An Efficient Computation Pattern for Parallel MCTS

S. Ali Mirsoleimani*†, Aske Plaat*, Jaap van den Herik* and Jos Vermaseren†
*Leiden Centre of Data Science, Leiden University
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
†Nikhef Theory Group, Nikhef
Science Park 105, 1098 XG Amsterdam, The Netherlands

***Keywords*-Monte Carlo Tree Search, Task Parallelism, Search Overhead, Computation Pattern, Pipeline**

Monte Carlo Tree Search (MCTS) repeats four steps in a loop: a path of nodes is selected from the root node to a leaf node inside a search tree (select), a new node is generated and appended to this path (expand), a simulation is performed until a terminal state in the state space is reached (playout), the search tree is updated with the acquired information from the simulated solution (backup) [1]. Each iteration of MCTS computes a *trajectory*. Figure 1 illustrates the sequential loop of MCTS. Efficient parallel implementation of MCTS algorithm is challenging due to several types of overhead:

- Communication overhead refers to the cost of sending a message from one processor to another over a network.
- Synchronization overhead is the idle time that some processors have to wait for the others to reach the synchronization point.
- Search overhead occurs when a parallel implementation of a search algorithm searches the useless part of a search space because access to non-local information is restricted.

The key to achieving a good performance in parallel search is to minimize such overheads. However, these overheads are not independent. For example, reducing search overhead usually increases synchronization and communication overhead [2]. The goal is to minimize the search overhead without increasing the other two overheads.

The first step towards designing an efficient parallel MCTS algorithm is finding possible concurrent tasks in MCTS. There are two levels of task decomposition in MCTS:

- Iteration-level tasks: in MCTS the computation associated with each trajectory is a separate task. Therefore, basing a task decomposition on mapping each iteration or trajectory onto a task might work well.
- Operation-level tasks: another level of task decomposition for MCTS is happening inside each iteration. For computing each trajectory the four MCTS operations can be treated as separate tasks.

The tasks need to cooperatively update a large shared search tree in parallel MCTS. This shared data is modified by multiple tasks and therefore serves as a source of dependencies among the tasks. In MCTS, there are two levels of data dependency:

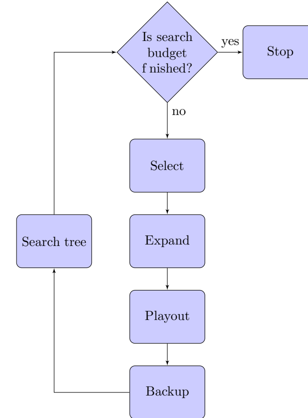

Figure 1. The main loop of the MCTS algorithm.

- Iteration-level dependency: computing each trajectory is semi-independent of the previous trajectory because trajectory *n* needs the results from previous trajectories to make an optimal decision. However, the required computations for iteration *n* is independent from the iteration *n-1*. This kind of dependency comes from Markov property. The information in each state should be enough to make a decision without knowing the history of the creation of the state. This information should be updated for each trajectory before the next trajectory being selected. Otherwise, it produces search overhead. It means going to parts of the tree which are not important or duplicate. Therefore, there is a subtle dependency between two consecutive trajectories. This dependency is not computational which means it is possible to perform the operation on two separate trajectories in parallel without fulfilling this dependency.
- Operation-level dependency: each of the four required steps for computing a trajectory is dependent to its previous step. Clearly, the expansion of a trajectory cannot occur until the selection computation is completed. Also, the simulation can not be performed until a trajectory has been expanded. The backup also needs

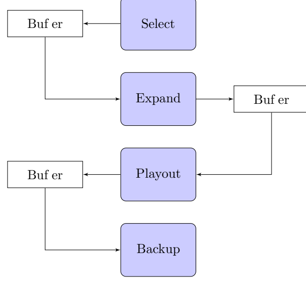

Figure 2. The MCTS pipeline flow chart.

the output of simulation.

MCTS needs a large number of trajectories or iterations to make an optimal decision. There are two possible designs based on tasks for parallel MCTS:

- Iteration-level Task Parallelism (ILTP): one way to describe the concurrency is to define the computation of each trajectory as a task [3]. The tasks have only iteration-level dependencies. The large number of tasks means that we can make effective use of any (reasonable) number of processing elements.
- Operation-level Task Parallelism (OLTP): int this case, the *pipeline* pattern can introduce parallelism into operation-level tasks in MCTS and satisfy their dependency requirements. A Pipeline is a linear flow of data from one task to another. Parallelism comes from repeating this pattern over and over. Pipeline pattern is one way to achieve concurrency for computations that are mostly parallel but require small sections of code that must be serial. In MCTS, the flow of data among operation-level tasks is regular, one-way and does not change during the algorithm. Each trajectory passes through all four steps in sequence. The steps can be executed in parallel on different trajectories if the data passes between them through buffers. Figure 2 illustrates how OLTP works.

It is important to compare the performance characteristics of ILTP and OLTP. Suppose there are 8 trajectories (p_1 to p_8) to process, on 4 processing elements (PEs), and each trajectory requires four steps (*S*, *E*, *P*, and *B*), which take 1 unit of time (*T*) each. Assuming there is no other processing that might affect the timings.

- ILTP: If you divide the 8 trajectories between 4 PEs, then each PE has 2 trajectories to process. After *4T* you will have 4 trajectories processed, after *8T*, 8 trajectories processed. Figure 3 shows the timeline for task-parallel MCTS.
- OLTP: with a pipeline, things work differently compare to ILTP algorithm. The four steps can be assigned one to each PE. Now the first trajectory has to be processed by each PE, so it still takes the full *4T*. Indeed, after *4T* you only have one trajectory processed (p_1), which

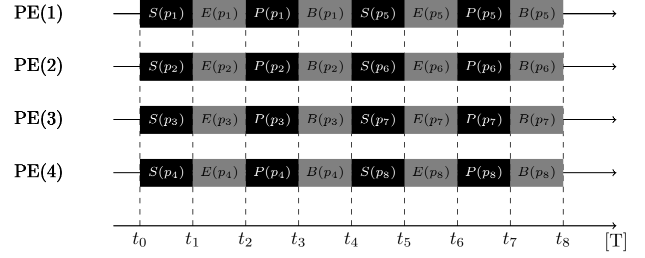

Figure 3.

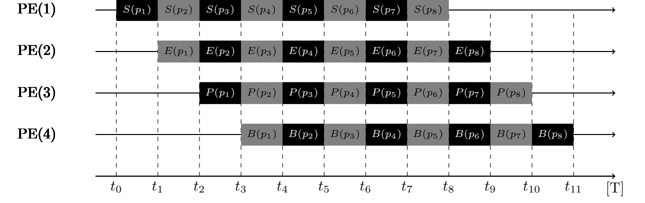

Figure 4.

is not as good as with the ILTP. However, once the pipeline is primed, things proceed a bit differently; after the first PE has processed the fourth trajectory (p_4), it moves on to the fifth, so once the final PE has processed the fourth item, it can perform its step on the fifth. You now get one item processed every *1T* rather than having the items processed in batches of four every *4T*. The overall time to process the entire batch takes longer because you have to wait *3T* before the final core starts processing the first item.

Acknowledgment

This work is supported in part by the ERC Advanced Grant no. 320651, HEPGAME.

References

[1] 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,” *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 1, pp. 1–43, 2012.

[2] Y. Soejima, A. Kishimoto, and O. Watanabe, “Evaluating Root Parallelization in Go,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4, pp. 278–287, Dec. 2010.

[3] S. A. Mirsoleimani, A. Plaat, J. van den Herik, and J. Vermaseren, “Scaling Monte Carlo Tree Search on Intel Xeon Phi,” in *Parallel and Distributed Systems (ICPADS), 2015 20th IEEE International Conference on*, 2015.