

ECSE 420 - Project

Final Report: Monte Carlo Tree Search Parallelization



Anna Bieber - 260678856
Kartik Misra - 260663577
Patrick Ghazal - 260672468
Jacques Zhang - 260733393

Date of Submission: Wednesday December 3rd, 2019 Instructor:
Professor Zeljko Zilic

Table of Contents

I.	Introduction	3
II.	Problem	3
III.	Implementation	5
IV.	Results	8
V.	Conclusion	9

I. Introduction

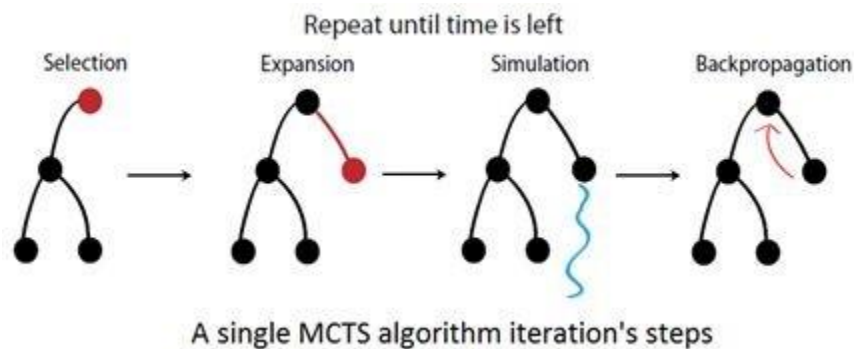
The goal of this project was to choose a topic and parallelize in some form, this included but not limited to a shared-memory platform, a distributed memory platform, or a massively parallel SPMD platform such as a GPU. For this project we decided to work on parallelizing the Monte Carlo Tree Search (MCTS) algorithm which is often used in AI for making optimal decisions. Research interest in MCTS has risen sharply due to its great success with computer Go and potential application to a number of other difficult problems. This report outlines the start point, the implementation and the results obtained.

II. Problem

a. Context

In the big picture: Monte Carlo uses a tree data structure to search and calculate outcomes of random simulations in order to find the most optimal solution. More specifically, MCTS builds its decision tree by iterating over the following 4 phases:

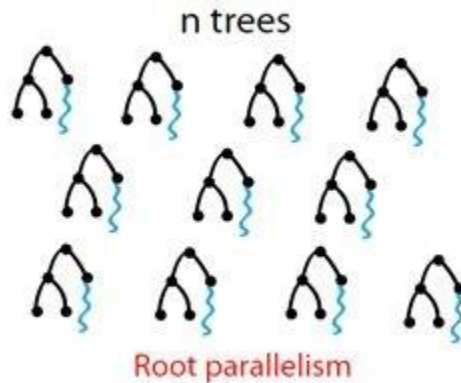
1. **Selection:** it finds the best node to start exploring (based on the specified criteria), until it comes across a node with unexplored children or the game is over.
2. **Expansion:** If the current selected node has unexplored children, select a child and add it to the decision tree.
3. **Rollout:** it simulates the game until the end from the currently selected node.
4. **Back Propagation:** it takes the rewards received by each player at the end of the game and pass those values back up the tree to update all node statistics.



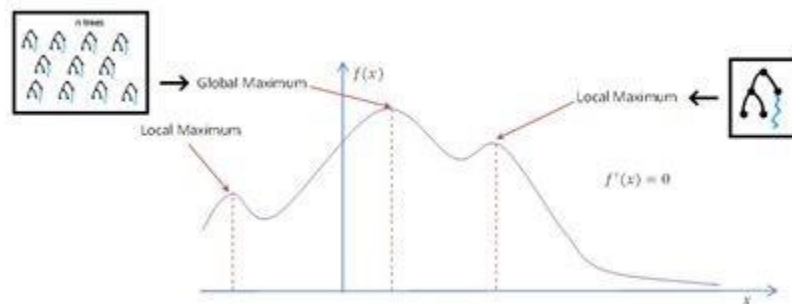
Finally, after some number of iterations, the algorithm outputs the children move with **the most number of simulations** which corresponds to the optimal move to choose. However, the problem is that: although this algorithm guarantees a solution, it is **not** necessarily the best one as it could be **sub-optimal**. In the next section, we elaborate on this idea and propose a solution using parallel computing.

b. A parallel Solution

The approach used in this project to solve the problem is called '*root parallelization*'. The idea is to start with n Monte Carlo trees that will each perform many iterations of the 4 phases (selection, expansion, simulation, backtracking) in parallel. We would have n threads, each of which will output its own suggestion of the 'next most optimal move'.



In the end, the results are summed up and the best move is chosen. The advantage of *root parallelization* is that, by building more MCTS trees, we reduce the effect of being stuck in a local max:



This means, if we were to run the algorithm on just a single tree, at some point in the game where we have many options to choose from, we might get a “good enough” move but it might not be the best move. However, by having many trees running the algorithm in parallel, we increase the chances of finding the **true global maximum** in an efficient manner.

III. Implementation

a. Methodology

As a problem to be solved, we chose to build an AI agent for the TicTacToe game. We expanded the original game to dimensions greater than $3 * 3$ in order to have a bigger branching factor and to achieve reasonable complexity for tree searching.

The technology that we “attempted” to use was CUDA. The steps are as follow:

1. We first create N threads (for N Monte Carlo trees) to be run in parallel.
2. Each thread takes as **input** : a copy of the current game state and the root of a new Monte Carlo tree.
3. Each thread then performs the **4 phases** of the MCTS algorithm for a chosen number of iterations.
4. At the end, each thread outputs a move it judges to be the most optimal.
5. Finally, we take the count for each optimal move outputted by the threads and choose the one with the highest count which we use as the **best move** to make.

However, we encountered issues when mapping TicTacToe C code to CUDA and also we had some trouble with tree pointers and GPU memory limitations. Instead of struggling for days and for the sake of having a **working proof of concept**, we decided to implement a Java-Thread version of the parallel Monte Carlo. Our Java program follows the same logic as explained for CUDA and works well on a smaller problem size. At the end, it actually outputted some interesting results to be discussed in the last part of this report.

b. The Code

As the focus of this project was mainly to **improve the MCTS algorithm** using **parallel computing techniques**, we did not bother too much on the general boilerplate code that implements TicTacToe with the Monte Carlo algorithm; thus, we found an online java implementation of TicTacToe + MCTS (http://codegatherer.com/mcts_tic_tac_toe.php).

The java classes that we added/modified to implement *root parallelization* are:

- **MctsThread.java** : this class represents a thread object and encapsulates its key **parameters** and **operations** that a given thread has to perform. As a reminder, each thread in the *root parallelization* algorithm performs multiple iterations of the 4 phases of the MCTS. At the end, each thread outputs the move that it finds to be the most optimal.

The parameters needed for a thread include: the current `MctsTicTacToePlayer`, the current `TicTacToeGame` board state, and a `ConcurrentHashMap` shared among all threads which stores the result outputted by each thread.

The `run()` method calls `getMove()` on the current `MctsTicTacToePlayer` and passes it the current state of the board; `getMove()` essentially performs many iterations of the 4 phases. Finally the `run()` method prints the computation results of its running thread (time taken, position chosen).

- **TicTacToeGame.java** : in the `play(player1, player2)` method of this class, `MctsThread`'s are created and launched to perform root parallelization. In the end, the results of the threads (stored in the `ConcurrentHashMap`) are used to decide the best move (global max move) by choosing the most popular move (with the most count) among all the optimal moves suggested by the threads.

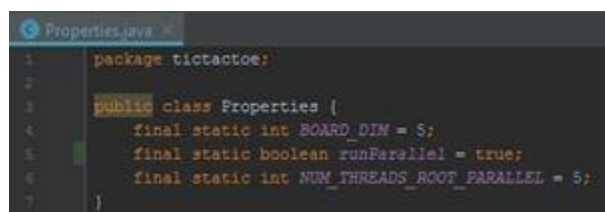
c. How to run the code

We provided the source code in the submission zip folder. You can use IntelliJ or Eclipse to open and run the java code following these steps:

1. Replace the `src` folder in your work environment by the **src** folder in the zip file.
2. Next, you have to run the main class inside `SingleGameRun.java`. For example, if you are using IntelliJ, you can select '**Edit Configurations**' and add an **Application** configuration where the Main Class is: `tictactoe/SingleGameRun`.
3. After the program starts running, the first thing you are going to see in the output terminal is a list of options (moves) that you (the player) can make. Choose a move. In the next turn, the parallelized MTCS player will make its move and associated computation data will also be displayed.

Changing the configurations

We created a **Properties.java** file where you can modify the configuration variables based on how you want the program to run.



```
1 package tictactoe;
2
3 public class Properties {
4     final static int BOARD_DIM = 5;
5     final static boolean runParallel = true;
6     final static int NUM_THREADS_ROOT_PARALLEL = 5;
7 }
```

For example, you can test with different board dimensions (modify `BOARD_DIM` variable to achieve larger branching factor and complexity). To get results for sequential run of the program, set the `runParallel` boolean to false. You can also run the *root parallelization* with different number of threads.

d. C/CUDA attempt (extra work done to try and utilize CUDA)

We attempted to build this program using C and CUDA basing ourselves on the lab material. However, in attempting to do so, we ran into several issues, which, as a result, have not given a working CUDA project, but we have included the code regardless to show our work.

```
global void simulate(Node node, bool currentTurn, Board board) {
    int start = rand() % node.numChildren + 1;

    double maxHeuristic = rand() / double(1000);
    int nodeIndex = 0;

    Node* tempNode;

    //go to depth 3
    for (Node* n : node.children) {
        for (Node* n1 : n->children) {
            for (Node* n2 : n1->children) {
                double random = rand()/double(1000);
                n2->setHeuristic(random);
                if (n2->getHeuristic > maxHeuristic) {
                    maxHeuristic = n2->getHeuristic;
                    nodeIndex = n2->getKey;
                    tempNode = n2;
                }
            }
        }
    }

    for (int i = 0; i < tempNode->depth-1; i++) {
        tempNode = tempNode->parent;
    }
    int index = tempNode->key;
    board.getNodeByIndex(index).setStatus(currentTurn);
}
```

To make things simple, we only allowed the program to visit tree nodes up to a depth of three, after which we would simply assign a random number to that node as its heuristic and then backpropagate the result back to the current turn node.

When scaling this on a larger scale grid, we ran into many problems:

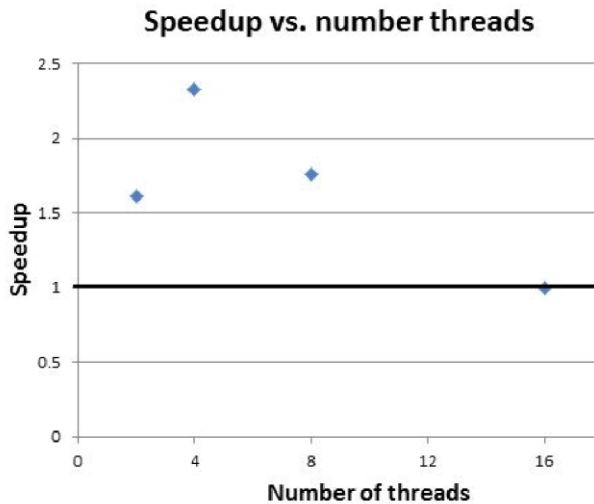
- After a few iterations of the heuristic being run, some of the pointers referencing nodes in the Monte Carlo Search Tree returns null pointers or simply didn't point to the right reference. We believe that an error in the design in the attempt to make memory usage as efficient as possible resulted in incorrect pointer and reference assignment
- When scaling the grid to a large size, we got a few odd errors such as memory buffer overflow or unallocated memory that crashed the program. We were unable to fix this and therefore concentrated our time on the Java Thread based implementation that was more promising.
- We have included our CUDA/C code in this submission for reference, but it is not the main source code for this project.

IV. Results

The results that we got match our initial expectations in terms of **performance** and **optimization correctness**.

a. Performance

Our first expectation was that the speedup would increase as the number of threads increases (until some max threads threshold). Here are the results of our experimentation:



```
*** SEQUENTIAL TIME: 5185  
-> thread 5 built a MCTS tree in [2933 milliseconds] that chooses move: [x = 4.0  
-> thread 4 built a MCTS tree in [2935 milliseconds] that chooses move: [x = 0.0  
-> thread 7 built a MCTS tree in [2874 milliseconds] that chooses move: [x = 0.0  
-> thread 1 built a MCTS tree in [2924 milliseconds] that chooses move: [x = 0.0  
-> thread 2 built a MCTS tree in [2977 milliseconds] that chooses move: [x = 0.0  
-> thread 0 built a MCTS tree in [2990 milliseconds] that chooses move: [x = 3.0  
-> thread 3 built a MCTS tree in [2943 milliseconds] that chooses move: [x = 1.0  
-> thread 6 built a MCTS tree in [2950 milliseconds] that chooses move: [x = 1.0
```

For the plot on the left, the experimentation was done with some variables kept **constant** : we required the very first move from the MCTS computation, the first move chosen by the human was the center cell, and we worked on a 5*5 board.

As can be seen in the above graph, the positive result is that: the parallel time is smaller than the sequential time until 16 threads where the thread overhead starts taking over and having more threads becomes useless. The overall trend shows that speedup increases only up to around 4 threads before starting to decrease. This shows that it would better to run the search on GPU to achieve higher level of parallelization with more processors (future work).

b. Optimization Correctness

Our second expectation was that, in the early stages of the game, the threads would output a bigger variety of optimal moves compared to the ending stages of the game. This is because there are more moves to choose from in the beginning when the branching factor is larger.


```

Run: Play x
- thread 0 started
- thread 1 started
- thread 2 started
- thread 3 started
- thread 4 started
- thread 5 started
- thread 6 started
- thread 7 started
+++ SLEEP TIME: 14.399999999999999
-> thread 5 built a MTCS tree in [2657 milliseconds] that chooses move: [x = 4.0, y
-> thread 1 built a MTCS tree in [4250 milliseconds] that chooses move: [x = 1.0, y
-> thread 2 built a MTCS tree in [4412 milliseconds] that chooses move: [x = 0.0, y
-> thread 6 built a MTCS tree in [4348 milliseconds] that chooses move: [x = 4.0, y
-> thread 7 built a MTCS tree in [4370 milliseconds] that chooses move: [x = 3.0, y
-> thread 3 built a MTCS tree in [4341 milliseconds] that chooses move: [x = 0.0, y
-> thread 0 built a MTCS tree in [4463 milliseconds] that chooses move: [x = 0.0, y
-> thread 4 built a MTCS tree in [4461 milliseconds] that chooses move: [x = 4.0, y

*** (MOVE, OCCURRENCES) MAP: [(4,0)=2, (1,3)=1, (0,4)=1, (0,0)=2, (3,3)=1, (4,4)=1]
+++ BEST MOVE: [x = 4.0, y = 0.0]
----O
-----
--X--
-----
-----

```

```

Run: Play x
- thread 0 started
- thread 1 started
- thread 2 started
- thread 3 started
- thread 4 started
- thread 5 started
- thread 6 started
- thread 7 started
+++ SLEEP TIME: 12.0
-> thread 4 built a MTCS tree in [1570 milliseconds] that chooses move: [x = 2.0, y
-> thread 1 built a MTCS tree in [2614 milliseconds] that chooses move: [x = 2.0, y
-> thread 2 built a MTCS tree in [2611 milliseconds] that chooses move: [x = 2.0, y
-> thread 5 built a MTCS tree in [2593 milliseconds] that chooses move: [x = 2.0, y
-> thread 6 built a MTCS tree in [2575 milliseconds] that chooses move: [x = 2.0, y
-> thread 0 built a MTCS tree in [2634 milliseconds] that chooses move: [x = 2.0, y
-> thread 3 built a MTCS tree in [2617 milliseconds] that chooses move: [x = 2.0, y
-> thread 7 built a MTCS tree in [2603 milliseconds] that chooses move: [x = 2.0, y

*** (MOVE, OCCURRENCES) MAP: [(2,0)=6]
+++ BEST MOVE: [x = 2.0, y = 0.0]
O-O-O
--XX-
--X--
-----
-----

```

The left figure shows the output of the threads after choosing the first move of the game (many options, so threads output different local max). The right figure shows the output of the threads after choosing the third move of the game (less options, so all converge to global max).

As can be seen, for the first move, threads don't get stuck in a local optimum which leads to a more thorough search.

V. Conclusion

This project allowed us to work on a topic that is of great interest to us and use the knowledge learned in class and in the labs and apply it to a “real” problem. It also allowed us to work in a team and learn from each other as we all came in with different knowledge. We really enjoyed working on this project and the different concepts we learned along the way.