Parallel MCTS-Minimax Hybrids

15618 FINAL PROJECT REPORT

SAI BHARGAV YALAMANCHI

PETER DE GROOT

OUR CODE BASE

OpenMP - https://github.com/ysaibhargav/simple-chess
CUDA - https://github.com/ysaibhargav/simple-chess/tree/CUDA/
CUDA Breadth First Search Attempt - https://github.com/ysaibhargav/simple-chess/tree/CUDA_BFS/

SUMMARY

We proposed, implemented and investigated various designs and algorithms for parallel Monte Carlo Tree Search - Minimax models. In this report, we describe how we achieved near subsecond execution times (speedup of ~10x on sequential models) with 16 threaded instances (OpenMP) to find the first optimal move in mate in 4 chess puzzles on latedays, as well as our efforts to parallelize pure minimax with CUDA.

BACKGROUND

Minimax with alpha-beta pruning has been the state of the art algorithm for chess engines (until the advent of AlphaZero recently). A drawback of using bare bones minimax is that the search space becomes combinatorially large and unwieldy to perform brute force enumerations in order to determine the best move. Monte Carlo Tree Search (MCTS) is a best-first tree search algorithm that focuses on the most promising lines of play. However it often succumbs to traps in games like chess [5] either due to

1. Missing a critical move,
2. Underestimation of utilities of certain terminal states,
3. Overestimation of utilities of intermediate suboptimal opponent moves.

Why hybrids?

While MCTS has a tendency to latch on to trap states, in sub-trees of games where such traps do not exist it tends to uncover the optimal action very quickly [5]. And while minimax guarantees finding the optimal action, it is computationally intractable for large game trees as the number of states suffers an exponential explosion in the branching factor.

The authors of [1] motivate the application of MCTS-minimax hybrids; the key idea is to trigger minimax searches at shallow sub-trees which are believed to contain terminal states. This information can be used to guide tree growth as well as avoid traps. Such hybrids would combine the benefits of MCTS and minimax, resulting in a best-first trap avoiding algorithm with only marginal increases in computational costs with carefully chosen heuristics and trigger criteria for minimax.

Our main focus in this project is the design and investigation of such algorithms along with methods to parallelize them for the task of solving mate in n puzzles in chess.

MCTS Background

A pure (non-hybrid) MCTS approach repeats a four-phase loop to iteratively grow the tree and determine the best action from the root node. Each iteration typically results in the expansion of the tree by one node and a single simulated game until termination. The iterations consist of the following four phases

1. **Selection** - the tree is traversed starting from the root node until a non-terminal node is found which hasn't been fully expanded. The traversal is done with the UCB selection policy which attempts to balance exploration and exploitation.
2. **Expansion** - the selected node is expanded by choosing one of its successors that is not already part of the tree.
3. **Simulation** - the game is simulated from the expanded node until termination with random actions from all players. MCTS trees are proven to converge in optimal moves for such random rollouts.
4. **Backpropagation** - the result of the simulation is backpropagated along the traversed path in the current iteration. The value estimates and the visitation counts for each of the nodes that lie on this path are updated.

DEFINITIONS

A **game state** is defined by a chessboard layout, which player moves next and the remaining depth until mate is delivered. Game states are operated on by minimax and there exists a one-one correspondence between game states and MCTS tree **nodes**.

Each node in the MCTS tree is associated with the player whose action would result in the node's corresponding game state. We call this player the **node's agent** and this action the **node's action**.

In our mate in n chess puzzles, all states at depth n are **terminal**. If the model finds a mate at move n, it is considered to be a **victory**. If it fails to find a mate at move n, it is considered a **loss**.

A **proven victory** is a victory that is attained with optimal play regardless of the moves made by the other player.

A **proven loss** is a loss that is guaranteed if the opponent plays optimally regardless of the moves made by the node's agent.

We say that a **node is proven** if it has been established via minimax search that there exists a line of play for the node's agent which will result in a proven victory or proven loss.

AI is the hybrid solver which is also the black player in the game.

Hybrids we investigate

[1] presents three different flavors of hybrids. Of the three, MCTS-MS (Monte Carlo Tree Search-Minimax in the Selection and Expansion phase) is the most suitable for our chess puzzles.

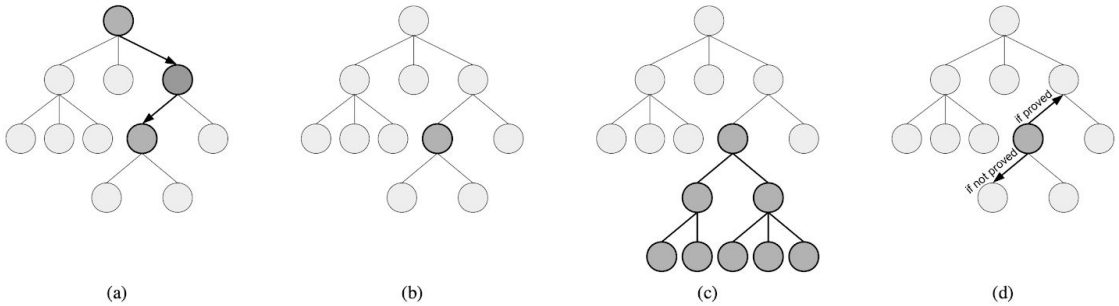


Fig. 2. The MCTS-MS hybrid. (a) Selection and expansion phases. The tree is traversed in the usual fashion until a node satisfying the minimax trigger criterion is found. (b) In this case, the marked node has reached a prespecified number of visits. (c) A minimax search is started from the node in question. (d) If the minimax search has proved the node's value, this value can be backpropagated. Otherwise, the selection phase continues as normal.

The key idea of MCTS-MS is to trigger minimax evaluations to prove a node's value in the selection phase based on some preset trigger criteria. Once a node is proven, this result is backpropagated without the need for further rollouts. This provides a useful signal for tree growth - it lets the tree search avoid branches that are proven to be losses and directs it towards branches that are proven to be victories. By construction, in our use case of mate in n puzzles, every node is provable.

We propose and implement caching of proven node results in our node data structures. Instead of triggering minimax searches repeatedly for qualifying nodes, we cache the result of the first minimax call and use them when the node is revisited.

We investigate the following minimax trigger criteria -

1. **MULTIPLE_VISITS** - (proposed in [1]) trigger minimax search if a node has been visited at least 10 (=VISIT_THRESHOLD) times (OpenMP)
2. **NONZERO_WINS** - (our proposed criteria) trigger minimax search if a previous simulation resulted in a victory for the AI (OpenMP)
3. **ALWAYS** - always trigger minimax search (CUDA)

Motivation behind using MCTS-MS for mate in n chess puzzles

If a node happens to be visited several times, there is a good chance that there is a winning move sequence (for the AI) originating from the node by virtue of UCB's tendencies to exploit during selection. It is worth confirming this belief via a minimax search. This motivates the MULTIPLE_VISITS trigger criteria.

If a possible winning sequence (for the AI) is found through random actions in a rollout, the next time one of the nodes on this path is visited, instead of performing rollouts we conduct minimax search to prove the node's value (the winning sequence provides a signal for the existence of a proven victory). This motivates the NONZERO_WINS trigger criteria.

With the ALWAYS trigger criteria, minimax searches are essentially called from randomly picked children of the root node. Statistically this is 2x faster than deterministic minimax.

Workload and parallelism

As described earlier, our focus is on mate in n puzzles. These puzzles all have a forced mate for black. We use the Auerswald collection of chess PGNs [6]. Some of the FENs from this collection we used during our development process are

1. `3r1bk1/ppq2ppp/2p2p2/2P5/QP1P3n/2N1P2P/P3B3/R4RK1 b - - 0 3 (MATE IN 2, TREE DEPTH 4)`
2. `r3k2r/pp2npp1/1q6/2b1P1p1/4Np2/2P5/PP4PP/R1B1QR1K b kq - 0 1 (MATE IN 3, TREE DEPTH 6)`
3. `5rk1/5r2/1NRR2p1/3PQ3/7p/7P/6PK/5q2 b - - 2 2 (MATE IN 4, TREE DEPTH 8)`

On average, there are 35(=b) legal moves for any given player in a game of chess [7]. A mate in n puzzle would result in a tree of size 1+b+b^2+...+b^(2n) ~ b^(2n), which is ~1.9b and ~2.2t nodes for mate in 3, 4 puzzles respectively. Trees of such sizes can vastly benefit from parallelism during the MCTS process, as each thread can help grow different parts of the tree independently with a carefully designed scheme. We present such schemes in the approaches section.

Data Structures

The code base we are using [8] contains the following structs: `ChessBoard, Move, Action, State`.

The `ChessBoard` struct contains the description of the current board in a char array of length 64. Each char contains the piece information (king, queen, rook, etc.) and whether it has been moved from its starting position (en passat, castling, and the option to move a pawn by two squares on its first move depend on this information). There is a char for the black and white king positions to accelerate the checks whether the king is in checkmate or stalemate.

The `Move` struct contains a char for the figure which is moving, the location it is moving from and the location it is moving to, and the piece it is capturing, if any.

The `Action` struct contains the `Move` that would be made is the action was applied, the minimax value associated with the move and a list of maintenance moves.

The `State` struct contains the `ChessBoard` as it currently stands, the depth integer to determine if the speculation range is too deep for our mate in x, and an integer to determine whether it is white to move in said position.

The `TreeNodeT` struct contains `State`, `Action` members along with pointers to parent nodes and proven child nodes. In addition to these it contains `agent_id`, the node's agent's ID, `num_visits`, `value`, `aux_value`, `depth`, `proved`, `is_root` scalar members. It also has a member `lck` of type `omp_lock_t` and `children`, `actions` vectors.

The MCTS search tree data structure is formed using a collection of `TreeNodeT` type nodes, with pointers linking successive hierarchies.

APPROACHES

Sequential code

We implemented our sequential code using a simple reference C++ chess code base [8] and C++ MCTS [9]. We wrote a version of minimax that finds victories/losses from a given starting state. Since MCTS is not the popular choice for an AI chess engine, implementing a correct MCTS-minimax hybrid posed an interesting challenge and we spent the first few weeks of the project setting it up. Our sequential hybrid implementation is 2x faster on average than our reference minimax code on chess puzzles in agreement with theory. For our mate in n solver to work, we had to load the FEN which contains a chess board's position and whether castling is still possible, among other pieces of information. In order to get the FEN loaded correctly, we had to write a parser to take the FEN text and place the pieces in the correct location in the chessboard array with the correct properties.

The chessboard is rendered in text format in a terminal and we specify moves by entering the start and end coordinates of the piece we want to move. An example puzzle looks like the following

```
MATE IN 3 PUZZLE
Loaded FEN: r3k2r/pp2npp1/1q6/2b1P1p1/4Np2/2P5/PP4PP/R1B1QR1K b kq - 0 1


   ___ ___ ___ ___ ___ ___ ___ ___
  |.T |   |   |   |.K |   |   |.T |
8 |___|___|___|___|___|___|___|___|
  |.o |.o |   |   | H |.o |.o |   |
7 |___|___|___|___|___|___|___|___|
  |   | Q |   |   |   |   |   |   |
6 |___|___|___|___|___|___|___|___|
  |   |   |   B   |   x   |   o   |
5 |___|___|___|___|___|___|___|___|
  |   |   |   |   |   h   |   o   |
4 |___|___|___|___|___|___|___|___|
  |   |   |   x   |   |   |   |   |
3 |___|___|___|___|___|___|___|___|
  |.x |.x |   |   |   |   |.x |.x |
```

```
2 |___|___|___|___|___|___|___|___|
  | t |   |.b |   | q | t |   | k |
1 |___|___|___|___|___|___|___|___|
    A   B   C   D   E   F   G   H

>> a1b1
```

In our setup, the AI is always black. In the interactive mode, a human player moves the white pieces. In the non-interactive mode, the AI makes the first move and the program exits. *Our engine only works for puzzles with a forced mate for black and black moves first.*

Root parallel MCTS-MS

In a root parallel scheme, threads have their own copies of the search tree and grow them independently of one another. The threads do not share information with one another. At the end of the MCTS iterations, all the trees are merged together (corresponding nodes' value estimates, visitation counts are aggregated). The best move is then determined from the merged tree [2].

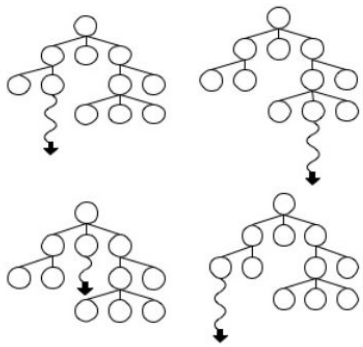


Figure (root-parallel): Suppose there are 4 threads, each thread grows a tree independently of one another in a root parallel approach

For our mate in n chess puzzles, if there exist winning sequences with a unique first (black's) move, a thread-wise work allotment scheme can be designed at the first level of the tree. For our use case, a general root parallel approach would have each of the threads trying to identify this first move of the winning sequence. Each thread would grow trees independently using its own random number generator (in the underlying growth phases); this implies there is no incentive for the threads to all grow different/non-overlapping parts of the tree. This is a **disadvantage of a general root parallel scheme**, and to remedy this **we divide the immediate successors of the root node as evenly as possible among the threads**. This guarantees that the threads work on independent branches of the tree.

Both our trigger heuristics always result in minimax calls from the immediate successors of the root node, so once a thread determines that such a successor results in a proven victory, we stop work on all other threads and return the successor node's action.

With OpenMP, since it is not possible to break/branch out of a parallel for region, we implement **work cancelation** using a shared variable. When a thread finds a proven victory for one of the root node's successors, we want to immediately stop any work the other threads are doing. If they are in the middle of a minimax search, waiting until completion can result in long delays. So as soon as a thread finds a proven node, we set a shared variable, *found_proven_node* to true, and within a **critical section** set another shared variable *proven_move* to the found proven move. The variable *found_proven_node* is **atomically** read at the start of the selection phase as well as within the minimax recursions. If it is found to be true, minimax triggers cascading returns of a default result value (which will never be used) and all subsequent parallel for MCTS iterations are skipped.

We break down our MCTS state struct into its scalar attributes (64 char array for the board squares and their occupants, black and white king positions, depth and player's turn) in order to successfully **offload to the Xeon Phi**. The state struct is reconstructed using these fields in the MIC code block. We then initialize an MCTS root node with this struct. The tree is constructed using this root node in the same region within the parallel for OpenMP areas by the threads.

An **alternate approach we tried** is with **shallow minimax searches**. We want to keep minimax searches shallow to accommodate puzzles of greater depths. However this means we can no longer find proven children of the root. To pick a move at the end of the MCTS iterations, we simply pick the most visited child of the root as done in conventional MCTS. To do this we have a shared array of size NUM_THREADS. This array records the per thread most visited root children. At the end of the MCTS iterations we pick the most visited 'per thread most visited' child from this array. The main issue with this approach is that it often fails to find the correct move. We hypothesize that the MCTS succumbs to traps, discussed in our background section. We verified this belief through sequential experiments with shallow minimax triggers in the selection process. This prevents us from looking at very deep (mate in 5+) puzzles with our primary approach.

Tree parallel MCTS-MS

In a tree parallel approach, the search tree is shared by all the threads, therefore each thread helps grow the same tree [2]. We implement tree parallel with local mutexes instead of a single global mutex which would lock the entire tree. **Using a global mutex is not suitable for our hybrids** because the lock is held for the entire duration of selection (which can possibly trigger minimax), expansion and backpropagation phases. While a thread is in possession of this lock, the other threads can only run simulations, whose duration is much less compared to the total time for the other three phases. This will result in a lot of idle time for the threads that do not have the lock.

With local mutexes, we only lock a single node if it requires modification (updating its value estimate or visit counts, addition of a successor, proving its value). To achieve this we add a member variable `lck` of type `omp_lock_t` to the `TreeNodeT` class. This is initialized in the node's constructor function and destroyed in the node's destructor function with `omp_init_lock` and `omp_destroy_lock` respectively. Each thread can only lock one node at a time, note that multiple nodes can still be locked at a time, each by a different thread. The number of locks is therefore equal to the number of nodes in the tree at any given point.

To prevent different threads from traversing the same subtrees or branches, we induce an **occupancy loss** to a node's value estimate should a thread traverse it. Upon backpropagation we reverse the occupancy loss. This is in line with the introduction of a virtual loss as proposed in [2]. The occupancy loss will reduce the value estimate of a traversed node and will therefore be less favored during exploitation, unless the value estimate of the node is significantly better than the others.

Locks are placed at the following locations

1. `get_best_uct_child` - this function determines the best child using the UCT exploration-exploitation value estimator of a given node. In order to estimate this value, the function needs to read the childrens' value estimates and visit counts. A lock is used to access these members of each child. A lock is also used on the node whose best UCT child is wanted. This will prevent other threads from expanding the node when we traverse its successor list to determine the best UCT child.
2. `proven_move, found_proven_move` - we use the root node's lock to set the `found_proven_move` flag and to write to the `proven_move` struct. We need these writes to occur atomically, otherwise data may be overwritten (interleaved in case of multiple proven moves found by threads simultaneously) and the root node's proven move information may be lost. This is the only lock used in root parallel.
3. Minimax evaluations during selections - while the node's value is being proven, we do not want other threads to also start a minimax search. This is just redundant work and does not add any value.
4. `proven_child` pointer values - this is applicable for shallow minimax searches, where the search is triggered for non-immediate successors of the root node. We cache the pointers to the proven child nodes in their parents. In order to do this we lock the parent node.
5. Expansion - we do not want multiple threads to add children to the node at the same time. This will result in child loss by overwrites. We lock the node which undergoes expansion to prevent this.
6. Backpropagation and occupancy loss addition/reversal - when a node's value estimates and visit counts are updated, these must happen atomically.

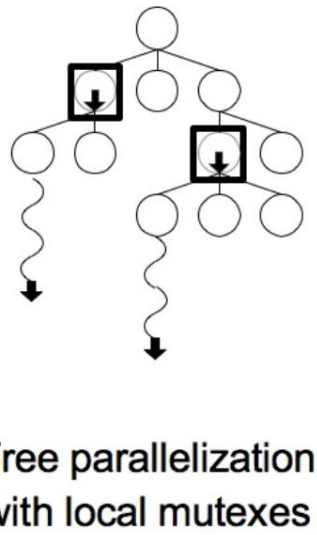


Figure (tree-parallel): Each thread helps grow the same tree but uses a local lock to update a node

Leaf parallel MCTS-MS, with minimax always triggered

Always triggering minimax means that the minimax searches are conducted at the children of the root node. We attempt to parallelize minimax with CUDA to make it more tractable.

Parallel minimax

Our target machine was the Tesla K40m GPU available on the latedays cluster. Said GPU had the CUDA 7.5 libraries installed.

For CUDA, we kept the depth first search component of the minimax algorithm to see how much the parallelism would accelerate the search. The approach was that when the depth first search hits the target depth, the CUDA algorithm triggers and performs the next stage of the calculation in parallel in order to accelerate outputs. When the target depth is reached by the sequential depth first search, the possible moves are generated and placed into an array of States which is copied to the GPU. The GPU executes the remaining recursive depth first search in parallel across the passed States. Each thread performs the depth first search on the `State` that it is assigned.

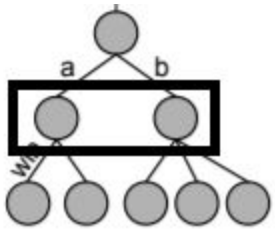


Figure (Leaf-Parallel): If the target depth for the CUDA implementation is set to the boxed area above, the States resulting from all possible moves of the node above are sent to the GPU for the rest of the depth first search.

A number of the stumbling blocks we ran into during this branch of the project include the fact that CUDA does not support any of the std libraries. This necessitated a rewrite of all functions that need to be called on the device in order to not use any of the standard libraries the original algorithm made use of (ex. vector, iterator). After which, the next attempt was to use the thrust libraries to imitate vector usage on the GPU. However, after rewriting the code again, we found out that the thrust vectors' length could only be modified on the host machine, disqualifying it from use in our project since we wanted it to dynamically modify its length. This meant we had to change the data structures on the GPU to an array. We chose to declare the array at a fixed length of MSIZE (100) since the average number of valid moves retrieved by a board position in chess is 35 [7]. This array would have to be allocated on every recursive call during the depth first search. At this point, all functions that were called by the CUDA cores were written in external class files and were re-written to manipulate an array. However, the program had issues calling those class files and having them manipulate data structures. Thus the external functions needed to modify the data structures were copied into the CUDA file and changed to non-class functions that would modify the data values passed into it (the data contained within the `State`). This resulted in a CUDA code that was fully contained in the file. This implementation worked at very shallow depths, but resulted in memory access errors when the depth increased (more recursive calls in the CUDA search). This problem was ameliorated by placing the 400 byte `Move` array that is placed on the stack with every recursive call into dynamic memory. The same process was performed for any new `States` or `ChessBoards` that were needed during the function call. We surmise that this memory error is due to the stack on each of the cores running out of space and thus causing memory errors to result. Moving the data structures onto the heap eased some of the pressure on the stack but left us limited to a depth of 2 as our maximum CUDA calling depth.

We made an attempt to increase parallelism in the CUDA implementation by collecting all the possible states at the target depth through a breadth first search and then sending all of those states to the GPU. However, the current implementation of the code is not returning the correct move. The timing of the return logic, even though the return value is currently incorrect, does produce an interesting result: it takes about 4 seconds for the data to be processed and returned even though the kernel takes only a second. This seems to indicate that a greater trigger depth and a much deeper search tree would be required before any CUDA optimizations would be considered beneficial. This would be a good path for future work in this branch.

RESULTS

Auerswald PGN collection description

The Auerswald PGN collection contains a collection of game position and the moves made that resulted in a win. The collection was put together in order to show different chess tactics in said provided positions with the actual moves played from said position and the game result. Only a subset of the positions in the collection contain forced mates and thus we had to filter through

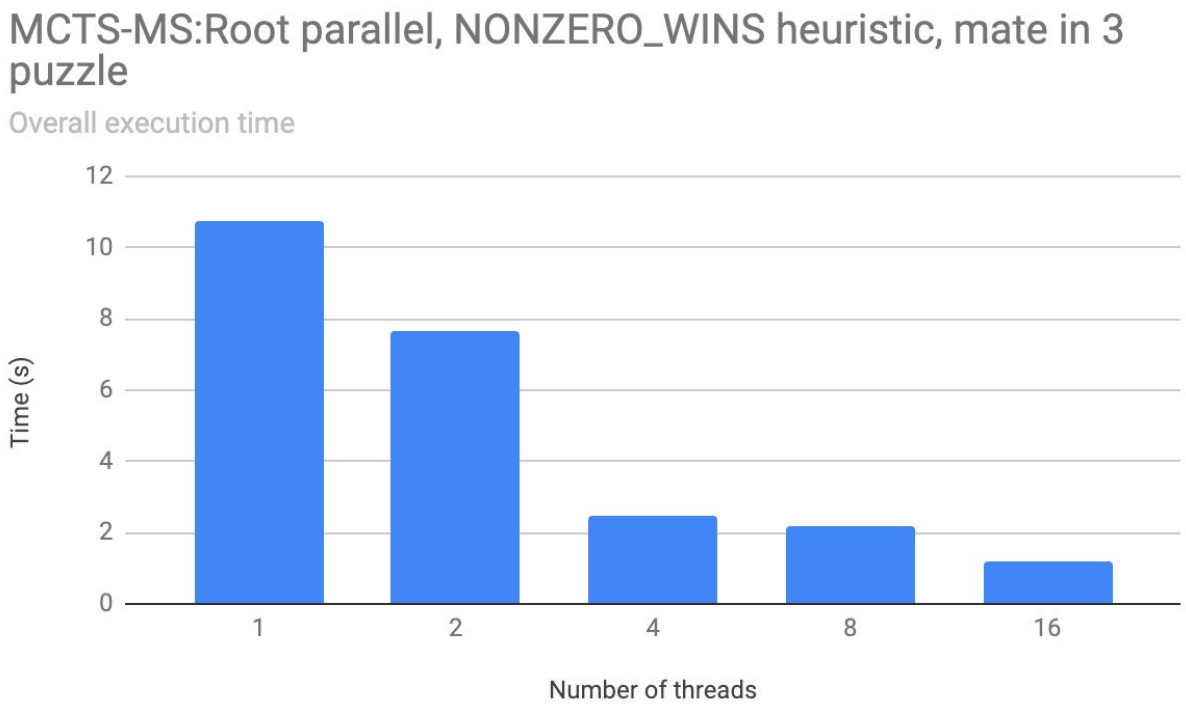
the collection for games that showed a forced mate in x to use as our inputs. We could then check the outputs based on the returned move and determine if it matched the move contained in the collection.

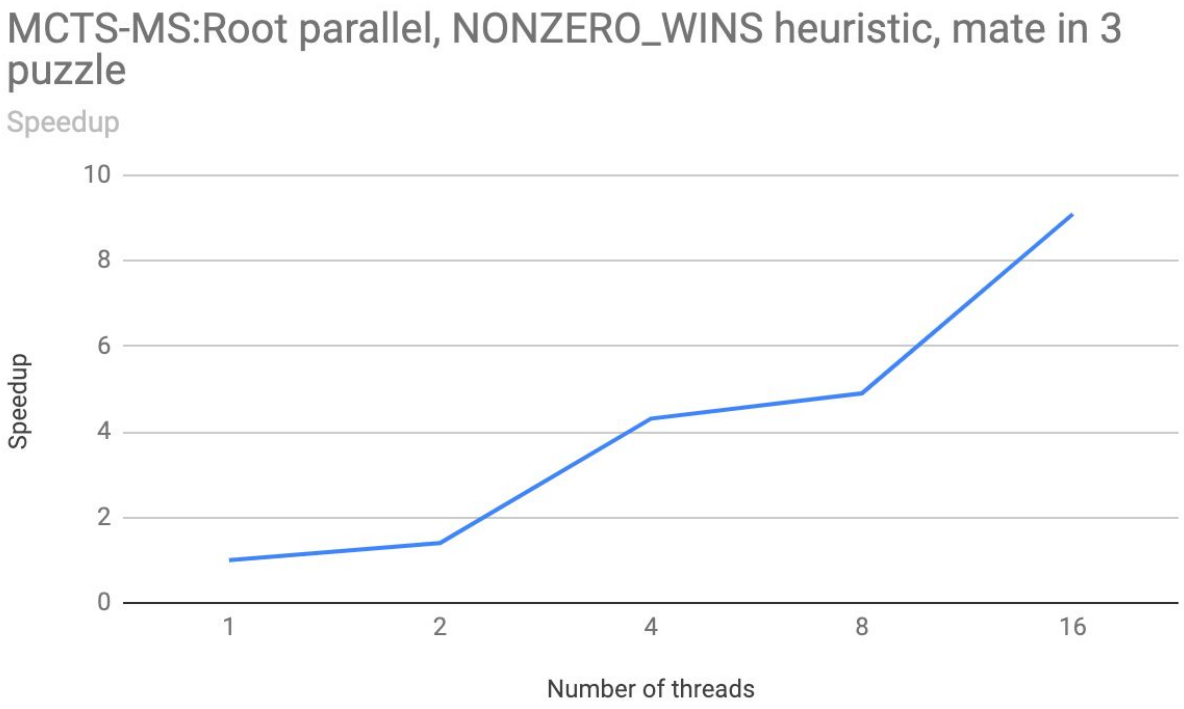
MCTS-MS plots

For the execution time results in all plots in this section, we take the median of 5 successful runs for the puzzles mentioned. The execution time is the time taken by the model to determine the first correct move of the winning sequence. The data for these experiments was collected using the latedays cluster. We confirm the same speedups from experiments on Xeon Phi but these have higher overall execution times, so we present our findings from our runs on latedays only.

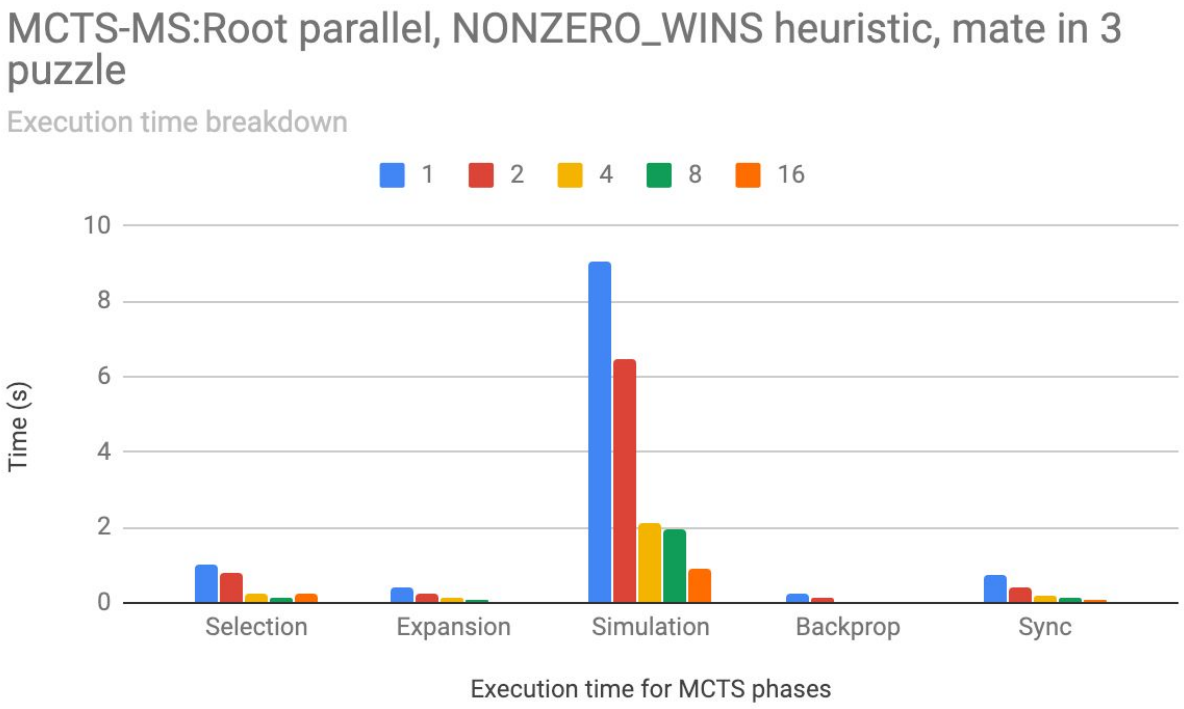
Root parallel, NONZERO_WINS, mate in 3 puzzle

The following results are for puzzle #2, mentioned in our Workload section. The game tree has 3*2=6 levels. We process 1,000,000 MCTS iterations across all threads. Note that our algorithm successfully returns the correct move 80% of the time for higher thread counts.

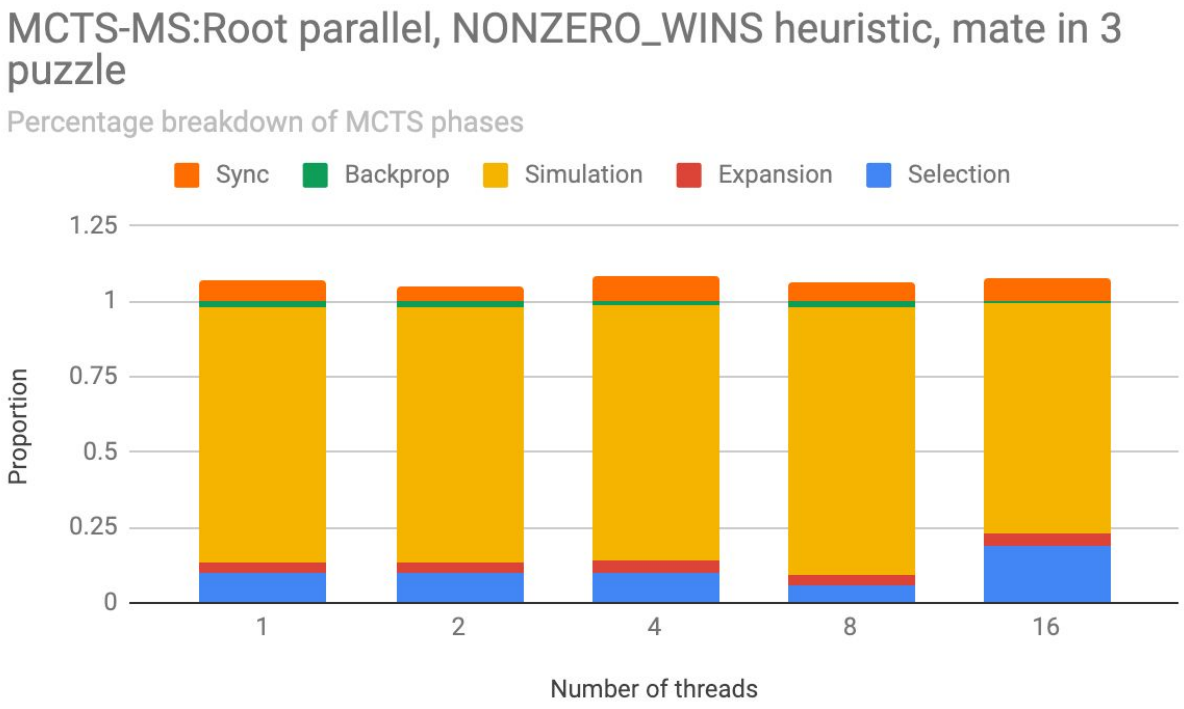




We observe a speedup of 1.4x for 2 threads, 4.3x for 4 threads, 4.9x for 8 threads and 9.1x for 16 threads (above two figures). We hypothesize that the less than ideal speedup is (i). Sensitive and affected by the randomness of tree search (we try to minimize this sensitivity in speedup measurement by taking the median execution time of several runs) (ii). Affected by cancellation overhead of minimax searches as soon as a proven move is found at the root.



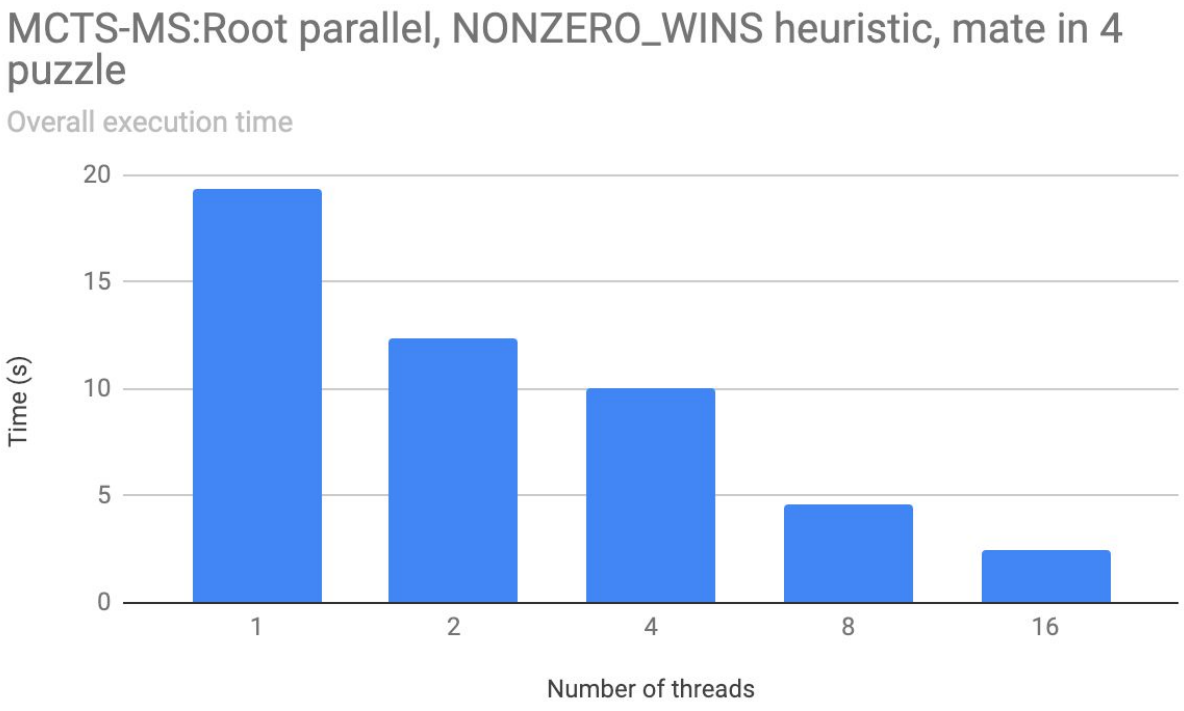
The execution time is dominated by the simulation phase. Minimax evaluations aren't expensive as the searches only span 2 moves. Most of the time is spent by threads executing random actions to perform simulation rollouts (above figure) (averaged over threads). Here **sync time** is the time spent performing atomic reads/writes.

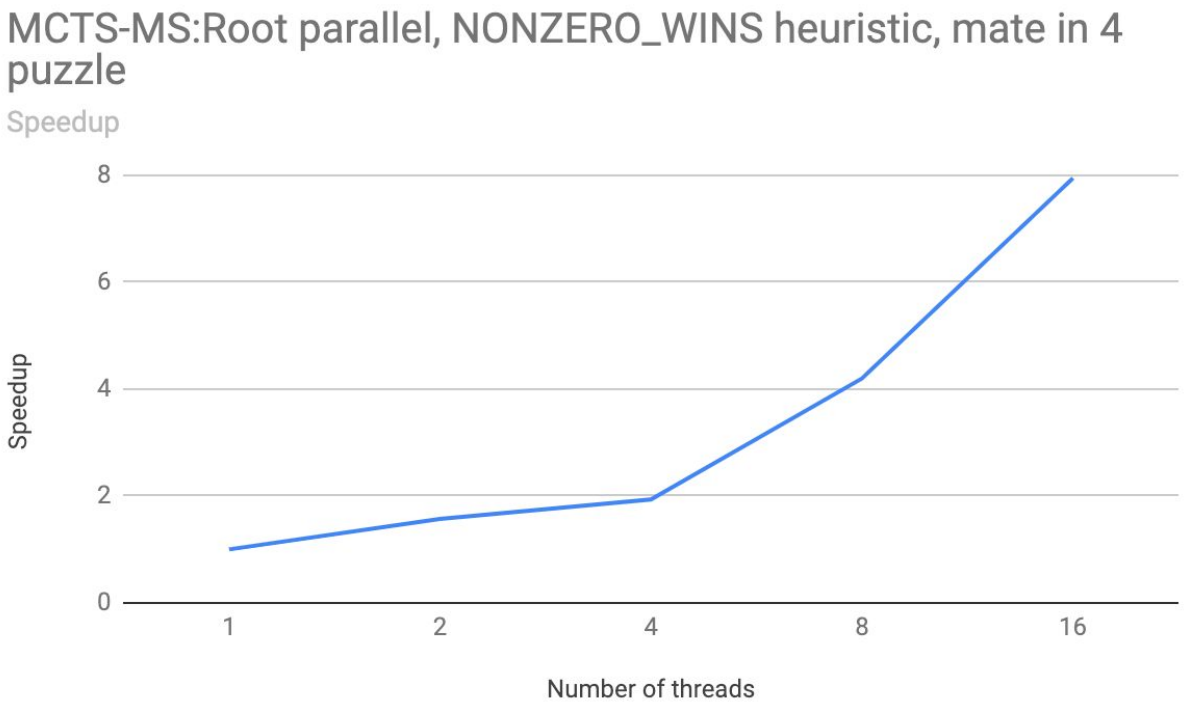


In our phase-wise percentage breakdown plot (averaged over threads), we choose to plot the atomic operations bar over 100% line as these atomic operations are already included in the four phases (above figure). The only atomic operations are (i). Reading and setting the `found_proven_move` flag, (ii). Setting the `proven_move` struct. After simulation, selection takes up the most time in its minimax searches. Atomic operations do not result in contentions for puzzles where there exists a single first optimal move for the winning sequence. Only one thread would ever write to the shared variables in this case (other threads will only try to read the `found_proven_move` flag and perform no atomic writes).

Root parallel, NONZERO_WINS, mate in 4 puzzle

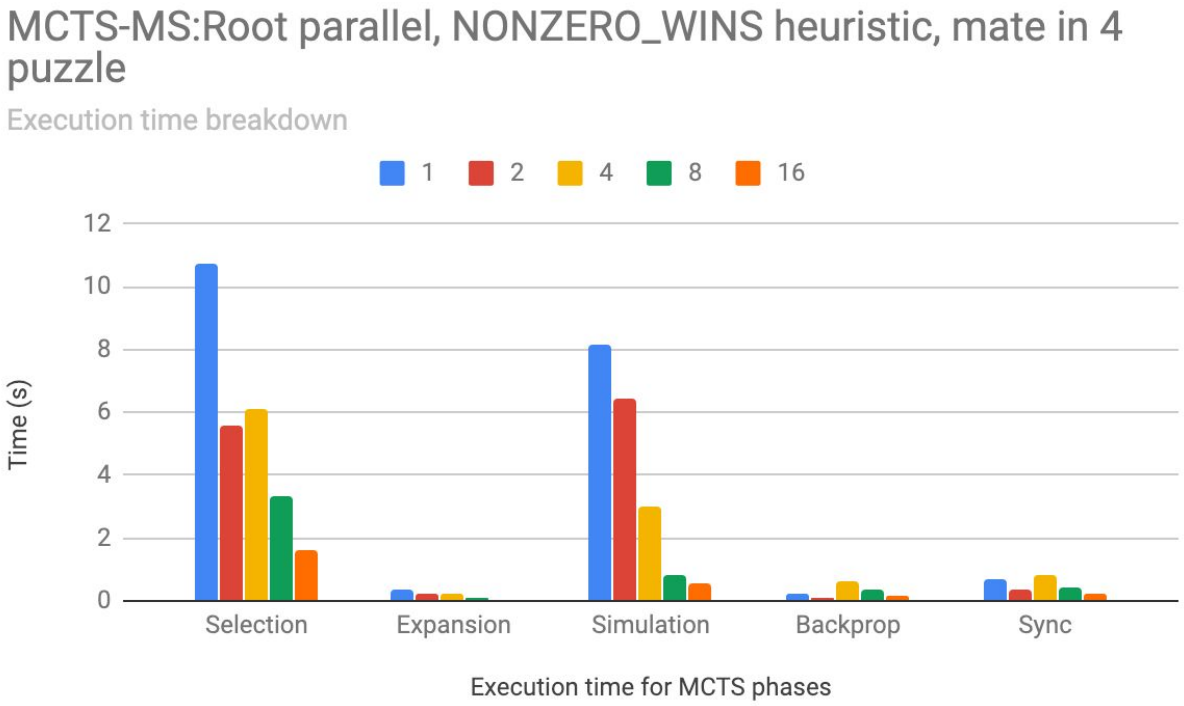
The following results are for puzzle #3, mentioned in our Workload section. The game tree has 4*2=8 levels. We process 1,000,000 MCTS iterations across all threads. Note that our algorithm successfully returns the correct move 80% of the time for higher thread counts.



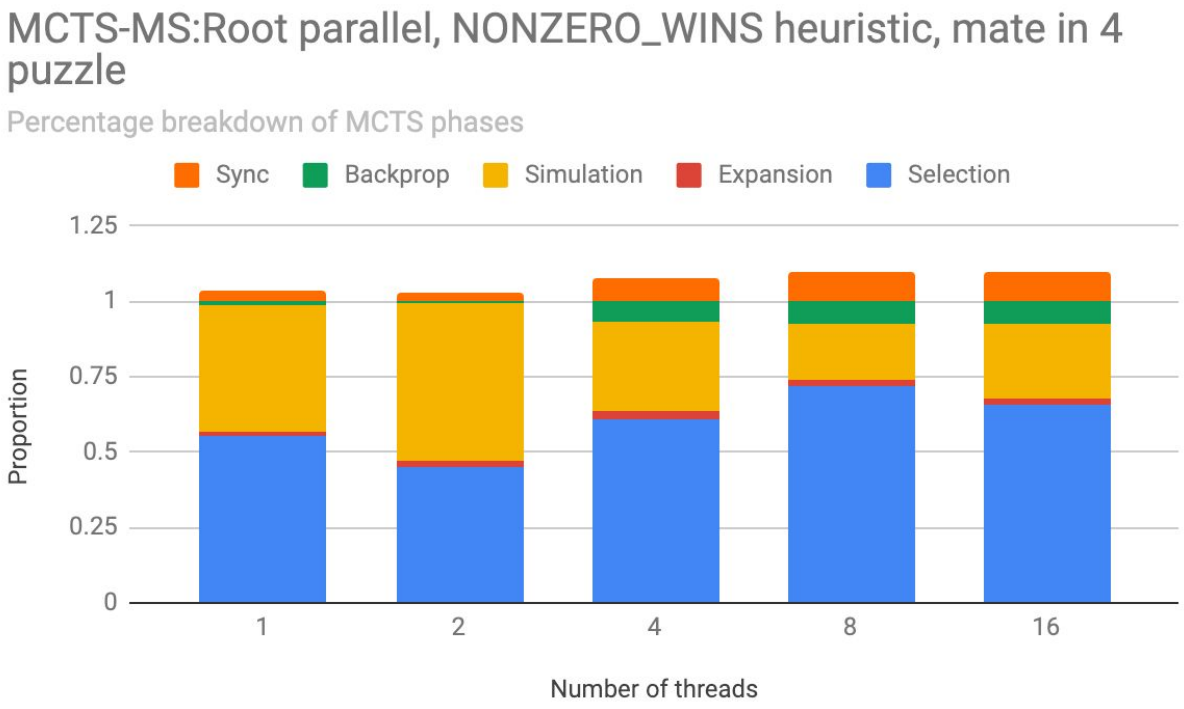


We observe a speedup of 1.6x for 2 threads, 1.9x for 4 threads, 4.2x for 8 threads, 7.9x for 16 threads (above two figures). As mentioned earlier, we hypothesize that the less than ideal speedup is (i). Sensitive and affected by the randomness of tree search (we try to minimize this

sensitivity in speedup measurement by taking the median execution time of several runs) (ii). Affected by cancellation overhead of minimax searches as soon as a proven move is found at the root.



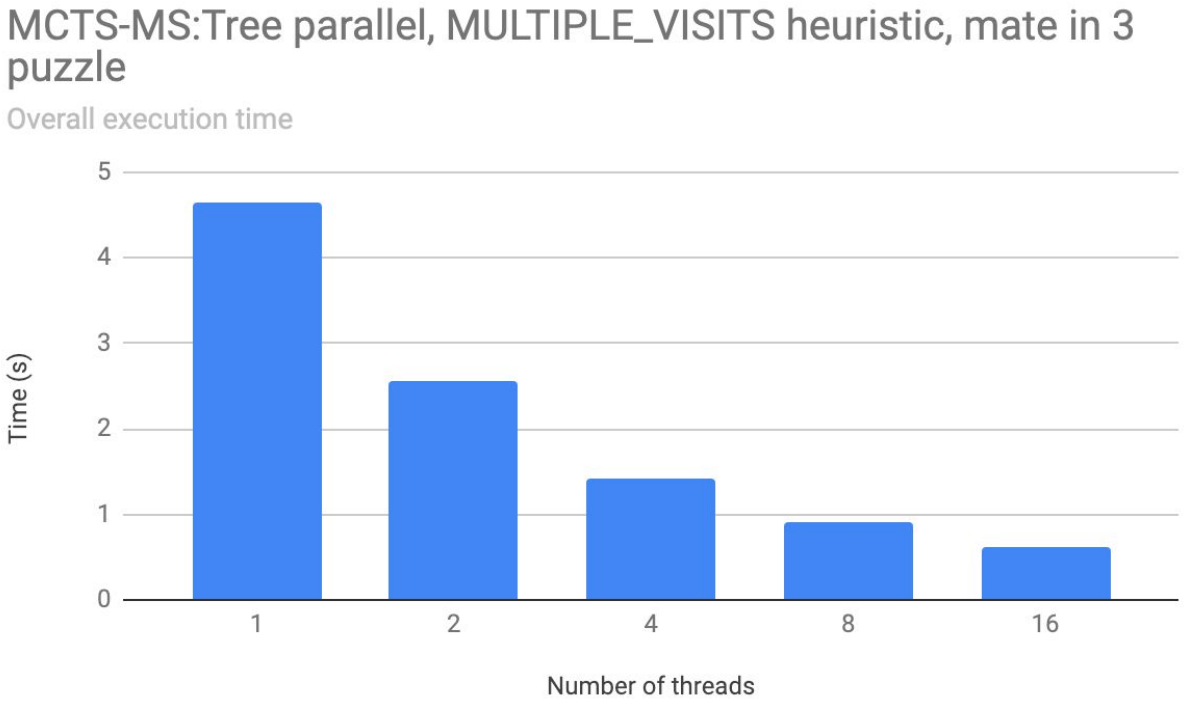
In our phase-wise execution time breakdown (averaged over threads), we observe that this time the selection time becomes more significant (above figure). This is in line with our expectations of the computational cost increase of minimax searches of 3 moves from 2 (previous experiment). Here **sync time** is the time spent performing atomic reads/writes.

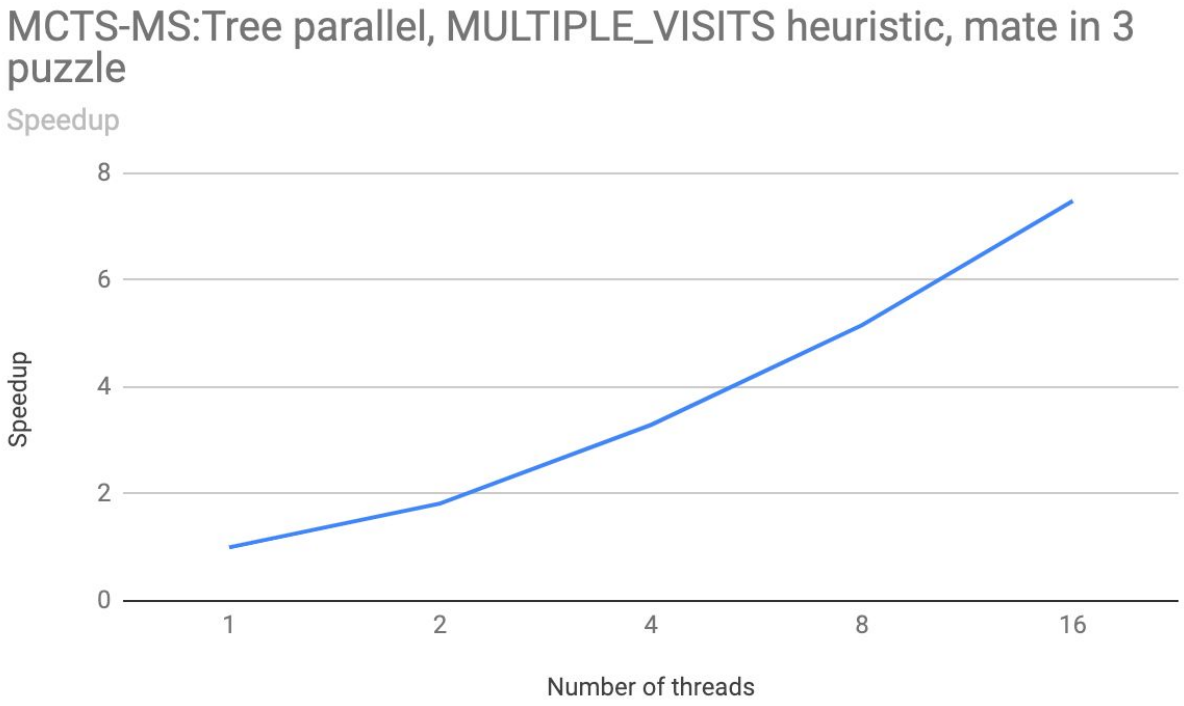


Again, in our phase-wise percentage breakdown plot (averaged over threads), we choose to plot the atomic operations bar over 100% line as these atomic operations are already included in the four phases (above figure). The atomic operations do not take up a significant portion for the same reasons discussed in the previous experiment's results section. Only one thread will ever need to write to the shared variables for our puzzle.

Tree parallel, MULTIPLE_VISITS, mate in 3 puzzle

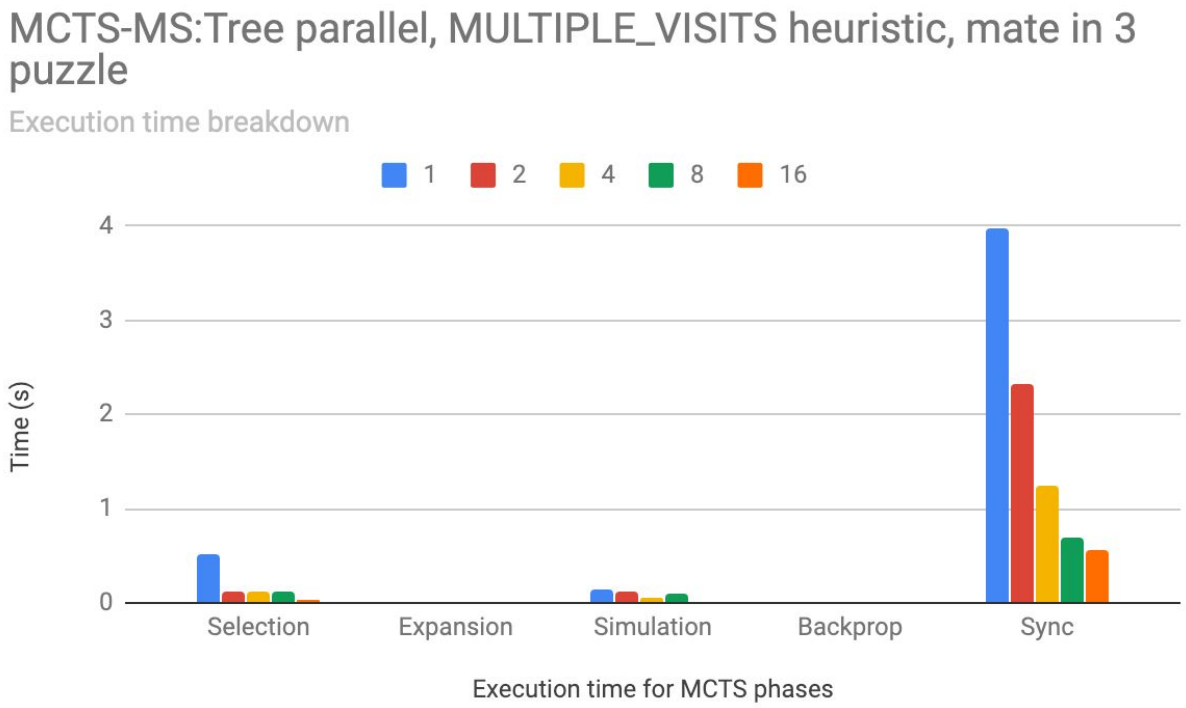
The following results are for puzzle #2, mentioned in our Workload section. The game tree has 3*2=6 levels. We process 4,000,000 MCTS iterations across all threads and use an OCCUPANCY_LOSS of -0.1, and use a VISIT_THRESHOLD of 10. Note that our algorithm successfully returns the correct move 70% of the time for higher thread counts.





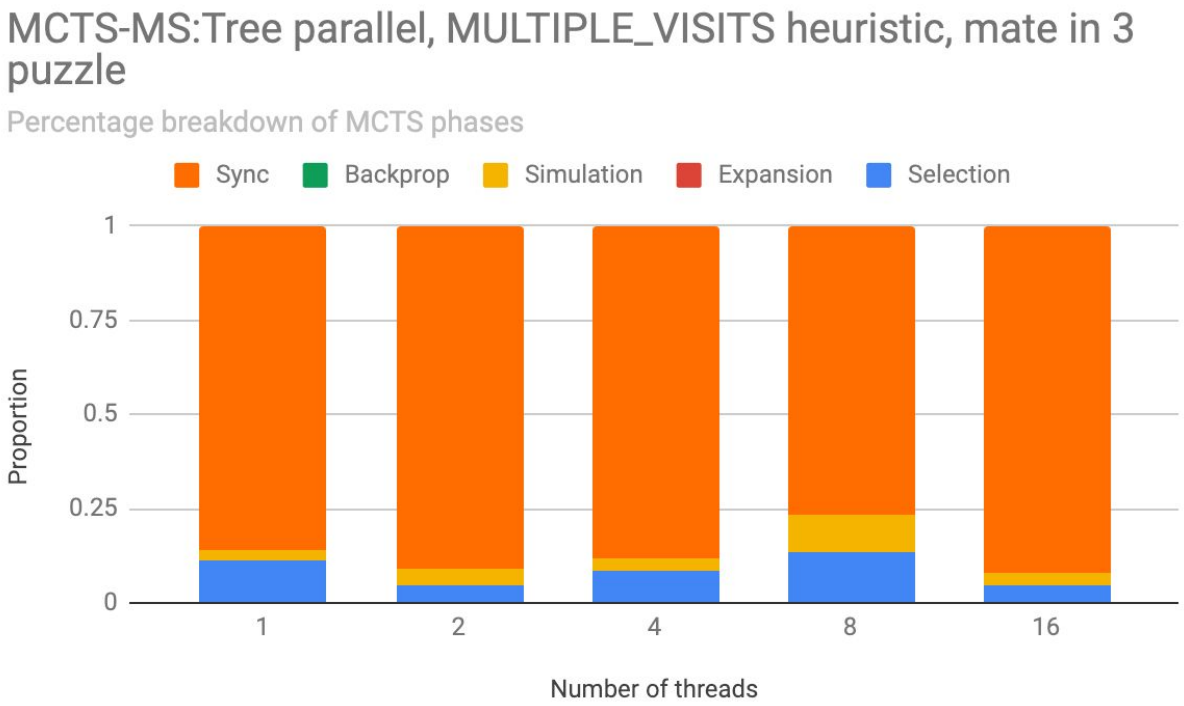
We observe a speedup of 1.8x for 2 threads, 3.3x for 4 threads, 5.2x for 8 threads, 7.5x for 16 threads (above figures). We hypothesize that the less than ideal speedup is (i). Sensitive and affected by the randomness of tree search (we try to minimize this sensitivity in speedup

measurement by taking the median execution time of several runs) (ii). Affected by cancellation overhead of minimax searches as soon as a proven move is found at the root and (iii). Overhead of using lock variables (note that i and ii are reasons we think affect root parallel as well).



In our phase-wise execution time breakdown (averaged over threads), we notice a significant jump in the sync time from our previous experiments with root parallel. **Sync time** is the time taken for a thread to initialize, destroy locks as well as wait until another thread releases a lock that it is trying to acquire. From our experiments we observe that the sync time is dominated by the overhead of initializing and destroying lock variables with `omp_init_lock` and `omp_destroy_lock` as opposed to waiting until another thread releases a lock. We hypothesize that this is the case because of
i. The tree size being extremely large (~1.9m) and
ii. The occupancy loss we induce upon node traversal.
The combined effect of these would make it unlikely for threads to process the same node simultaneously.

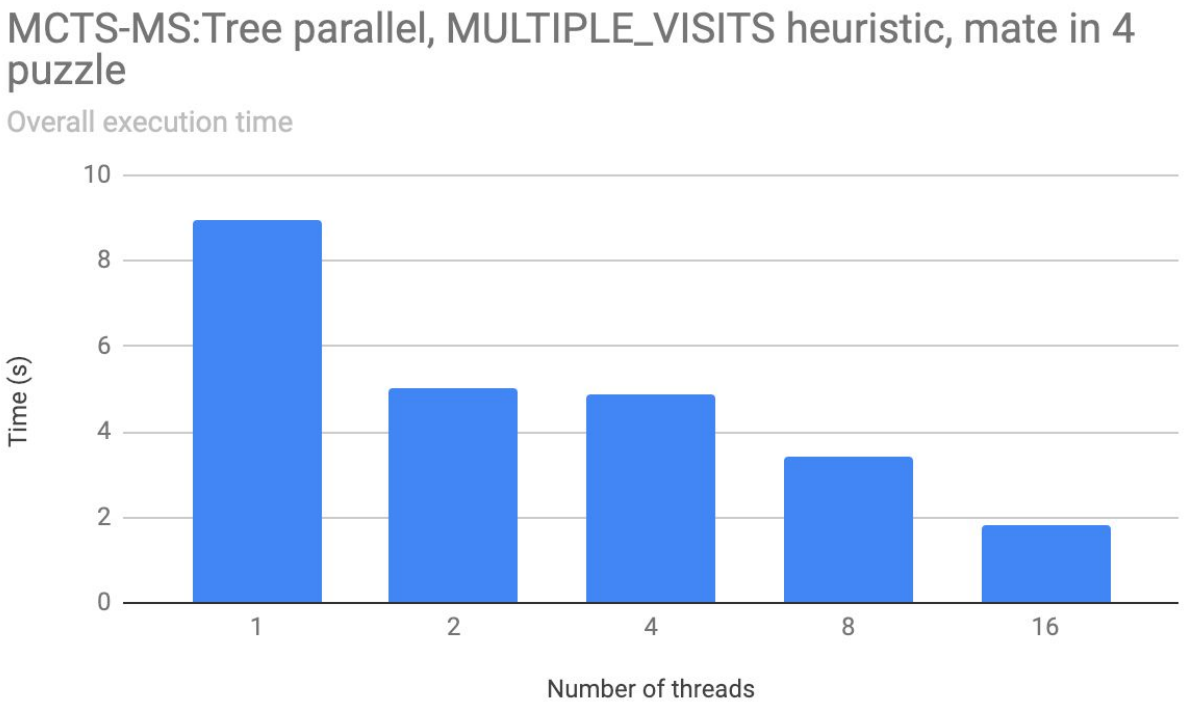
Another observation is that the lock overhead time scales well as the number of threads is increased. This is because on average, each thread would contribute to tree growth equally, measured in the number of nodes added. Since we have one lock variable per node, if there are M iterations and N threads, at the end of all iterations each thread will have created M/N locks. This explains the scaling effect we see in the sync times.

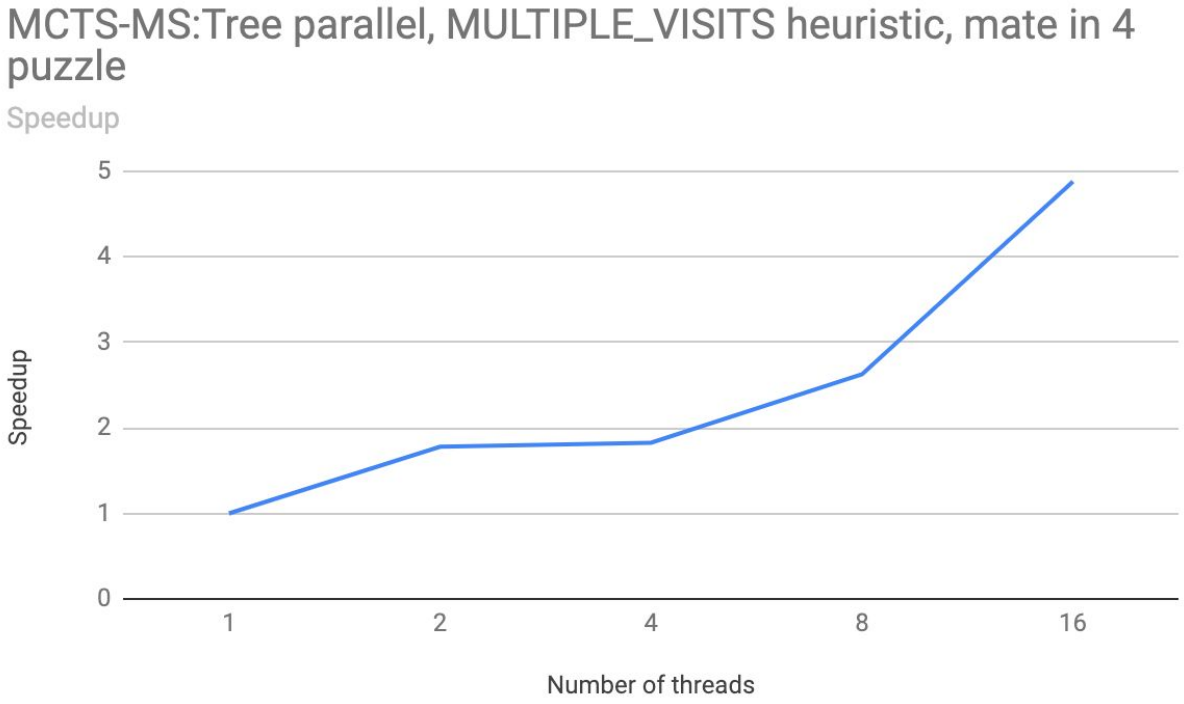


In our phase-wise percentage breakdown of the phases (averaged over threads), we see that sync occupies a significant portion. This is because 2 move minimax searches are inexpensive and the selection phase therefore does not require a lot of time. Another notable point is that the proportion of the phases remains relatively the same across the number of threads, this is because the work division is even among threads in this scheme.

Tree parallel, MULTIPLE_VISITS, mate in 4 puzzle

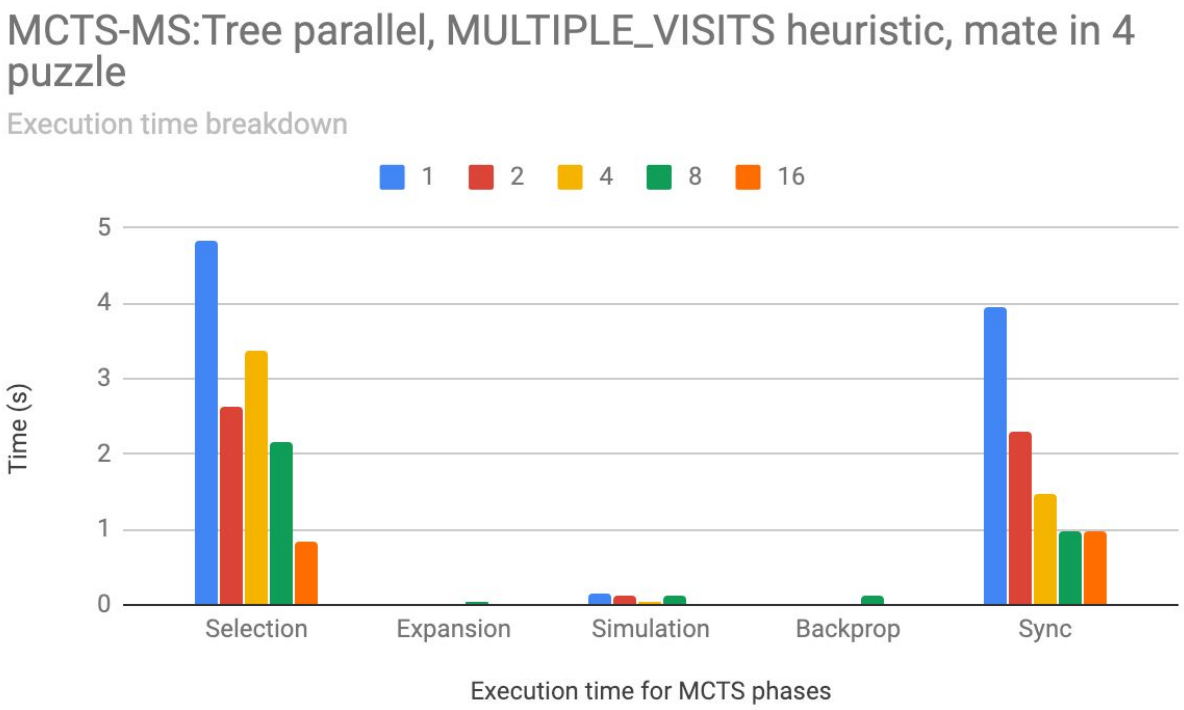
The following results are for puzzle #3, mentioned in our Workload section. The game tree has 4*2=8 levels. We process 4,000,000 MCTS iterations across all threads and use an OCCUPANCY_LOSS of -0.1, and a VISIT_THRESHOLD of 10. Note that our algorithm successfully returns the correct move 70% of the time for higher thread counts.



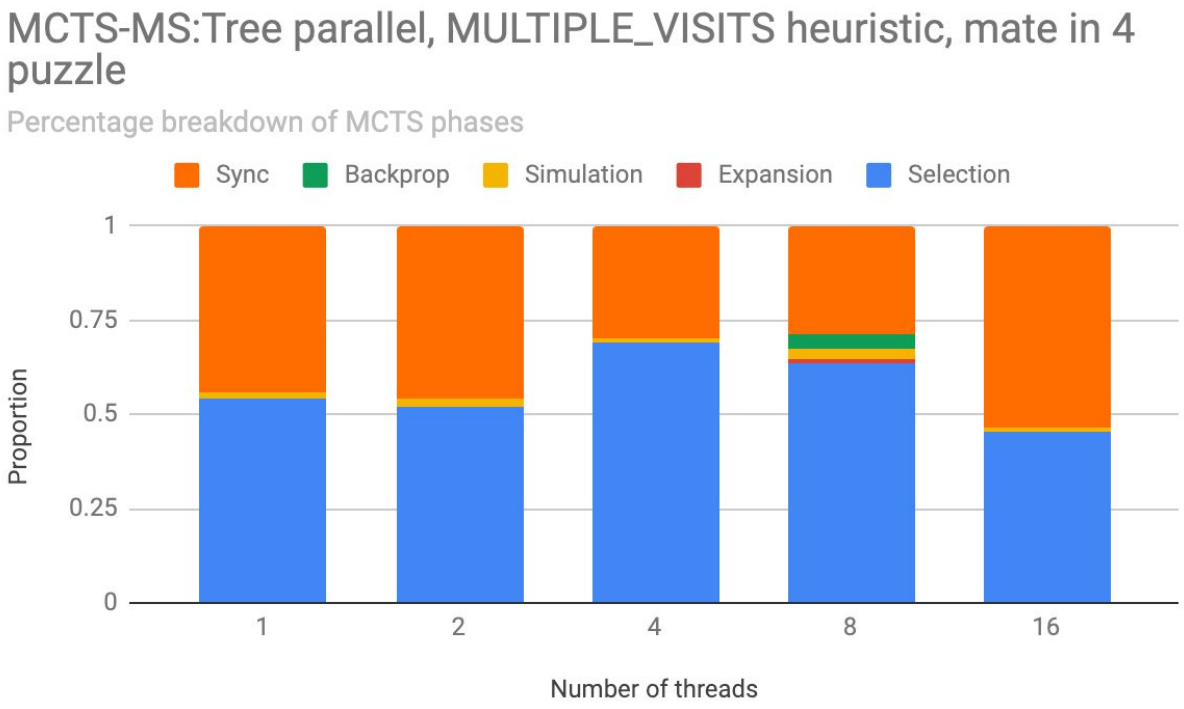


We observe a speedup of 1.8x for 2 threads, 1.8x for 4 threads, 2.6x for 8 threads, 4.9x for 16 threads (above figures). As mentioned in our mate in 3 analysis in the previous subsection, we

hypothesize that the less than ideal speedup is (i). Sensitive and affected by the randomness of tree search (we try to minimize this sensitivity in speedup measurement by taking the median execution time of several runs) (ii). Affected by cancellation overhead of minimax searches as soon as a proven move is found at the root and (iii). Overhead of using lock variables (note that i and ii are reasons we think affect root parallel as well).



Selection times are more significant for this workload as 3 move minimax searches are move expensive. This is similar to what was observed in the mate in 4 root parallel solver. The sync times were already explained and analyzed in the previous subsection.



We see in the percentage breakdown that the selection times are typically over 50% across all thread counts.

Parallel minimax plots

For the CUDA implementation of the parallel minimax, we tested the implementation on inputs with a depth of 3 and a depth of 4 (mate in 3 and mate in 4 respectively). The measured time is the amount of time it takes for the algorithm to return the best move for the provided State in a single occurrence. These values are compared against a single-threaded CPU implementation of the depth first search minimax algorithm.

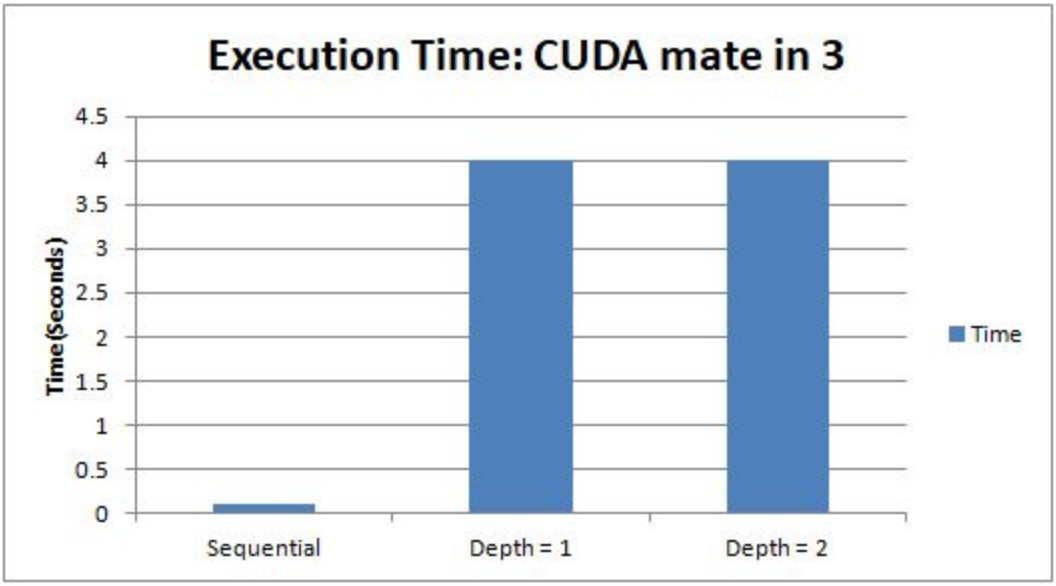


Figure: Execution time to return the best move for depth first search minimax at varying trigger depths for a mate in 3 problem

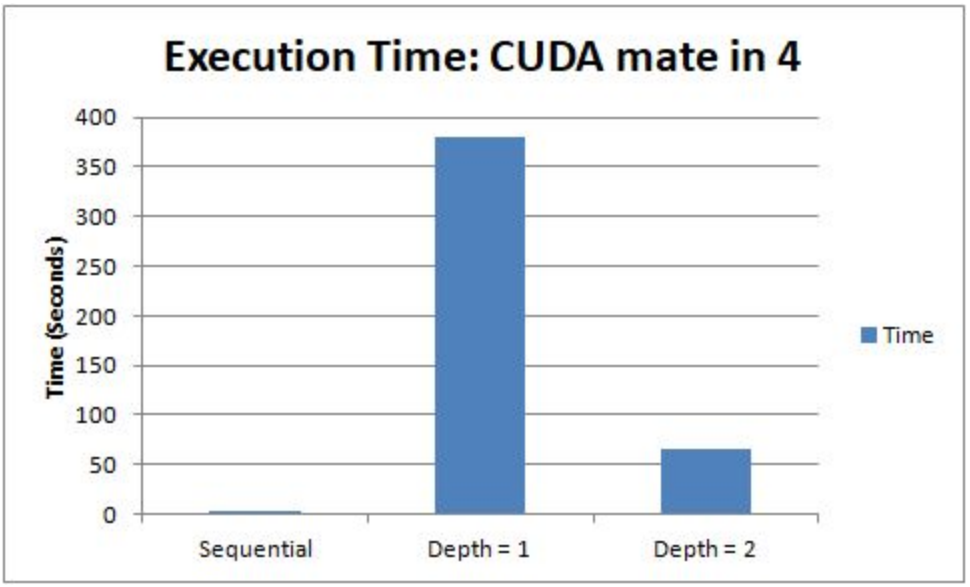


Figure: Execution time to return the best move for depth first search minimax at varying trigger depths for a mate in 4 problem

As shown in the timing plots above, the current CUDA implementation does not take advantage of enough parallelism to outweigh the overhead costs of launching a kernel every time the depth first search in the sequential version of the code reaches the target depth we set. Since at most each position that is passed into the algorithm has ~35 moves per state, that means that only ~35 threads will do meaningful work when the kernel is deployed. This limited occupancy does not offset the overhead cost of launching the kernel nor the fact that CUDA cores are simpler and would take longer to compute the same recursive path in a single thread. If the branching factor were more extreme, the occupancy of the GPU would increase and thus increase potential gains from parallelizing over CUDA.

The reason there is a exponential increase in execution times between mate in 3 and mate in 4 problems with the same trigger depth is that the since the depth increased, there are more cases when the CUDA target depth is reached due to the branching factor (120x slowdown for lowest level in mate in 4, 40x slowdown in mate in 3).

When the timing function is placed around the CUDA kernel launching plus overhead, it ticks up by a second every 10 kernel launches. This implies that the cost of launching multiple kernels on the GPU using the depth first search methodology leads to a time cost that scales with the number of times the kernel is called. The mate in 4 timing results for CUDA shown above seem to agree: when the trigger depth is increased (more of the depth first search is calculated on the GPU), the time to completion is reduced by a factor of 6x. This is still much greater than the sequential execution time of 3 seconds, a 20x slowdown due to the overhead.

Unfortunately, if the depth value is increased to try and reduce the number of kernel launches and thus the amount of overhead, the GPU returns with a memory error. We surmise this is due to the stack running out of room due to the increased amount of recursion calls since the problem was previously ameliorated by moving data structures that were on the stack onto the heap.

Based on the above results, the choice of GPU for such shallow trees is a bad one. The OpenMP branches of root and tree parallel showed greater promise of accelerating the MCTS-minimax hybrids.

CONCLUSION

We investigated the MCTS-MS hybrid model with different trigger heuristics as well as parallel schemes to solve mate in n chess puzzles. While root parallel approaches enjoy better scaling in the speedup with the number of threads (almost 10x with 16 threads for mate in 3), we show in our experiments that tree parallel approaches are in general faster in the overall execution time (0.62 seconds for mate in 3 with 16 threads). A promising line of future work would be to experiment with shallow minimax searches (only trigger at deeper nodes) to accommodate larger workloads (mate in 5+). The challenge would be to tune parameters such as OCCUPANCY_LOSS, VISIT_THRESHOLD and experiment with trigger heuristics to ensure that the tree search stays safe of traps and a reasonable speedup is obtained.

For the leaf parallel application, the CUDA implementation of the depth first search algorithm has an incredibly significant cost that slows down the computation significantly. The shallow trees and the branching factor of only ~35 lead to a low amount of parallelism that can be exploited to accelerate a comparatively small amount of searching due to the shallow tree. The issue of the GPU throwing memory errors if the CUDA trigger depth is set to greater than 2 also significantly restricts the amount of search work that can be offloaded: there is a limit to how much of the depth first search can be placed onto the GPUs. A possible future path in this branch is to continue the development of the breadth first search method as mentioned in the approaches section.

REFERENCES

[1] MCTS-Minimax hybrids:
https://dke.maastrichtuniversity.nl/m.winands/documents/mcts-minimax_hybrids_final.pdf

[2] Parallel MCTS:
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.159.4373&rep=rep1&type=pdf

[3] Parallel Minimax: http://olab.is.s.u-tokyo.ac.jp/~kamil.rocki/rocki_ppam09.pdf

[4] MCTS survey: http://mcts.ai/pubs/mcts-survey-master.pdf

[5] MCTS chess: https://www.ke.tu-darmstadt.de/lehre/arbeiten/bachelor/2012/Arenz_Oleg.pdf

[6] Auerswald collection: http://gorgonian.weebly.com/pgn.html

[7] Levinovitz, Alan (12 May 2014). "The Mystery of Go, the Ancient Game That Computers Still Can't Win". Wired. Retrieved 2014-06-02. The rate at which possible positions increase is directly related to a game's "branching factor," or the average number of moves available on any given turn. Chess's branching factor is 35. Go's is 250. Games with high branching factors make classic search algorithms like minimax extremely costly."

[8] Reference C++ chess implementation: https://github.com/tobijk/simple-chess

[9] Reference MCTS implementation: https://github.com/memo/ofxMSAmcts

DISTRIBUTION OF WORK

Sequential:
- MCTS-minimax-hybridization: Sai
- FEN loader: Peter

Root parallel (OMP): Sai
Tree parallel (OMP): Sai
Leaf parallel (CUDA): Peter

Credit distribution: 60-40 (Sai-Peter)