Parallel Monte Carlo Tree Search Scalability Discussion

Kamil Rocki and Reiji Suda

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

Abstract. In this paper we are discussing which factors affect the scalability of the parallel Monte Carlo Tree Search algorithm. We have run the algorithm on CPUs and GPUs in Reversi game and SameGame puzzle on the TSUBAME supercomputer. We are showing that the most likely cause of the scaling bottleneck is the problem size. Therefore we are showing that the MCTS is a weak-scaling algorithm. We are not focusing on the relative scaling when compared to a single-threaded MCTS, but rather on the absolute scaling of the parallel MCTS algorithm.

Keywords: Monte Carlo, Scalability, Reversi, SameGame.

1 Introduction

Monte Carlo Tree Search (MCTS)[1][2][3] is a method for making optimal decisions in artificial intelligence (AI) problems, typically move planning in combinatorial games. It combines the generality of random simulation with the precision of tree search. In this paper we are focusing on the parallel MCTS usage and its scaling limitations. First we will very briefly explain how the MCTS algorithm works.

1.1 MCTS Algorithm Overview

A simulation is defined as a series of random moves which are performed until the end of a game is reached (until neither of the players can move). The result of this 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 (Figure 1) which are repeated until a particular condition is met (i.e. no possible move or time limit is reached).

1.2 MCTS Iteration Steps

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,

D. Wang and M. Reynolds (Eds.): AI 2011, LNAI 7106, pp. 452–461, 2011.

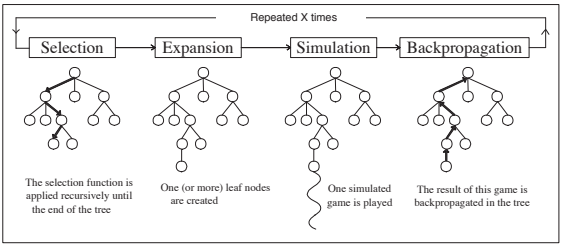

Fig. 1. A single MCTS algorithm iteration's steps (from [1])

the formula used to calculate the node value is the Upper Confidence bound applied to Trees (UCT)[2].

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

Where:
T_i - total number of simulations for the parent of node i
C - a parameter to be adjusted (low - exploitation, high - exploration).

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.

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

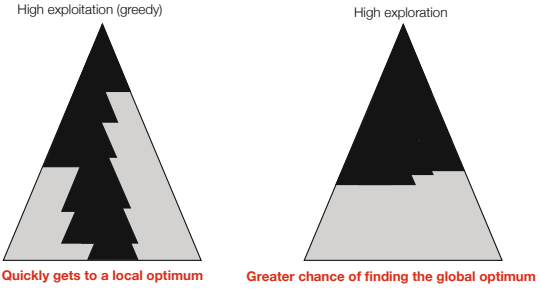

Fig. 2. Explotation and Exploration illustrated)

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.

2 Parallelization of Monte-Carlo Tree Search

2.1 Some of the Existing Parallel Approaches

In 2007 Cazenave and Jouandeau[4] propose 2 methods of parallelization of MCTS and later in 2008 Chaslot et al.[3] propose another one an analyze 3 approaches(Figure 3):

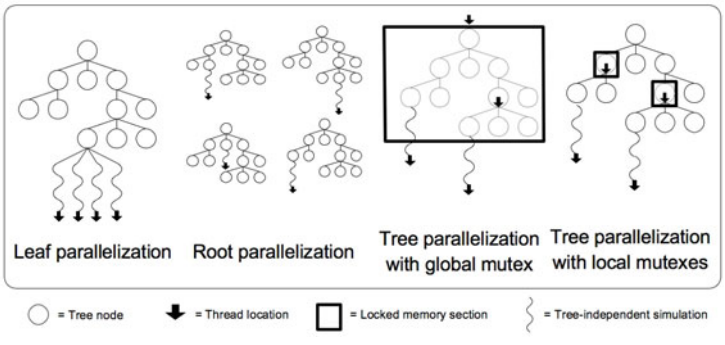

Fig. 3. Parallel MCTS as in [3])

1. **Leaf Parallelization.** It is one of the easiest ways to parallelize MCTS. Only one thread traverses the tree and adds one of more nodes to the tree when a leaf node is reached (*Selection* and *Expansion* phases). Next, starting from the leaf node, independent simulated games are played for each available thread (*Simulation* phase). When all games are finished, the result of all these simulated games is propagated backwards through the tree by one single thread (*Backpropagation* phase).
2. **Root Parallelization.** Cazenave[4] proposed a second parallelization called *single-run* parallelization. It is also referred to as *root parallelization*[3]. It consists of building multiple MCTS trees in parallel, with one thread per tree. The threads do not share information with each other. When the available time is spent, all the root children of the separate MCTS trees are merged with their corresponding clones. For each group of clones, the scores of all games played are added. The best move is selected based on this grand total. This parallelization method only requires a minimal amount of communication between the threads.It is more efficient than simple leaf parallelization[3], because building more trees diminishes the effect of being stuck in a local extremum and increases the chances of finding the true global maximum.

The goal of this paper is to determine which parameters affect the scalability of the algorithm when running on multiple CPUs/GPUs using root parallelization.

3 Methodology

3.1 MPI Usage

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 (Figures 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.

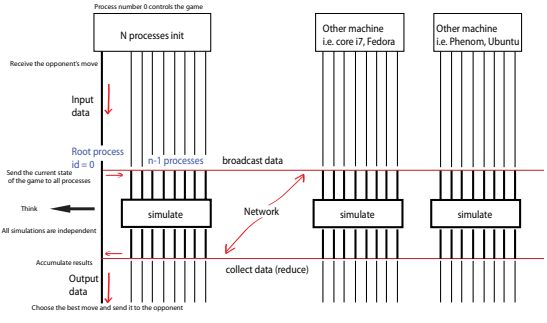

Fig. 4. MPI Processing scheme

3.2 Reversi

In general there are some basic features of graphs presenting the results which need to be explained, it is important to understand how to read them as they may seem to be complicated.

1. **Number of simulation per second** in regard to the number of CPU threads, GPU threads or other factors - (i.e. Figure 5). By this I mean the average number of random playouts performed (MCTS algorithm - step 3) during the game.
2. **Score** (or average score) in regard to the number of CPU threads or GPU threads. Score means the point difference between player A and player B who play against each other. The higher score, the better. If score is greater than 0, it means a winning situation, 0 means a draw, otherwise a loss.

3. **A game step** is a particular game stage starting from the initial Reversi position until the moment when no move is possible. There can be up to 60 game steps in Reversi. As the game progresses, the average complexity of the search tree changes, the way the algorithm behaves changes as well. Instead of showing the score or the speed of the algorithm in regard to the number of cores, I also show the performance considering the game stage.
4. **Win ratio** - Another type of measuring the strength of an algorithm. It means the proportion of the games won to the total number of games played. The higher, the better.

3.3 Samegame

A modified version of the MCTS algorithm is used[7] to be able to solve SameGame puzzle in order to get as many points as possible. The first reason to do this was to test if MCTS algorithm can be easily applied to other domains. And the second one to check the scalability limitations, since it is easier to adjust fewer parameters in SameGame rather than having a 2-player game and to see if there are any similarities in 2 problems. Before applying MCTS to SameGame it was not sure if the problem itself may cause limitations in scalability.

4 Results and Analysis

4.1 CPU MPI Implementation

Figure 5 shows the average number of simulations per second in total depending on the number of cpu cores. The very little overhead is observed during the MPI communication. The number of simulations per second increases almost linearly and for 1024 threads the speedup is around 1020-fold (around 8 million simulations/sec).

Figure 4.1B shows the average score and win ratio of MCTS parallel algorithm playing against sequential MCTS agent depending on the number of cpu cores. From this graph it can be seen that obviously when 1 root-parallel MCTS thread plays against the sequential MCTS, they are equal (the winning percentage of around 48% for each of them - the missing 4% are the draws). In this graph I present the absolute score (not the score difference between players, so it is not so tightly associated with the winning ratio, but still the more, the stronger the algorithm). When the number of cores doubles, the parallel algorithm wins in more than 60% of the cases, and when the number of threads equals 16, it reaches 90%, to get to the level of around 98% for 64 threads. Here, a high increase in the algorithm's strength is observed when the thread number is increased up to 16, later the strength increase is not as significant, which can lead to a conclusion that the root parallel MCTS algorithm can have some limitations regarding the strength scalability given the constant increase in the speed shown in Figure 5A. Actually this has been studied ([4][3][6]) and some conclusions have been formed that such a limit exists and that the root-parallel algorithm performs well only up to several cores.

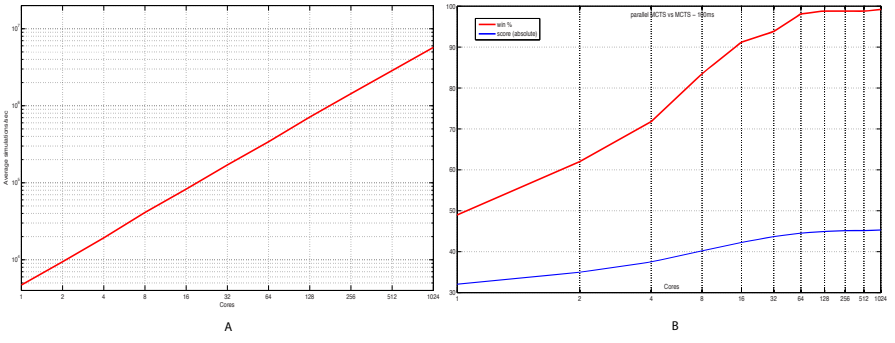

Fig. 5. (A) Average number of simulations per second in total depending on the number of cpu cores (multi-node), (B) - Average score and win ratio of MCTS parallel algorithm playing against sequential MCTS agent depending on the number of cpu cores

The first and main conclusion of the results obtained is that the root parallelization method is very scalable in terms of multi-CPU communication and number of simulations performed in given time increases significantly. Another one is that there is a point when raising the number of trees in the root parallel MCTS does not give a significant strength improvement.

4.2 Multi GPU Implementation

Multi-GPU implementation follows the same pattern as the multi-CPU scheme. Each TSUBAME node has 3 GPUs and using more than 3 GPUs requires MPI utilization. In Figure 6 it can be seen that just like with the multiple CPUs, there is no inter-node communication bottleneck and the raw simulation speed

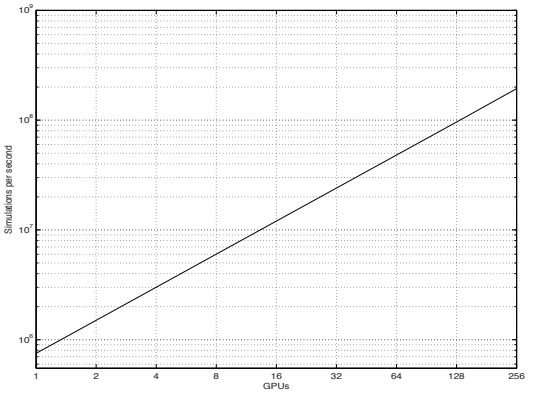

Fig. 6. Number of simulations per second depending on the GPU number, 56x112 threads per GPU

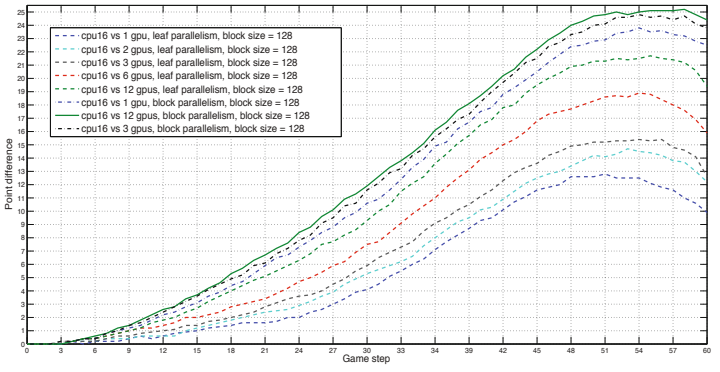

Fig. 7. 16 CPU cores vs different GPU configurations at particular game stages

increases linearly reaching approximately $2 * 10^8$ simulations/second when 256 GPUs (128 nodes, 2 GPUs per node) are used. Another graph (Figure 7 shows that just like in the case of multiple CPUs, even when the simulation speed increases linearly for multiple devices, the strength of the algorithm does not.

4.3 Scalability Analysis

The next illustrations (Figure 9) shows results of changing the MCTS environment to analyze the scalability affecting parameters. First in figures 9ABC we see how changing the sigma(C - exploitation/exploration ratio) affects it and Figure 9 presents how changing the problem size impacts the strength increase of the parallel approach. There are 2 things to be observed which are important. First, the sigma parameter affects the scalability in the way, that when the exploitation of a single tree is promoted, the overall strength of the algorithm improves better with the thread increase. In my opinion, this is due to the deeper tree search for each of the separate trees. When the exploration ratio is higher, then

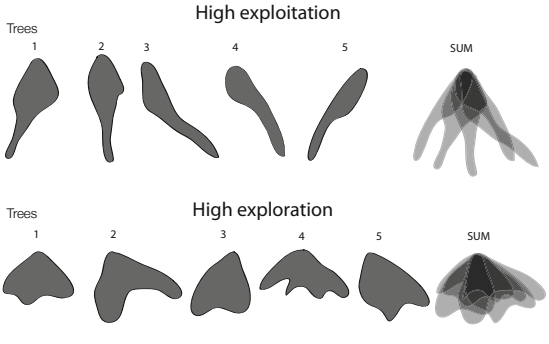

Fig. 8. Explanation of performance change caused by sigma constant adjustment

the trees basically form similar trees and reducing it propagates more diversified tree structures among the threads. Then in the next figure we see that as the problem size decreases, the improvement while parallelizing the algorithm also diminishes. It means that parallel MCTS algorithm presents *weak parallelism*, so it would also mean as long as we increase the problem size, the number of threads can grow and the results improve.

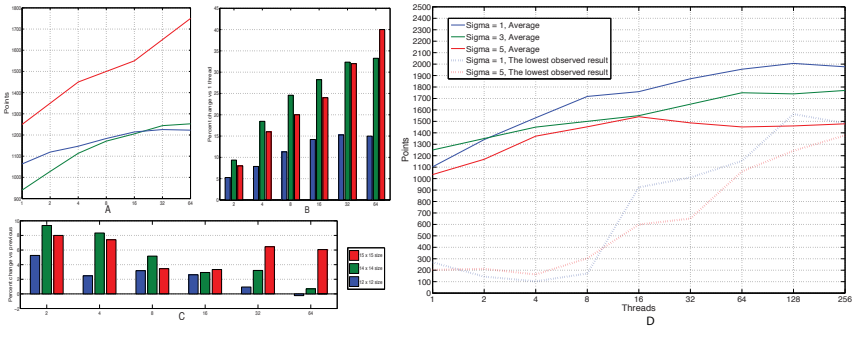

Fig. 9. Scalability of the algorithm when the problem size changes, (A) - absolute score, (B) - relative change in regard to 1 thread score, (C) - relative change when number of threads is doubled, sigma = 3, Root parallelism, (D) - Differences in scores during increasing number of threads and changing sigma constant, Root parallelismMax nodes = 10k

4.4 Random Sampling with Replacement

What is interesting in this case, when large number of threads is simulating at random is to obtain the probability of having exactly x distinct simulations after n simulations. Then, assuming that m is the total number of possible combinations (without order), $D(m, n)$ can be calculated, which is the expected number of distinct samples.

$$P(m, n, x) = \frac{\binom{m}{n}\binom{n-1}{n-x}}{\binom{m+n-1}{n}}$$

$P(m, n, x)$ is the probability of having exactly x distinct simulations after n simulations, where m is the total number of possible combinations (according the theorems of combinatorics, sampling with replacement). Then:

$$D(m, n) = \sum_{x=1}^{n} x * P(m, n, x)$$

It is hard to calculate D(m,n) for big numbers because of the factorials, but according to [5] an approximation can be used. Let $r = \frac{n}{m}$. Then:

$$D(m,n) \sim m(1-e^{-r}) + \frac{r}{2}e^{-r} - \frac{O(r(1+r)e^{-r}}{n}$$

The first term is the most important. Having this I was able to analyze the impact of having large number of samples in regard to the state space size and check how many of those samples would repeat (theoretically). For an instance if:

$m = 10^8$, $n_1 = 10^4$, $n_2 = 10^7$

In the first case (n_1): $\frac{D(m,n_1)}{n_1} \sim 99.5\%$ - almost no repeating samples.

Then if I consider (n_2): $\frac{D(m,n_2)}{n_2} \sim 95.1\%$ - around 4.9% samples are repeated.

This means that the scalability clearly depends on the space state size/number of samples relation. As the number of samples approaches the number of possible paths in a tree, the algorithm will lose its parallel properties and even finding the exact solution is not guaranteed, since we would have to consider infinite number of samples. It can be concluded that as the tree gets smaller (the solution is closer) the number of repeated samples increases (the higher the line the more repeated samples there are). When the tree is shallow enough (depth is lower than 10) it is very significant. If the state-space is small, the impact of the parallelism will be diminished (Figure 10). Low problem complexity may be caused by the problem is simple itself.

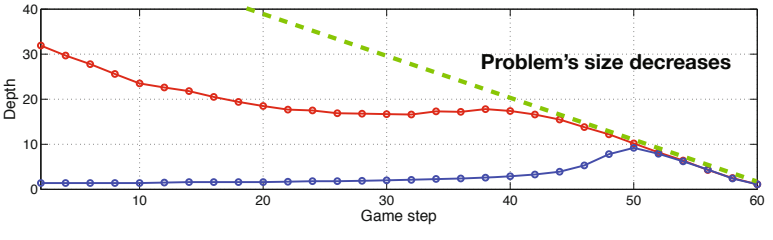

Fig. 10. Explanation of performance change caused by the problem size change

5 Conclusions

Results from Reversi and SameGame show the same problem. Both in case of CPU and GPU usage an improvement limit exists. The communication is not the problem as the simulation speed increases linearly. The most likely causes of this issue is the problem size and therefore repeating samples. We were able to gain performance improvement in both problems by increasing the thread number. Therefore at a certain point a scaling problem arose. We showed that the scaling is affected by:

1. The problem complexity - MCTS shows weak scaling
2. Random number generation, bounds for the number of unique random sequences, repeating samples
3. Exploitation/exploration ratio - the higher exploitation, the better scaling, more unique trees in root parallelism
4. The implementation itself - i.e. leaf parallelism/root parallelism.

References

1. 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)
2. Kocsis, L., Szepesvari, C.: Bandit based Monte-Carlo Planning. In: Proceedings of the EMCL 2006, pp. 282–293 (2006)
3. Chaslot, G.M.J.-B., Winands, M.H.M., van den Herik, H.J.: Parallel Monte-Carlo Tree Search. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 60–71. Springer, Heidelberg (2008)
4. Cazenave, T., Jouandeau, N.: On the parallelization of UCT. In: Proceedings of the Computer Games Workshop, pp. 93–101 (2007)
5. Kolchin, V.F., Sevastyanov, B.A., Chistyakov, V.P.: Random Allocations (1976)
6. Segal, R.B.: On the Scalability of Parallel UCT. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 36–47. Springer, Heidelberg (2011)
7. Schadd, M.P.D., Winands, M.H.M., van den Herik, H.J., Chaslot, G.M.J.-B., Uiterwijk, J.W.H.M.: Single-Player Monte-Carlo Tree Search. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 1–12. Springer, Heidelberg (2008)