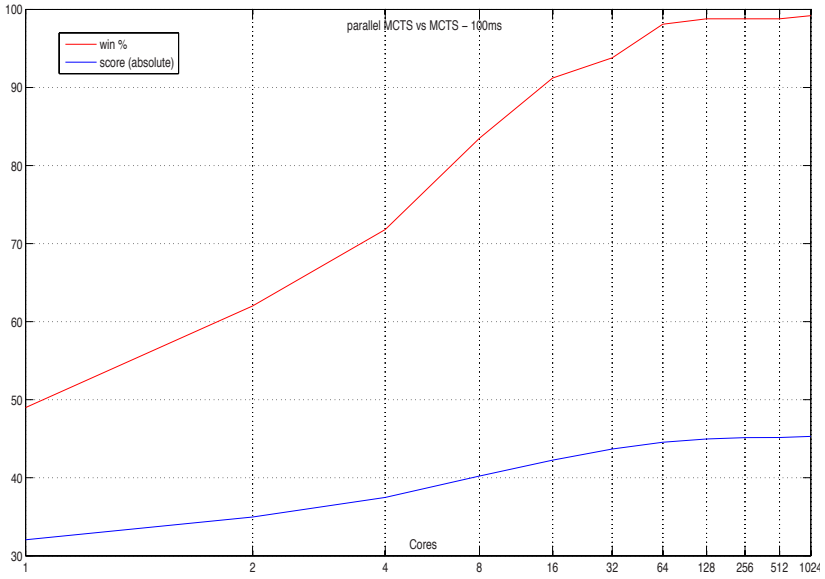

Fig. 5. TSUBAME: Average score and win ratio of MCTS parallel algorithm playing against sequential MCTS agent depending on the number of cpu cores

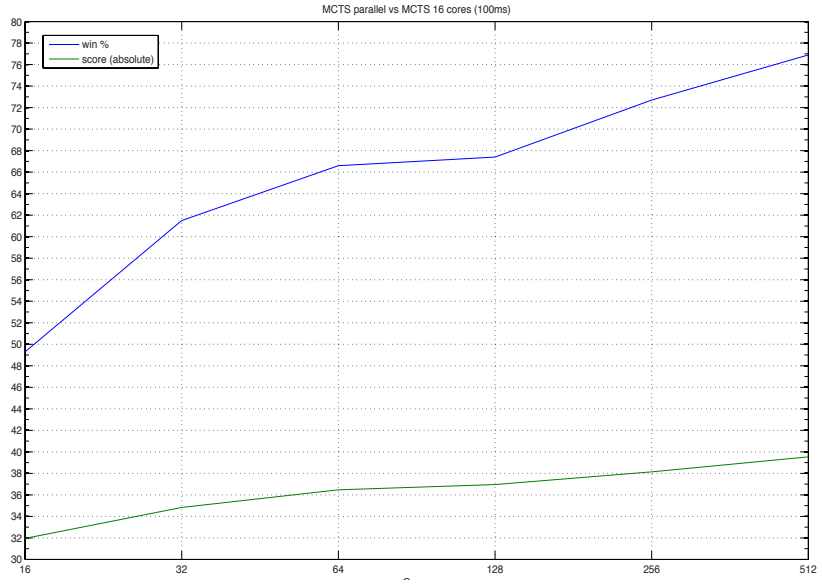

Fig. 6. TSUBAME: Average score and win ratio of MCTS parallel algorithm playing against 16-core MCTS parallel agent depending on the number of cpu cores

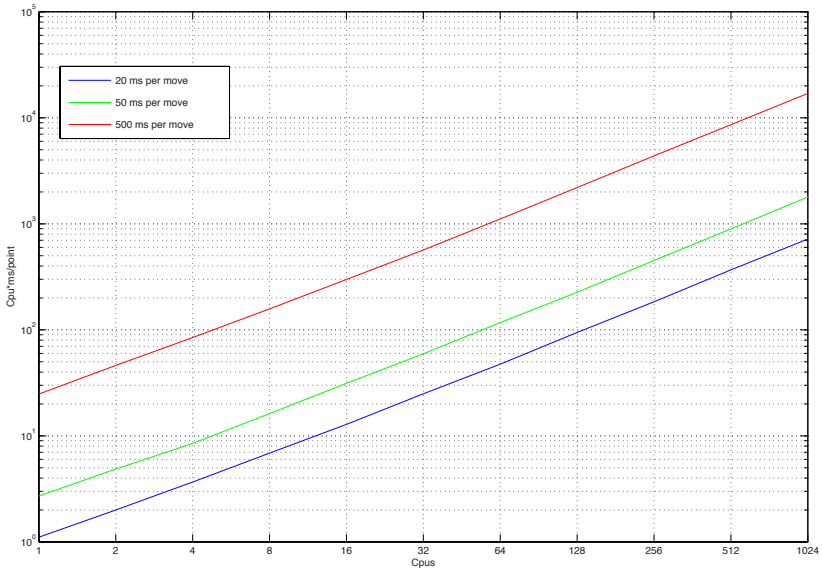

Fig. 7. TSUBAME: Average cost of a single point (CPU*ms/point) depending on the time spent on the sequential search and number of cpu cores

5 Conclusions

The first and main conclusion of the results obtained is that the root parallelization method is very scalable in terms of multi-CPU performance. The explanation may be that it is less likely for more threads to be stuck in the local optimas, which leads to the more thorough search. Moreover, the MPI communication overhead is very low and in case of a system such as TSUBAME, other mixed implementations (MPI/pthreads) provide worse performance. That is very promising if the great number of processors is considered. Additional conclusion that may be drawn from the results is that according to the time per cost chart it is obvious that dedicating more CPU cores instead of sequential MCTS search time significant power/parallel performance can be observed. This means that using more cpu-cores at the same time in terms of CPU-time utilization is better than longer search time using sequential MCTS algorithm. The main cause of that is that searching multiple trees at the same time provides better results due to no local minimum stall limitaiton.

6 Future work

- GPU implementation - (Fig. 8)
- Applying the parallel MCTS to Go
- Running the application on the new TSUBAME 2.0 system and tuning it to achieve the best possible performance

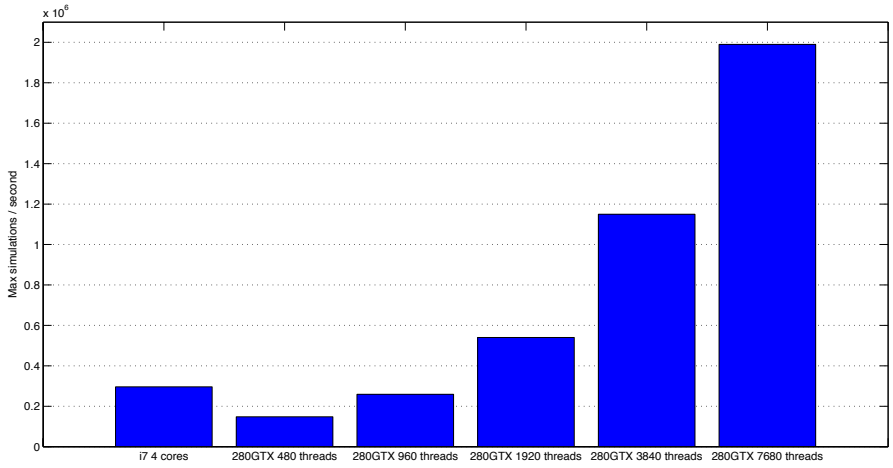

Fig. 8. GPU advantage - Simulations per second (average for multiple games) - 2.66 GHz core i7 vs 280 GTX GPU

– Comparing the performance of presented to other existing algorithms and MCTS implementations

References

[1] Knuth, D. E., Moore, R. W.: An analysis of alpha-beta pruning. Artificial Intelligence, 6 (1975): 293–326

[2] Manohararajah V.: Parallel Alpha-Beta Search on Shared Memory Multiprocessors, Master Thesis, Graduate Department of Electrical and Computer Engineering, University of Toronto, 2001

[3] Schaeffer, J., Brockington M. G.: The APHID Parallel $\alpha\beta$ algorithm, Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing, 1996, p. 428

[4] Coulom R.: Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search, 5th International Conference on Computer and Games, 2006

[5] Kocsis L., Szepesvari C.: Bandit based Monte-Carlo Planning, 15th European. Conference on Machine Learning Proceedings, 2006

[6] Guillaume M.J-B. Chaslot, Mark H.M. Winands, and H. Jaap van den Herik: Parallel Monte-Carlo Tree Search, Computers and Games: 6th International Conference, 2008