Parallelization of Information Set Monte Carlo Tree Search

Nick Sephton*, Peter I. Cowling*, Edward Powley†, and Daniel Whitehouse*
*York Centre for Complex Systems Analysis, Department of Computer Science, University of York, United Kingdom
Email: njs523@york.ac.uk, peter.cowling@york.ac.uk, dw830@york.ac.uk
†Orange Helicopter Games, York, United Kingdom.
Email: ed@orangehelicopter.com

***Abstract*—Process parallelization is more important than ever, as most modern hardware contains multiple processors and advanced multi-threading capability. This paper presents an analysis of the parallel behaviour of Information Set Monte Carlo Tree Search and the Upper Confidence Bounds for Trees (UCT) variant of MCTS, and certain parallelization techniques (specifically Tree Parallelization) have different effects upon ISMCTS and Plain UCT. The paper presents a study of the relative effectiveness of different types of parallelization, including Root, Tree, Tree with Virtual Loss, and Leaf.**

I. Introduction

Since 2006, Monte Carlo Tree Search (MCTS) [1], [2], [3] has proven a very strong technique game artificial intelligence. It has seen success in many games, most notably Go, which have previously proven significantly challenging for classic AI techniques. MCTS is a tree building technique which uses (normally random) simulations of games to estimate the value of certain decisions in a decision space. Since its creation, many enhancements have been proposed which modify its operation [4].

The vast majority of modern computers, games consoles and even mobile devices have multi-core processors. Algorithms using multiple parallel threads of execution are required to use these processors to their full potential. MCTS is readily adapted to parallel execution, with several methods having been proposed [5], [6].

MCTS has traditionally been applied to games of *perfect information*: that is, games where the full state is observable to all players at all times and moves are deterministic, non-simultaneous and visible to all players. More recent work has applied MCTS to games of *imperfect information*. Generally this means games with *information asymmetry*, i.e. games where parts of the state are hidden and different parts are hidden from different players. The class of imperfect information games also includes those with chance events, simultaneous moves or partially observable moves. This paper focusses on *Information Set MCTS (ISMCTS)* [7], [8]. ISMCTS works similarly to regular MCTS, but each simulated playout of the game uses a different *determinization* (a state, sampled at random, which is consistent with the observed game state and hence could conceivably be the actual state of the game).

Previous work on ISMCTS has focussed solely on the single-threaded version of the algorithm. This paper adapts parallelization techniques for standard (perfect information) MCTS to ISMCTS. Some parallelization techniques involve multiple threads searching the same tree, in which case it is necessary to use synchronisation mechanisms such as locks/mutexes to ensure multiple threads do not update the same part of the tree simultaneously. If threads spend most of their time waiting for mutexes to be unlocked, the efficiency of the algorithm is diminished. Games of imperfect information tend to have a larger branching factor than games of perfect information. Furthermore, the determinizations in ISMCTS restrict each iteration to a different sub-tree of the overall search tree, reducing the likelihood that two threads will attempt to take the same branch simultaneously. From this we hypothesize that threads in parallel ISMCTS will spend different amounts of time waiting on mutexes than in the perfect information case, and the relative efficiency of tree parallelization will be different.

Our results indicate that tree parallelization remains as efficient across both UCT and ISMCTS, and clearly highlights root as the technique of choice in terms of efficiency. It also suggests that adding virtual losses for this particular game is not an effective technique. Our results also speak clearly of the inefficiency of leaf parallelization, particularly with respect to parallelization across a large number of agents. Our parallelization schemes are explored in section IV.

The work here focuses specifically on the efficiency of the parallelization techniques (i.e. the number of iterations within a given time budget), which means that optimality of decision is not considered here.

The remainder of this paper is organised as follows. In Section 2, we present related works on MCTS and parallelization which are relevant to this study. Section 3 discusses the game we have chosen to use for this study; Lords of War by Black Box Games. In Section 4, we outline the solution methods which were used during this study. Section 5 contains a discussion of our results. Finally Section 6 presents a few conclusions, and outlines some possible future work which may be performed following this study.

II. Related Work

A. Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search is an adaptation of the standard tree search methodologies seen in more traditional mini-

Algorithm 1 Basic MCTS Process Summary

```
function TREE_SEARCH(s_0)
    v_0 = new TreeNode(s_0)
    while t_i < t_max do
        v_1 ← TREE_POLICY(v_0)
        r_1 ← DEFAULT_POLICY(v_1.s)
        BACKUP(v_1, r_1)
        return BEST_CHILD(v_0).a
```

max/expectimax AI, and also includes decision sampling to increase the generality of the tree search and remove the requirement for heuristic knowledge (although many strong implementations still employ such knowledge to strengthen the search). By taking samples of the decision space and using the results to guide the construction of a search tree, it is possible to find effective asymptotically optimal decisions for that decision space.

MCTS was first invented in 2006 [1], [2], [3], and has sparked a great deal of further research and experimentation since that time. MCTS has seen much success in the field of Go [9], which proves challenging for more traditional AI techniques such as minimax search.

MCTS operates by building a game tree step-by-step, and running a single playout from a leaf state at each iteration. From these game playouts, a reward signal is received from the terminal game state, and the information is propagated upwards back through each parent node, modifying that node's value as it does so. The iterative growth of the tree is non-symmetrical, and controlled by a tree policy which attempts to balance exploitation against exploration by selecting potentially high reward nodes.

The basic MCTS algorithm is made up of 4 steps, and works as follows:

- Selection: The algorithm moves down through the tree using the *tree policy* until it reaches a node which has unexpanded children or a terminal node.
- Expansion: If the selected node has unexpanded child nodes, then one (or more) of those nodes are added to the tree.
- Simulation: A simulation is run from each of the new child nodes using the *default policy*, normally to a terminal state.
- Back-propagation: The simulation result is backed up through the parent nodes of the selected node, updating statistics until it reaches the root node.

B. Upper Confidence Bound applied to Trees (UCT)

Upper Confidence Bound applied to Trees refers to the use of MCTS with a random default policy, but using a specific tree policy known as *UCB1*. UCB1 treats the choice of a child node as a multi-armed bandit problem [3], [10], and selects the child node that has the best expected reward as approximated by Monte Carlo simulations.

During tree policy operation, the UCB1 equation (see equation 1) is used to evaluate each child node to determine which node should be selected for expansion and simulation. The terms used in the equation are as follows: $\bar{X}_i$ is the average reward from child node i (the node currently being evaluated), C is the *Exploration Constant*, n is the total number of visits to the parent node, and n_i is the total number of visits to the child node i.

$$UCB1 = \bar{X}_i + C\sqrt{\frac{2\ln n}{n_i}} \tag{1}$$

The UCB1 equation provides a balance between exploration and exploitation by scaling the number of visits to a given node against the rewards from that node's children. The term UCT is commonly used to describe using MCTS with the UCB1 algorithm and a default policy of random selection.

Kocsis and Szepesvári [3], [10] showed optimality of UCT given that when provided with enough budget, UCT allows MCTS to converge to an optimal decision, even in cases when the budget is only sufficient to search a portion of the tree.

C. Information Set Monte Carlo Tree Search

Information Set Monte Carlo Tree Search (ISMCTS) [8], [11] is an enhancement to MCTS for making decisions in games of imperfect information. ISMCTS effectively negates an established weakness of MCTS known as *Strategy Fusion* [12].

An information set is a collection of game states that are identical to the true state of the game from the perspective of the observing player. By collecting many similar states into sets, the game tree is vastly simplified. For example, in a card game where an opponent has a hidden hand of cards, the player's information set would be every game state which corresponds to all combinations of the opponent's hidden cards.

ISMCTS operates in a similar manner to vanilla MCTS. It uses determinized games during simulation, but does not eliminate hidden information. Rather than determinizing the game state only once, a random determinization is used for each simulation, effectively creating a search across a large number of possible combinations of hidden information. ISMCTS is currently implemented in the successful commercial mobile game, Spades by AI Factory [13].

D. Parallelization Techniques

Related work in this area has established four methods of parallelization, which are often referred to by different names. Cazenave et al. [5] suggest three different methods; Single-Run Parallelization, Multiple-Run Parallelization and At-the-leaves Parallelization. These have been provided with different names in other literature, but our preferred names come from Chaslot et al. [6], who named Single-Run Parallelization as *Root Parallelization*, reflecting the complete parallelization of the MCTS from the root, and renaming At-the-leaves Parallelization as simply *Leaf Parallelization*. Chaslot et al. also provides another method named Tree Parallelization.

Root parallelization creates a tree for each thread and builds those trees independently. When the build process is

completed, the trees are amalgamated and their combined statistics are used to determine the optimal move from the current position. Multiple-run parallelization is similar, except after the tree statistics have been amalgamated, the tree is copied and sent back out to the others threads for further independent processing. This cycle can repeat multiple times until a budget is reached.

In leaf parallelization, a single tree is built by a master thread, and that thread spawns child threads to run all simulations.

Cazenave et al. indicate that the results of Root Parallelization are comparable to those from Multiple-Run Parallelization, and as the former is far easier to implement, it is preferable.

Chaslot et al. [6] report from their experimentation on Go that Leaf Parallelization seems a poor method, taking between 2-4 times less time to reach the same result as unparallelized MCTS when using 16 processors.

They acknowledged Root Parallelization as the stronger technique, but also stated that Tree Parallelization with Virtual Loss performs comparably on smaller Go boards. This is supported by Bourski et al. [14], however Mehat et al. [15] later contested this, stating that Tree Parallelization showed improved results, and that the improvement is related to the ability to keep all threads consistently busy.

The terms used by Chaslot et al will be used throughout the remainder of this paper.

III. Experimentation Game

The game chosen for experimentation was *Lords Of War* by publisher Black Box Games, a strategic card game that uses a board for card placement and thus considers the relative positions of cards. The objective is to eliminate twenty of your opponent's cards. In addition, some of the cards are designated *Command Cards*, and a victory is also attained if a player eliminates four of their opponent's Command Cards.

The game is played on a 7×6 square board, where each square can hold a single card. Each card has a number of attacks, each of which has an associated value, and are directed towards adjacent squares. When a card is being attacked with a total value greater than its defence value, the attacked card is eliminated and removed from the board. A limited selection of cards also have ranged attacks with can affect non-adjacent cards, but only if the card making the ranged attack is not under direct attack itself.

Players take turns to place a card, evaluate combat between all cards on the board, then choose either to return a surviving friendly card from the board to their hand, or to draw a new card from their deck. The complete rules and a number of tutorial videos are available on the Lords of War website[1].

In the mid-game, it is common for most moves between average players to result in a capture, so a game with average human players rarely goes beyond turn 50 (the point where each player has made 25 moves), and can finish much earlier if a player is careless with their Command Card placement. AI games typically took 50-60 turns, as there was no heuristic knowledge included in this instance to guide the searches towards optimality.

Experimentation with Lords of War has revealed that it has rich strategy, with mid-game states commonly having a branching factor in the range 25 - 50.

IV. Solution Methods

A. Game Engine

The experimental MCTS engine and Lords of War game were implemented in C++ and all experiments were run on a Intel(R) Xeon(R) CPU E5645, with two processes (2.40GHz & 2.39GHz), each with 6 cores & hyperthreading and 32GB of RAM.

A total of four different methods of parallelization were implemented (Root, Tree, Tree with Virtual Loss and Leaf). When appropriate to the style of parallelization, C++ 11 support for $mutex$[2], $future$[3] and $lock\_guard$[4] was used to lock nodes that were being processed. The only nodes that are locked are those selected by for the Expansion step of the MCTS process (see section II-A).

B. Root Parallelization

Root Parallelization [5] (also known as slow tree parallelization or Single-Run Parallelization) describes the process by which each system runs a separate MCTS from the same game state, then the results are amalgamated by a master process. As each system would have a different random seed, different results should be generated.

Root parallelization was implemented as shown in algorithm 2. A separate tree is built by each agent, and then the node statistics of the first level nodes are combined to determine the overall most visited node, and thus the decision to select.

C. Tree Parallelization

Tree Parallelization [6] was implemented as shown in algorithm 3. A single shared tree is maintained, and each agent works to add nodes to that tree and update statistics in the tree nodes. $Mutex$, $future$ and $lock\_guard$ are used to ensure that thread safety is maintained (i.e. no two agents attempted to write to the same memory at the same time, or read from memory that was being altered).

Tree Parallelization often uses a technique known as *Virtual Loss* in order to discourage selection of the same node by two different threads. While a node is locked, an additional loss is reported any time it is evaluated, in order to make that node appear less valuable for selection, and thus decrease the amount of time spent waiting for a node to unlock. Chaslot et al. [6] attributes the use of virtual losses to Coulomb through a personal communication.

[1] http://www.lords-of-war.com/

[2] http://en.cppreference.com/w/cpp/thread/mutex

[3] http://en.cppreference.com/w/cpp/thread/future

[4] http://en.cppreference.com/w/cpp/thread/lock_guard

Algorithm 2 Root Parallelization

```
function DoRootParallelization(nAgents)
    treeList = list(MCTS_Tree)
    agentList = list(MCTS_Agent)
    threadList = list(Thread)

    for nAgents do
        agentList ← newAgent
        treeList ← newTree
        newThread(newAgent.Run, newTree)
        threadList ← newThread

    for threadList do
        thread.Join()

    statistics = list(MCTS_Statistics)

    for treeList do
        statistics ← tree.GetStats()
return statistics.GetBestMove()
```

Algorithm 3 Tree Parallelization

```
function DoTreeParallelization(nAgents)
    agentList = list < MCTS_Agent > ()
    threadList = list < Thread > ()

    for nAgents do
        agentList ← newAgent
        threadList ← newThread(newAgent.Run(tree))

    for threadList do
        thread.Join()
return tree.GetBestMove()
```

The *Virtual Loss* modification described by Chaslot et al. [6] (and attributed to Coloumb [2]) was also implemented in a separate set of experiments. Before we begin a simulation on a node, in addition to locking its local $mutex$, we also add a loss to that node's statistics to reduce the chance of the node being selected by another thread.

D. Leaf Parallelization

Leaf parallelization is implemented as shown in algorithm 4. A single tree is maintained, a single "parent" agent is used to operate on that tree. Whenever a simulation run is required, the parent agent hands that simulation to a "child" agent which then runs independently. Child agents are checked to see if they are clear of an existing simulation before new child agents are created up to the limit by the number of agents.

E. Experimentation

As we are interested in the speed of decision-making and not the optimality of the decision that results, all experiments dealt with single decisions instead of complete games. The state that

Algorithm 4 Leaf Parallelization

```
function DoLeafParallelization(nAgents)
    agentList = list(MCTS_Agent)

    for nAgents do
        agentList ← newAgent

    primaryAgent.RunLeaf(agentList)
    statistics = list(MCTS_Statistics)

    for treeList do
        statistics ← tree.GetStats()
return statistics.GetBestMove()

function RunLeaf(nAgents)
    agentList = list < MCTS_Agent > ()
    for nAgent do
        agentList ← newAgent
    SimStartNode = RunWithoutSim(tree)
    while currentAgent.IsBusy() do
        currentAgent = GetNextAgent()
    currentAgent.RunSim(SimStartNode)
```

was used for most experimentation is that of the game after the first two "Issuing the challenge" moves described in the rulebook (essentially an initial setup for the game). Two cards are placed during the initial set up, both of which were the Orc General card (see figure 1). The initial set up position is displayed in figure 2, and is referred to as S_1 for the remainder of this paper.

During experimentation, the player decks were stacked so they would draw identical cards, and the order was maintained between tests, to ensure that all the examined decisions were identical.

The following series of experiments were then performed, each repeated 1000 times on Plain UCT and ISMCTS:

- Root Parallelization (between 1 and 8 threads)
- Tree Parallelization (between 1 and 8 threads)
- Tree Parallelization with Virtual Loss (between 1 and 8 threads)
- Leaf Parallelization (between 1 and 8 threads)

During these experiments, the Plain UCT was running on the perfect information game (i.e. all hidden information was made visible), and the ISMCTS agent was playing the imperfect information game and determinizing on each iteration. All experiments were run with 5000 MCTS iterations, and an exploration constant of 1.4. In cases when parallelization was used, the MCTS iterations were split across different agents, with each agent receiving a static 5000/n iterations to perform.

It was expected that Root Parallelization with one agent would perform identically to UCT with no parallelization, as only one tree is created and there is no mutex locking during the process. Tree Parallelization with one agent was included to determine the effects of the mutex locking & unlocking on

Fig. 1. Orc General - Gonke Longtooth

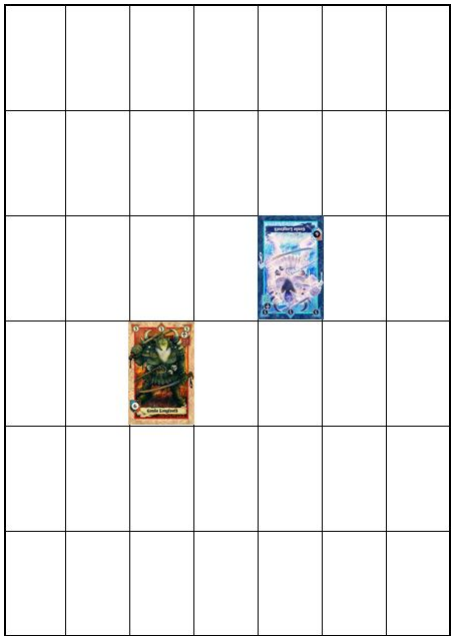


Fig. 2. Experimentation state S_1 with two Orc General cards.

the decision speed, as this should be the only factor that is different between the two processes.

V. Results

The results of the state S_1 experimentation are displayed in table I and II, and comparative graphs of these results can be seen in graphs (a) and (b) (please note the difference of scale between the two graphs).

There is a negligible difference between using Tree and Tree with VL in both MCTS and ISMCTS. As there is little overhead to adding or removing the virtual loss, then the main difference in speed would be seen when a virtual loss causes a different node to be selected by the selection policy. The fact that the results for both are nearly identical suggests this is rarely happening, and thus either the selection choice is very clear and a single loss is not affecting the choice, or that the choice is very unclear as the statistics are similar in most nodes at a given level, and nodes are effectively being chosen at random. Chaslot et al. [6] reported that Tree with Virtual Loss performs as well as Root Parallelization on smaller boards in the game of Go, but this does not seem to be the case with Lords of War.

Leaf parallelization is clearly a far slower technique than any other used here. Using more than 3 agents does not result in a speed increase. We see that agents numbered above 3 are almost never used: the simulations assigned to earlier agents are already complete by the time a simulation would be assigned to an agent numbered 3 or higher.

Tree Parallelization shows itself to be a competitive technique in terms of speed, but still a lot slower than Root Parallelization in both MCTS and ISMCTS. If we calculate the difference in speed between Tree and Root in MCTS, then the difference in speed between Tree and Root in ISMCTS, it can be seen that the difference is comparatively lessened in ISMCTS, but that the speed decrease caused by ISMCTS is still more significant.

As discussed earlier, we can see the effects of using $mutex$es to lock nodes by comparing the difference in performance between root and tree parallelization when using 1 agent, however this only accounts for the actual cost of the locking procedure, not the expense caused by causing any threads to wait. The average of this difference is very small, the best estimate being less than 16ms (due to the resolution of the timer used). This indicates that the time spent locking $mutex$es is very low, and almost all of the expense comes from threads waiting to obtain lock on a $mutex$. We can see a similar difference between the ISMCTS runs of root and tree parallelization.

In order to see the relative effects of different parallelization techniques on UCT and MCTS, we can compare the relative efficiency of individual agents within each technique. Efficiency is calculated as $\frac{t_1}{n \cdot t_n}$, where t_n is the decision time for n agents. In particular, the efficiency for $n = 1$ is $\frac{t_1}{1 \cdot t_1} = 100\%$. The efficiency of each technique for $n > 1$ agents is shown in table III. In an ideal scheme with 100% efficiency, using n agents would result in an n-fold increase in speed: adding the

TABLE I
UCT PARALLELIZATION RESULTS (MS)

nAgents	1	2	3	4	5	6	7	8
Root	505.74	253.47	191.71	154.77	135.94	113.21	104.41	91.74
Tree	514.09	299.13	207.51	163.61	133.19	109.64	105.74	98.05
Tree (VL)	511.58	297.60	198.75	152.99	131.45	110.64	105.86	98.42
Leaf	1226.18	676.75	615.35	618.23	630.62	619.71	626.06	634.27

TABLE II
ISMCTS PARALLELIZATION RESULTS (MS)

nAgents	1	2	3	4	5	6	7	8
Root	1061.07	533.23	376.74	341.36	273.27	259.20	220.66	194.92
Tree	1057.69	768.74	528.83	432.33	348.83	291.88	282.98	259.58
Tree (VL)	1056.35	751.91	531.36	444.21	344.75	295.64	283.22	264.05
Leaf	1885.77	1087.50	981.75	1019.97	975.29	987.03	1049.48	1005.02

second agent would cause overall speed to double resulting in a decision time of $t_2 = \frac{t_1}{2}$, and so on.

We can see from graphs (c) and (d) that root spreads the load between agents most effectively, with one exception of note - the 2nd agent in leaf parallelization on MCTS.

VI. CONCLUSIONS & FUTURE WORK

A. Conclusions

It can be seen that in no combination of tested factors did ISMCTS outperform UCT in terms of efficiency. It is important to remember however that ISMCTS (unlike Plain UCT) is designed to handle games of imperfect information, which inevitably adds overheads to its operation, but also makes it suitable in situations where plain UCT is inapplicable or may struggle.

There does not appear to be a significant difference in the slow-down caused by tree parallelization between ISMCTS and UCT, which is contrary to our original hypothesis. This indicates that the slow-down is unlikely to be caused by threads awaiting locked resources, but more likely by code associated with the locking and unlocking process.

One possible explanation is that due to the shape of the trees, the locking of initial nodes is having a larger effect on an ISMCTS tree than a UCT tree. As mentioned previously, the expansion step of the MCTS process locks the node to be expanded. The UCT tree will only have a relatively small number of nodes at the first level (approximately 20-30), meaning that once these nodes are expanded, other agents are cleared to continue through the root node without locking. The ISMCTS tree will have a large number of nodes at the first level (approximately 150), which will cause a significant initial delay as multiple agents compete to lock the root node. This effect may continue on other promising nodes during the early stages of the tree building process.

Of all approaches attempted, root Parallelization across high numbers of agents was the most time efficient approach, although the speed increase became mostly negligible when adding more than 4-5 agents.

It's also worth noting that tree parallelization is less effective when applied to ISMCTS, which is somewhat surprising. Using tree parallelization in an environment where agents are unlikely to be blocked should increase efficiency, but the relative decrease in efficiency suggests that more blocking is occurring.

It's interesting that adding virtual loss has almost no effect on the effectiveness of tree parallelization in Lords of War. This may be due to the positioning of valuable states in the game tree - if there are few positions of high value in a tree, then adding a virtual loss is unlikely to dissuade from their further immediate exploration.

B. Future Work

During the parallelization experimentation, the MCTS iterations were split evenly and statically assigned to the working threads. If a thread had finished early, it simply ended and did no further work. If iterations could be assigned dynamically as threads became available, then the process could be more efficient.

The branching factor of the game (or state) under examination may be relevant to the effectiveness of tree parallelization, as a higher branching factor should result in less thread waiting time. Experimenting with games or states with different branching factors would be interesting follow up work.

Future work on Lords of War will consider playing strength directly, rather than via the proxy of number of iterations performed, although we imagine results will be similar.

C. Acknowledgements

The work displayed here was supported by EPSRC (http://www.epsrc.ac.uk/), the LSCITS program at the University of York (http://lscits.cs.bris.ac.uk/), and Stainless Games Ltd (http://www.stainlessgames.com/).

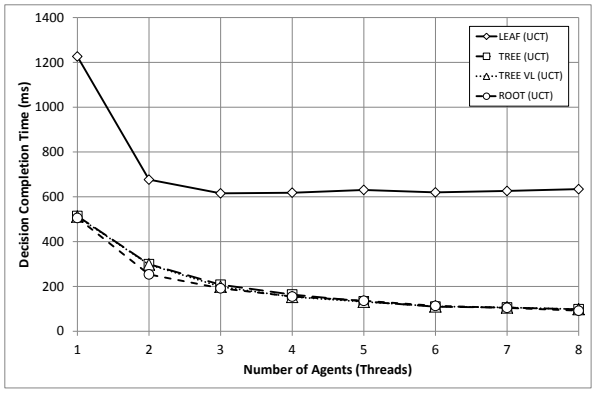
We thank Black Box Games for their support in working with their game Lords of War.

REFERENCES

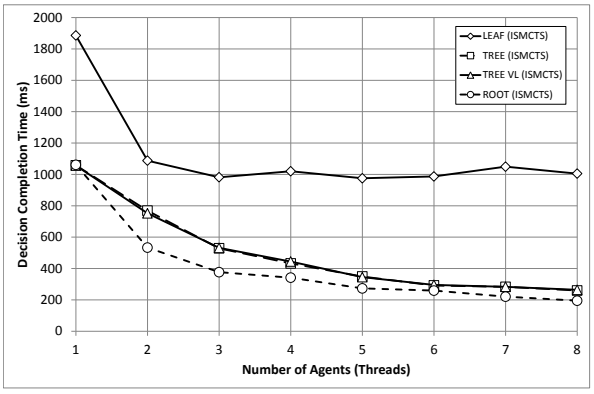
[1] G. M. J.-B. Chaslot, J.-T. Saito, B. Bouzy, J. W. H. M. Uiterwijk, and H. J. van den Herik, "Monte-Carlo Strategies for Computer Go," in *Proc. BeNeLux Conf. Artif. Intell.*, Namur, Belgium, 2006, pp. 83–91.

TABLE III
COMPARATIVE EFFICIENCY OF INDIVIDUAL AGENTS

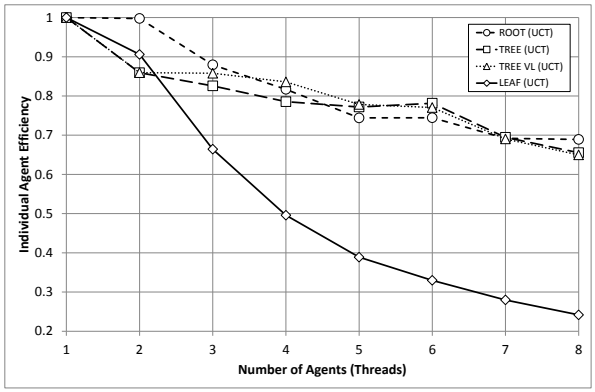
nAgents	2	3	4	5	6	7	8	
Root (UCT)	1.00	1.00	0.88	0.82	0.74	0.74	0.69	0.69
Tree (UCT)	1.00	0.86	0.83	0.79	0.77	0.78	0.69	0.66
Tree VL (UCT)	1.00	0.86	0.86	0.84	0.78	0.77	0.69	0.65
Leaf (UCT)	1.00	0.91	0.66	0.50	0.39	0.33	0.28	0.24
Root (ISMCTS)	1.00	0.99	0.94	0.78	0.78	0.68	0.69	0.68
Tree (ISMCTS)	1.00	0.69	0.67	0.61	0.61	0.60	0.53	0.51
Tree VL (ISMCTS)	1.00	0.70	0.66	0.59	0.61	0.60	0.53	0.50
Leaf (ISMCTS)	1.00	0.87	0.64	0.46	0.39	0.32	0.26	0.23



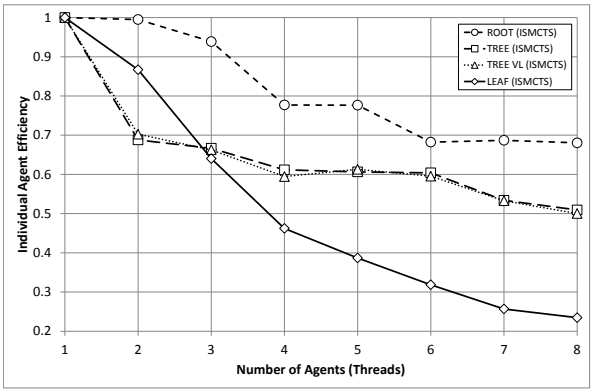
(a) UCT on state S_1



(b) ISMCTS on state S_1



(c) MCTS Parallelization efficiency by agent



(d) ISMCTS Parallelization efficiency by agent

[2] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *Proc. 5th Int. Conf. Comput. and Games, LNCS 4630*, Turin, Italy, 2007, pp. 72–83.

[3] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo Planning," in *Euro. Conf. Mach. Learn.*, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds. Berlin, Germany: Springer, 2006, pp. 282–293.

[4] C. Browne, E. J. 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," *IEEE Trans. Comp. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, 2012.

[5] T. Cazenave and N. Jouandeau, "On the Parallelization of UCT," in *Proc. Comput. Games Workshop*, Amsterdam, Netherlands, 2007, pp. 93–101.

[6] G. M. J.-B. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel Monte-Carlo Tree Search," in *Proc. Comput. and Games, LNCS 5131*, Beijing, China, 2008, pp. 60–71.

[7] D. Whitehouse, E. J. Powley, and P. I. Cowling, "Determinization and Information Set Monte Carlo Tree Search for the Card Game Dou Di Zhu," in *Proc. IEEE Conf. Comput. Intell. Games*, Seoul, South Korea, 2011, pp. 87–94.

[8] P. I. Cowling, E. J. Powley, and D. Whitehouse, "Information Set Monte Carlo Tree Search," *IEEE Trans. Comp. Intell. AI Games*, vol. 4, no. 2, pp. 120–143, 2012.

[9] S. Gelly and Y. Wang, "Exploration exploitation in Go: UCT for Monte-Carlo Go," in *Proc. Adv. Neur. Inform. Process. Syst.*, Vancouver, Canada, 2006.

[10] L. Kocsis, C. Szepesvári, and J. Willemson, "Improved Monte-Carlo Search," Univ. Tartu, Estonia, Tech. Rep. 1, 2006.

[11] E. J. Powley, D. Whitehouse, and P. I. Cowling, "Bandits all the way down: UCB1 as a simulation policy in Monte Carlo Tree Search," in *Proc. IEEE Conf. Comput. Intell. Games*, Niagara Falls, Ontario, Canada, 2013, pp. 81–88.

[12] J. R. Long, N. R. Sturtevant, M. Buro, and T. Furtak, "Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search," in *Proc. Assoc. Adv. Artif. Intell.*, Atlanta, Georgia, 2010, pp. 134–140.

[13] D. Whitehouse, P. I. Cowling, E. J. Powley, and J. Rollason, "Integrating Monte Carlo Tree Search with Knowledge-Based Methods to Create Engaging Play in a Commercial Mobile Game," in *Proc. Artif. Intell. Interact. Digital Entert. Conf.*, Boston, Massachusetts, 2013.

[14] A. Bourki, G. M. J.-B. Chaslot, M. Coulm, V. Danjean, H. Doghmen, J.-B. Hoock, T. Hérault, A. Rimmel, F. Teytaud, O. Teytaud, P. Vayssière, and Z. Yu, "Scalability and Parallelization of Monte-Carlo Tree Search," in *Proc. Int. Conf. Comput. and Games, LNCS 6515*, Kanazawa, Japan, 2010, pp. 48–58.

[15] J. Méhat and T. Cazenave, "Tree Parallelization of Ary on a Cluster," in *Proc. Int. Joint Conf. Artif. Intell.*, Barcelona, Spain, 2011, pp. 39–43.