Distributed Artificial Intelligence

Come over

Aidan O'Neill

March 6, 2024

Abstract

We design and implement an efficient distributed agent that plays the board game Hex. The agent uses *Monte Carlo Tree Search*, a best-first search algorithm which considers all possible moves and explores the moves that appear to be most beneficial.

To implement this distributed agent, we develop a library of functions that performs high-performance computation across many nodes; these functions are particularly suitable to artificial intelligence and machine learning applications.

Declaration

On my honor I have neither given nor received unauthorized information regarding this work. I have followed and will continue to observe all regulations regarding it.

Acknowledgements

I want to thank my two advisors, Dr. Hammurabi Mendes and Dr. Bryce Wiedenbeck for all their time and support. Writing this document would not have been possible without them.

Contents

Chapter 1

Introduction

We present our efforts to create an efficient distributed agent that plays the board game Hex. Board games have long been a subject of research in artificial intelligence (AI); only shortly after Alan Turing wrote *On Computable Numbers* and created the field of Computer Science, Claude Shannon wrote on how to program a computer to play chess [22]. Recently, the AI community has achieved many milestones, as artificially intelligent agents have played games ranging from deterministic games such as Go to probabilistic and thus non-deterministic games such as Backgammon [4] [28]. In addition, agents can play Poker, a multi-player game in which players do not know what cards their opponents hold, allowing for more complex game-play [26]. More recently, agents have played Settlers of Catan, a multi-player, non-deterministic game [25]. All of these agents, as well as the current world champion agent that plays the board game Hex, `MOHEX 2.0`, use a technique known as Monte Carlo Tree Search (MCTS) to play their respective games [15].

Our agent uses MCTS to play Hex; we use Hex as it is a relatively simple game that has been extensively studied and thus is part of a rich sub-field in game-theory. Hex is a two-player board game in which players alternate turns placing tokens on a board in order to connect opposite ends of the board. We can think of these players as agents: in general, we understand an agent as something which interacts with its environment by making choices. Just as players of Hex are presented with a position on the board at a given moment in time, agents are presented with states, $S = \{s_1, s_2, ..., s_n\}$. Agents must then make choices between different possible actions, $A = \{(s_a, s_b), (s_c, s_d), ..., (s_y, s_z)\}$. Thus, actions are mappings from one state to another: when an agent chooses an action, the game transitions to the next state specified by that action. In Hex, these actions are different legal moves, and the set of actions thus corresponds to the set of all squares that have not yet been claimed by either player. We refer to how an agent chooses which action to take as its policy, P; a policy is an exhaustive mapping from states to actions. The policy that an agent adopts determines its playing strength - agents that have better policies necessarily choose better moves.

MCTS is a well-studied algorithm that allows an agent to learn and adopt a policy by repeatedly interacting with its environment. MCTS is a best-first search technique in that it explores those actions which are more promising preferentially before exploring those actions which seem less promising. MCTS provides us with one way to negotiate the trade-off between needing to explore those actions we have not yet evaluated and exploit those actions which seem most promising.

As MCTS becomes stronger as it explores more moves, to create a powerful MCTS agent, we need to leverage as many computational resources as possible towards the algorithm. Thus, we wish to take advantage of the processing power of multiple machines on the same algorithm; we call this distributed computing. While powerful, distributed computing makes already complex problems more difficult. Now, in addition to solving the question at hand, we must worry about communication between machines, communication that both complicates and occasionally slows down programs if not handled carefully.

To develop our distributed Hex agent, we create a flexible system designed for message passing between different machines as well as different threads within the same machine. This system, built on a message passing paradigm, allows for us to leverage the additional resources afforded by distributed computing while abstracting many of the difficulties inherent to distributed computing. To build this system, we use Message Passing Interface (`MPI`) as a foundation for collective operations between multiple processors and threads. We also use remote direct memory access (RDMA) to bypass central processing units (CPUs) and execute anonymous functions on remote machines efficiently on the Network Interface Controller (NIC).

This paper presents our experiments attempting to implement an efficient distributed MCTS agent that plays Hex, a two-player deterministic game. In addition, we demonstrate our distributed system's efficiency and applicability to machine learning and artificial intelligence, as well as its flexibility and utility for other developers. Finally, we explore possible avenues for future research in distributed MCTS as we continue to explore ways to fully take advantage of the processing power of the distributed paradigm.

Chapter 2

Hex

2.1 Hex

AI's history has been a progression from algorithms encoding human knowledge towards generic search algorithms. For instance, Stockfish, previously one of the strongest engines to play Chess, encodes extremely game-specific scenarios into even more specific penalties or rewards. For instance, the code below is typical of the Stockfish source code:

```
constexpr int QueenSafeCheck  = 780;
constexpr int RookSafeCheck   = 1080;
constexpr int BishopSafeCheck = 635;
constexpr int KnightSafeCheck = 790;
```

Here, we see that there is a penalty of 780 points for a given position if the opponent's queen is able to check your king without being immediately taken, as well as other specific penalties for a rook, a bishop or a knight being able to safely check your king [24]. This sort of thinking is intuitive to those who have studied chess: people are commonly taught that different pieces are worth different numbers of points (a queen is worth nine points, a knight is worth three, and so on and so forth), and this informs how they play the game. According to this line of reasoning, it makes sense that an expert system such as Stockfish simply takes this way of thinking and extends it to every conceivable possible type of position reachable in a given game. However, recent advances have moved away from this kind of heuristic analysis of game states towards algorithms that are aheuristic, i.e. do not use such functions developed a priori to approximate the goodness of different positions. In this newer approach, the agents learn what sorts of moves and positions are desirable in a given game through feedback from playing against other opponents or themselves. One chess engine which uses such an approach, AlphaZero, recently beat Stockfish at chess; AlphaZero learned how to play chess simply by playing against itself and receiving feedback from the results of these games [23]. In this chapter, we will explore these two general approaches through the lens of the board game Hex.

Hex is a two-player board game invented by Piet Hein in 1942 and perhaps independently by John Nash in 1948. To play the game, the two players alternately claim squares on a board in an attempt to connect their two respective ends of the board, as pictured in Figure 2.1. Though Piet Hein used the 11x11 board shown in Figure 2.1, other board sizes may be used (13x13 and 19x19 are also especially common, in homage to the game of Go, and Nash used a 14x14 board). Unless

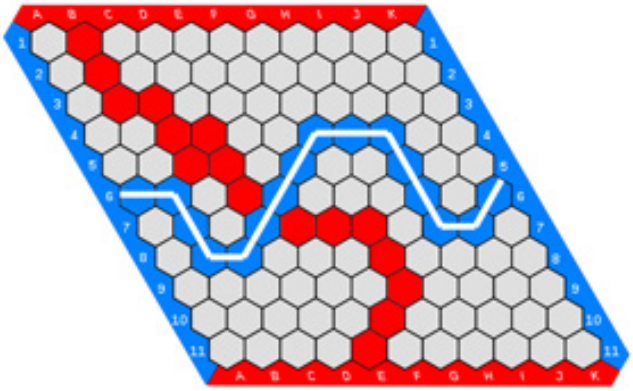

Figure 2.1: An example hex game [29]

otherwise stated, the reader should assume we are referring to the 11x11 version of the game. As the first player to make a move has a significant advantage in Hex, there is a variation of the game which practically reduces this advantage. In this variation, which uses a rule called the swap rule, the first player begins by selecting any square on the hex board. The second player may choose to either take that square for themselves or capture a different square. After this initial move, the game continues normally. Though this does significantly reduce Player 1's advantage and therefore serves to balance the game in practice, we use the vanilla version of Hex without this rule. Hex is deterministic as there is no random element to the game. Furthermore, Hex is zero-sum: every game of Hex must have a winner and there can be at most one winner, as proved in [10].

2.2 Previous Implementations

Hex has been the subject of much research in artificial intelligence [1] [13] [31] [17] [9] [15]. Though Gale proved that a winning strategy exists for the first player to make a move using a strategy-stealing argument, the specific strategy is yet unknown for boards of size 10x10 and greater [18] [10]. Arneson, Hayward, and Henderson achieved a milestone when they solved many 9x9 boards weakly using a technique known as modified H-search that applies Nash's strategy stealing argument and inferior weak cell analysis to reduce the search space [13]. We explore the idea of search space more thoroughly in Section 2.3. More recently, Young, Vasan, and Hayward outlined an agent which uses another technique known as deep Q-learning to play Hex. Finally, McCarver and LeGrand developed a Hex agent using well known genetic algorithms originally outlined by Fogel to create an Artificial Neural Network (ANN) [9]. The current world champion Hex agent, `MOHEX 2.0`, develops its policy using MCTS [15]. We refer the reader to [1], [13], [31], [17], [9] and [15] for more information on the implementation details of their respective agents.

Figure 2.2: Example Position in a game of Hex

2.3 State Space

If we implement an algorithm that explores every possible combination of moves one can make in a given board game, we can always find some optimal policy. We call this set of every possible combination of legal moves the *state space*. For instance, consider a game of 3 by 3 Hex which has progressed to the position in Figure 2.2, with Player *Red* to move next. *Red* wins by connecting the board horizontally; *Black* wins by connecting the board vertically. The state space is pictured in Figure 2.3.

If we assign Player *Red* winning a value of 1 and Player *Black* winning a value of -1 an agent can explore the state space using a method called Minimax search. Minimax search entails starting from the root node, i.e., the current position in a given game, and proceeding in a depth-first fashion. When it reaches a state in which the game has finished, which we will refer to as a leaf node or a terminal node, it uses the result of these leaf nodes to propagate up the tree. If the position results in a win for *Red*, we assign the leaf node a value of (1, -1), with the first number in the tuple representing the value of the leaf node for player *Red* and the second number in the tuple representing the value of the leaf node for player *Black*. Conversely, if the position results in a win for *Black*, we assign the leaf node a value of (-1, 1). When we use the result to propagate up the tree, we assume that the given player whose turn it is will choose that child node which is most beneficial. For instance, if it is *Red*'s turn to play, and any of the children of the given node result in a win for *Red* (have a value of (1, -1)), the parent node will similarly be assigned a value of (1, -1). Conversely, if it is *Black*'s turn to play, and any of the children of the given node result in a win for *Black* (have a value of (-1, 1)), the parent node will be assigned a value of (-1, 1). Thus, in the tree pictured in Figure 2.3, we see that *Red* will choose the middle path. *Black* is then indifferent between their two moves: both result in a loss for *Black*. This is intuitive to those who have played Hex. *Red* must prevent *Black* from winning by playing in the top middle hex square. After playing in that square, it is impossible for *Black* to connect the top and the bottom of the board, and *Black* cannot prevent *Red* from connecting the left and right. We can immediately improve on this algorithm through an approach called $\alpha\beta$ pruning. Imagine that it is Player *Red*'s turn to play from the same position pictured in Figure 2.2. If the search discovers playing in the middle path immediately, there would be no need to explore any other part of the tree, as *Red* has already discovered a winning continuation. It is important to note that *Red* needs to explore both paths after playing in the top middle hex square, as just because one of the paths results in a win does not mean that both do. *Red* must find a forced winning continuation. Thus, the $\alpha\beta$ search algorithm would, in this case, explore the state space pictured in Figure 2.4, a striking improvement over the state space pictured in Figure 2.3.

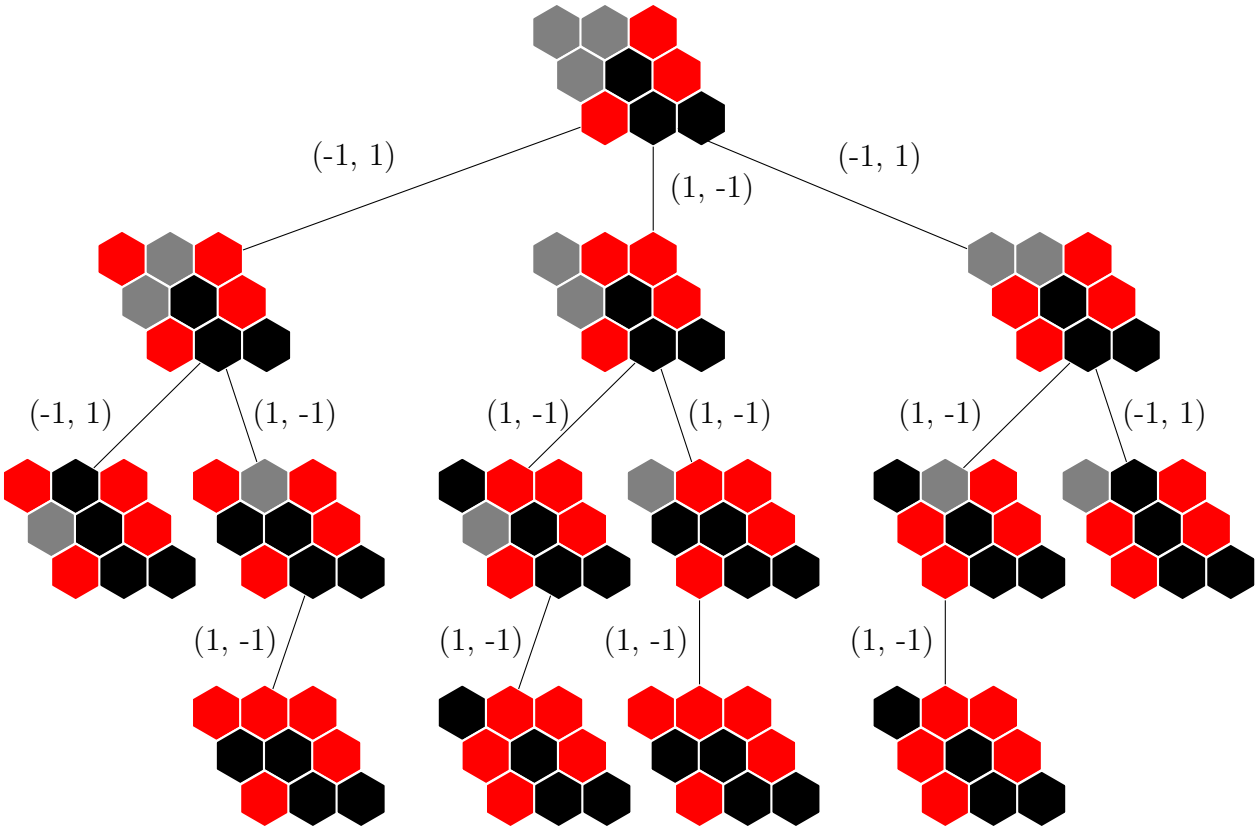

Figure 2.3: An example State Space from a Hex position

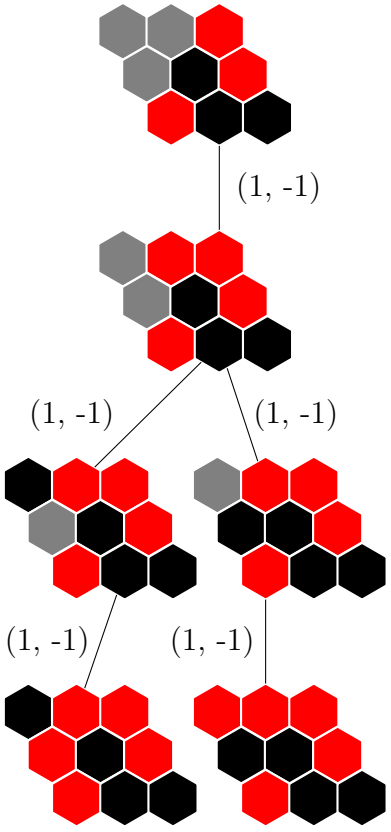

Figure 2.4: An example $\alpha\beta$ search from a Hex position

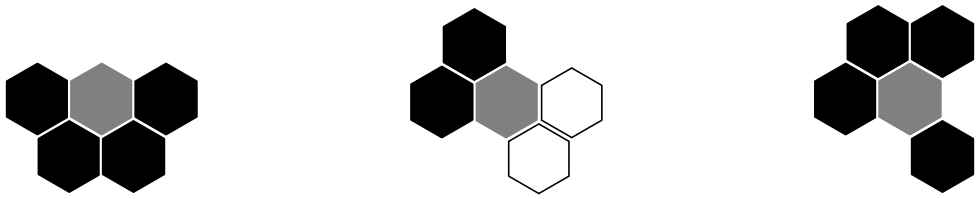

Figure 2.5: Hex Configurations with Dead Cells

While this is a clear improvement, in its worst case $\alpha\beta$ still grows proportionally to the size of the state space. Though both Minimax and $\alpha\beta$ search are guaranteed to find the optimal move in a given position, they take an incredibly long time to terminate for large search spaces. Hex has been proven to be `P-Space Complete`, and even with a technique such as $\alpha\beta$ which prunes the state-space, the size of the state-space for an 11×11 Hex board is $\approx 10^{57}$ [27]. Developing intuition for such large numbers is difficult at best. Suffice it to say that 10^{57} is the same order of magnitude as the number of hydrogen atoms in the sun. Clearly, algorithms which explore every possible move for a game of Hex will not terminate in our lifetime.

2.4 Game-Specific Knowledge

While our approach is to use aheuristic algorithms that do not employ advanced game-specific knowledge á la Stockfish, `MOHEX 2.0` (the current strongest agent) and many of the other strongest hex agents recognize common patterns in Hex in order to inform and prune the state space. Using this advanced knowledge of the game allows for much more efficient search for the best moves [14]. Specifically, we call a cell *inferior* if no matter which player captures it, the outcome of the game is not changed. For example, if a cell is surrounded by other cells that have already been captured by one player, capturing the surrounded cell cannot help either player connect their two respective ends of the board. Just as we can reason about individual cells, so too can we extend this analysis to sets of cells: a set of inferior cells can be called *dead* if it does not matter who captures it. In Figure 2.5, we see some examples of sets of dead cells: black cells have been captured by Player 1, white cells have been captured by Player 2, and grey cells have not been captured. Grey cells in all of these hex configurations are dead: who captures them will provably have no impact on the game.

In addition, a set of inferior cells can be called *captured* if one player has effectively gained control over that set of cells. In each of the hex configurations in Figure 2.6, by inspection we know that the set of cells is black-captured; we can assign them to black without altering the outcome of the game. Examples of inferior cell analysis are drawn from [3] and [15]. Needless to say, there are many examples of inferior cells, and using this advanced state-space knowledge can allow us to aggressively prune our search tree. In one study of famous hex puzzles, between 40% and 73% of the moves could be completely ignored due to inferior cell analysis of the board state. While this number can vary dramatically depending on the given

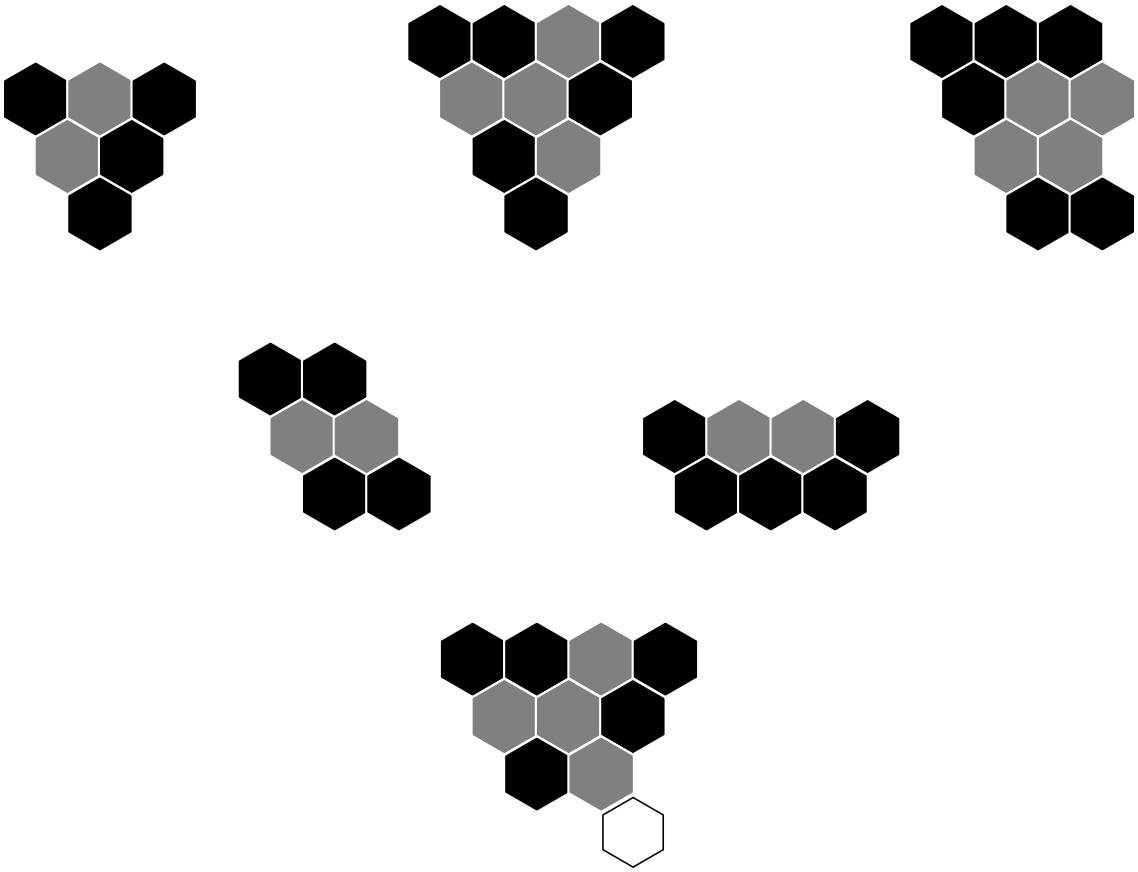

Figure 2.6: Hex Configurations with Captured Cells

Figure 2.7: A bridge in Hex

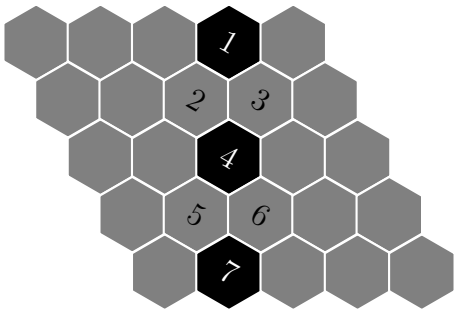

Figure 2.8: Adding bridges in Hex

board state, inferior cell analysis has dramatically improved the strength of some of the strongest Hex agents, especially `MOHEX 2.0` [15].

In addition to inferior cell analysis, many agents such as `MOHEX`, an earlier iteration of `MOHEX 2.0`, use connection strategies to inform their game-play. Here, they attempt to iteratively connect small portions of the board [15]. For example, in the Figure 2.7, we can consider the two black hexagonal squares connected, as no matter which of the unoccupied squares white plays in, black can play in the other, creating a connection between the two black cells. Other Hex-agents attempt to "add" these bridges together, and thereby create a winning strategy. For example, on a 5 by 5 board, an agent might attempt to achieve the following bridge network as in Figure 2.8.

If white plays in either square 2 or 3, black can answer with the other, connecting tiles 1 and 4. Similarly, if white plays in either of 5 or 6, black can answer with the other, connecting 4 and 7. Thus, no matter where white plays, black can connect 1 to 4 to 7 and thus the top and bottom of the board, so has a winning strategy through this series of bridges.

Thus, we see how different agents that play Hex can use either aheuristic approaches to explore a state space or algorithms informed by heuristics that employ advanced analysis of the game. We elect to use the first approach, especially as it is more easily extendable to aheuristic machine learning algorithms, which are becoming increasingly ubiquitous. We discuss our approach, MCTS, in the following section.

Chapter 3

MCTS

3.1 Motivation

Best-first search is a search mechanism which uses a heuristic to evaluate which moves seem most promising and then explore these most promising moves. We can think of MCTS as an algorithm by which we can programatically check those moves which we think of as most interesting. We understand those moves which are most interesting as those which provide the greatest benefit to the player in question. As we build a search tree by iteratively exploring these most promising moves, we note that we can terminate the algorithm at any time during execution. If we halt the algorithm part-way, we simply use our heuristic algorithm to evaluate the moves we have explored so far and adopt a policy that returns the most promising move we have seen. Of course, if the algorithm terminates, it means that we have fully explored the search space and thus have found the best possible move, similarly to $\alpha\beta$-pruning.

3.2 Sequential Monte Carlo Tree Search

To find the best move at a given board state, we iteratively build a tree of game states. Each node in this tree represents a given game state. If one node is the child of another, it means the child's game state is that state which occurs after an agent makes a legal move from the position represented by the parent node.

MCTS can be divided into four basic steps: *Selection*, *Expansion*, *Simulation* and *Backpropagation*. *Expansion* and *Simulation* are pictured in Figure 3.1. We build the tree one node at a time, by first finding that node which seems most promising (*Selection*). We then add one child to this most-promising node (*Expansion*), and evaluate the merits of this child by performing playouts from this child (*Simulation*). Playouts are one way to evaluate the goodness of a new game state - we simply play random moves until one player or the other wins. After playing n playouts from a given position, we see which player has randomly won from that position more often, giving us an estimate of the goodness of a given node for each player. We then use this estimate to update the goodness of those game states which we passed through to reach a given leaf game state; we call these game states predecessor nodes (*Backpropagation*). We then use these updated estimates of goodness to inform which node the algorithm thinks is most promising for the next execution of the MCTS algorithm. We will refer to a single iteration of the MCTS algorithm

in its entirey (i.e., *Simulation*, *Expansion*, *Simulation* and *Backpropagation*) as a *rollout*.

3.2.1 Selection

During the *Selection* stage, we select the most promising child node of our current node. We recursively perform Selection until we reach a node with unexplored child nodes. We can think of the question of which child node of a given node to visit as a multi-armed bandit problem. The multi-armed bandit problem is one in which a person is faced with a set of choices, with each choice generating a payoff when it is picked. The goal of the person is to maximize the sum of their payoffs over time as they make more and more choices. Thus, the dilemma facing the person is a tension between making those choices which have been beneficial in the past and trying new choices which they know less about but may be even more beneficial. This tension is referred to as the exploration-exploitation trade-off. We can think of the Selection stage of the MCTS algorithm as a stochastic multi-armed bandit problem in that when we make a choice, there is some probability distribution over possible payoffs for that choice, so we gain comparatively less information from making a given choice than if payoffs were constant. Thus, when we perform Selection on the MCTS tree we wish to choose a function that successfully negotiates this exploration-exploitation trade-off. We have limited resources so want to exploit those moves which seem promising, but also do not want to overlook potentially interesting and comparatively unexplored moves. There are many different possible selection functions which negotiate this trade-off: among these functions are the Gittins index, λ-greedy, UCB1, UCB2, UCB-Tuned, MOSS, KL-UCB, Bayes-UCB, POKER, Thompson Sampling and BESA [6].

Following the work of Kocsis and Szepesvári, we use the UCB1 Selection algorithm, which gives us an Upper Confidence Tree (UCT) version of MCTS. We use UCB1 as it ensures that we explore each child node for a given node once before visiting any child node twice. In addition, it has the advantage that we need not specify two distinct phases: one in which we explore those nodes which we have visited less, and one in which we exploit those nodes which have proved most beneficial. Rather, we just have one general phase in which we are constantly negotiating this tradeoff [6].

In particular, when we select a child node, we select the node that maximizes the formula

$$\frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{\log N_v}{N(v_i)}}$$

where $Q(v_i)$ is the number of wins that our player has achieved from the given child node, $N(v_i)$ is the number of times we have visited the given child node, N_v is the number of times we have visited the current node, and c is our exploration constant. $\frac{Q(v_i)}{N(v_i)}$ is therefore the fraction of times we have won from a given node and thus corresponds to the amount which we wish to exploit well-visited nodes. As $N(v_i)$ is the number of times we have visited the child node and N_v is the number of times we have visited the node, $\frac{\log N_v}{N(v_i)}$ is higher the fewer times we have visited the child node, and lower the more we have visited it. Thus, this amount varies inversely proportionally to the amount we have explored a given child node. If we wish to incentivize exploration, we raise c, and if we wish to incentivize exploitation, we

lower c. Unless otherwise noted, we use the default value of $\approx \sqrt{2}$ for c; this value is a hyperparameter that should be fine tuned for optimal performance [5]. If we are selecting a node, we have visited both the node in question and its child node at least once, thus N_v and $N(v_i)$ will both be at least one, and so we will never take the log of 0 or divide by 0.

3.2.2 Expansion and Simulation

Once we have finished recursively selecting nodes according to UCT, we will have reached a node with as-yet unvisited children. Once we have reached such a node, we create a new child node randomly, and then evaluate this new leaf node during the *Simulation* step. To evaluate this new node, some agents use state-space knowledge and therefore a heuristic. Other agents use a neural network to evaluate the given state-space. We can understand a neural network as a black-box heuristic [20] [23]. Though we consider both of these alternative approaches subjects for future research, in our MCTS algorithm, we use random playouts from the leaf node as a way to evaluate the goodness of the new node. We use this approach as it is our intention to keep our algorithm aheuristic in order to keep our algorithm as generic as possible. If we perform n playouts, whichever player wins more playouts is determined as the temporary winner for that node.

3.2.3 Backpropagation

In the backpropagation step of the algorithm, we recursively backtrack up the tree to visit the successor nodes of the newly expanded leaf node. We use the result of the *Simulation* portion of our algorithm to update the goodness of our parent nodes. In particular, we increment $N(v_i)$ for all of the successor nodes that we have visited on our path down the tree and increment $Q(v_i)$ depending on the results of the *Simulation* step of the MCTS algorithm.

3.3 Concurrency

In an attempt to increase the speed of a given algorithm, we can allow multiple threads acting within the same processor to operate concurrently. This multi-threading results in dramatic speed-ups. Many laptops today have four cores, and large multiprocessors can have up to thousands of cores. A sequential program only takes advantage of one of these cores to execute instructions in a specific order. On the other hand, a multi-threaded program allows different threads to execute instructions on all cores at the same time. We call simultaneous execution *threads acting in parallel*. Ideally, given the same amount of time, four threads acting in parallel perform the same work in a fourth of the time as one thread acting sequentially. This improvement influences the performance of algorithms such as MCTS, in which the strength of a given agent is largely determined by the amount of computation power available to said agent. Thus, we must take advantage of the potential gain in computational power afforded by multi-threading.

However, multi-threading presents two major difficulties. The first is the issue of concurrency. Successful execution of threads acting in parallel requires careful

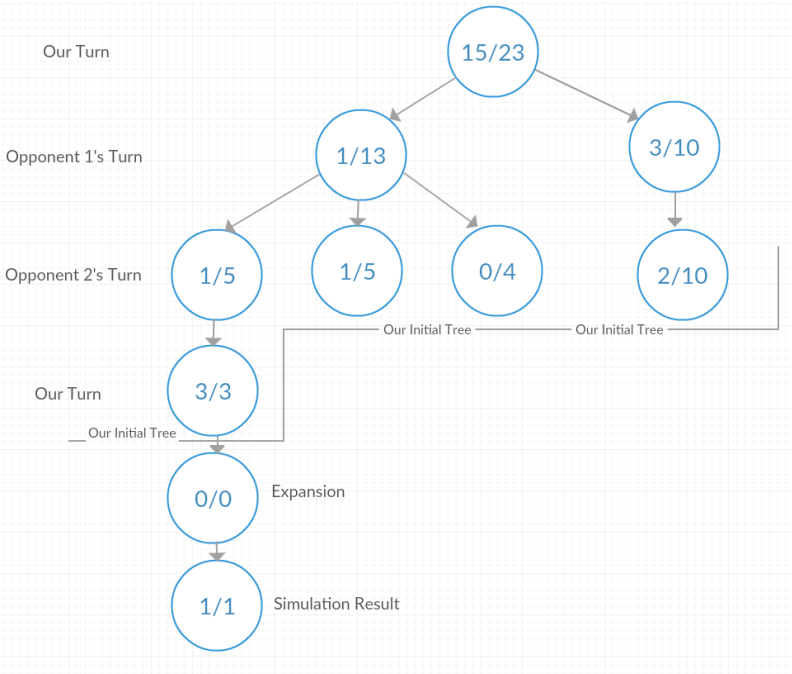

Figure 3.1: A simple, multi-player MCTS Tree, with $\frac{a}{b}$ representing that the node has resulted in a wins for a given player out of b total visits

management, since when multiple threads access, modify, create and delete various parts of the same data structure, the data structure can easily become corrupted.

For example, consider a program in which we have a very small MCTS tree and two threads. Both threads share a pointer to the root node, A. The first thread inserts B as a child node and increments the value for the number of visits to A, $N(A)$, changing it from 0 to 1. At the same time, the second thread inserts C as a child node and increments the value for the number of visits to A. However, as the second thread loads the value for $N(A)$ while the first thread has $N(A)$ loaded, it believes that $N(A)$ is only 0, and also changes it to 1. Thus, $N(A)$ is 1 when it should be 2. A concurrent algorithm is one which correctly manages situations such as this and avoids corruption of shared data.

To design a concurrent algorithm, we use two different mechanisms - the first of these we will refer to as locks or mutexes. Locks allow for mutual exclusion between threads. If we lock some data structure, only that thread which acquired the lock can access or modify that data structure until it unlocks it. Coarser-grained locks are locks that protect broader swaths of a data structure, and finer-grained locks are locks that protect smaller pieces of a data structure.

Consider the MCTS algorithm. A coarse-grained lock might have threads try to lock the tree in its entirety, allowing that single thread that acquires the lock to perform a rollout. A finer-grained lock might let different threads explore different parts of the tree, and only lock single nodes when we need to access or update them.

In addition to locking, another mechanism we can use is atomic variables. Atomic variables guarantee that concurrent operations will occur completely and will be visible to other threads. Specifically, they support a Compare-and-Swap (CAS) operation. CASing a variable takes the value that we expect a variable to hold and exchanges it for a new variable if and only if the value that we expect to see is equal to the value actually stored in memory. If the value differs, the CAS operation

fails. This operation occurs atomically, i.e. in one step. CASing thus allows for threads to read and write to memory safely in a concurrent setting. Where locks act as sentinels, ensuring that no other thread accesses a given piece of data, atomic variables act as a security system, in that they detect if other threads have modified a piece of data but not do not prevent them from doing so. Locks are pessimistic in that they make sure that no other threads access the same point in memory, whereas CAS operations are opportunistic, as they try an operation and then fail. If a CAS operation fails, we simply try again. Generally, locks are easier to implement whereas CASing takes better advantage of concurrency in algorithms.

The other difficulty of multi-threading is to take advantage of the additional resources that we use in a multi-threading environment. We call this the *parallelism* of the algorithm. For instance, consider the sequential MCTS algorithm as outlined in 3.2. We can trivially create a correct, albeit very inefficient, concurrent MCTS algorithm by simply locking the entire tree and allowing one thread to perform operations on the tree at a time. This algorithm is even slower than the sequential algorithm. We only have one thread performing rollouts at a time, and acquiring and releasing the lock takes time, as does context-switching between threads that perform operations on the tree. Context-switching is when the operating system saves the state of a particular thread so as to allow a different thread to execute instructions on the CPU. Due to this, a coarse-grained lock that locks the entire MCTS search tree takes no advantage of opportunities for parallelism in the MCTS algorithm. Thus, we prefer finer-grained locking to coarser-grained locking. Additionally, we prefer using atomic variables to locking, as atomic variables do not waste time acquiring and releasing locks, and do not pessimistically prevent other threads from accessing a variable in question but instead detect if another thread has happened to access that variable. Though atomic operations sometimes fail, using a try-and-check and repeat approach allows us to more fully use the resources available to us.

Thus, especially through the careful use of atomic variables, we can take advantage of parallelism in the MCTS algorithm. Taking advantage of parallelism allows for us to harness much more computational power, as we explore in Section 3.4.

3.4 Concurrent MCTS

Originally, efforts to parallelize MCTS were divided into three broad categories: leaf parallelization, root parallelization, and tree parallelization. In their paper Cazenave and Jouandeau explore leaf parallelization [7]. In this schema, only the Simulation portion of the MCTS algorithm is executed in parallel: Selection, Expansion and Backpropagation are all executed by a single thread. Leaf parallelization has the advantage of being relatively easy to implement, but simulation takes only a small portion of execution time for MCTS so it takes little advantage of the resources afforded by parallelizing the MCTS algorithm.

In addition, Cazenave and Jouandeau explore root parallelization. Here, each thread has responsibility for a single MCTS search tree; we effectively create t shadow trees, one for each thread. Once every thread has completed their MCTS search algorithm, the results are combined, and that node which was visited the most across all trees is selected as the best possible move. Once again, we can implement this with ease, but because the trees do not share information, each tree

navigates the trade off between exploration and exploitation without the help of the other trees. Hence root parallelization, similarly to leaf parallelization, takes little advantage of the resources afforded by parallelizing the MCTS algorithm. For instance, if one thread discovers a very good move early on, which it takes other threads significantly longer to find due to the randomness of MCTS, the work of the other threads is wasted. As such, following the work of Chaslot, Winands, and Den Herik, the efforts to parallelize MCTS have concentrated on variants of tree parallelization, which has proved the most effective means to implement a concurrent MCTS algorithm [8]. Here, all threads share access to and grow the same MCTS search tree. Each thread has responsibility for a single rollout, and performs the entire rollout from the root node to the leaf node and then back up the tree.

Though a trivial concurrent implementation of MCTS does afford us more processing power, making full use of the additional resources achieved by multithreading and multiprocessing paradigms proves to be significantly more difficult. MCTS only converges to the minimax algorithm explored in Section 2.3, and thus, the optimal value, as the tree gets very large. Improvement of computational speed through parallelization therefore directly affects our agent's strength.

To attempt to optimize the efficiency of the concurrent MCTS algorithm, Chaslot, Winands, and Den Herik explore different mutex locations within the algorithm. As different threads are responsible for different rollouts, multiple threads can try to access or modify the same node at the same time as they perform their separate rollouts. This simultaneous access can result in faults, as described in Section 3.3. However, as we discussed in Section 3.3, we prefer atomic operations to mutexes, as atomic operations are optimistic whereas mutexes are pessimistic. Thus, in lieu of locking nodes in order to only allow one thread to access each node at a given time, we make many of the variables in the node atomic. This atomicity allows us to take advantage of CAS operations.

In a concurrent MCTS algorithm built using CAS operations, we wish to reduce the amount threads attempt to access the same part of the data structure at the same time. We call this the amount of contention in the algorithm. Though preferable to locks, CAS operations are expensive and result in overhead, and this overhead increases as CAS operations fail and threads have to try to perform the same tasks over and over again until they succeed. One way to reduce contention is to introduce virtual losses as threads travel down the tree. That is, when we select a child node v_i, we increment the number of times we have visited it, $N(v_i)$. Thus, during backpropagation, we only update $Q(v_i)$. These virtual losses serve to dissuade multiple threads from traveling down exactly the same portions of the tree and therefore reduces contention in the tree. In additional, we explore distributed schemes that attempt to decrease this contention in Section 3.5.

3.5 Distribution

In Section 3.3, we explore the utility of multi-threading, a paradigm in which multiple threads in the same processor with access to the same memory coordinate on the same algorithm. Contention in the concurrent algorithm causes CAS operations to fail, increasing the amount of time it takes to perform rollouts. This is due to the fact that every rollout in MCTS begins from the root node. Naturally, many threads try to access and modify the root node. As threads do so at the same time,

CAS operations fail more often and the time it takes to perform rollouts increases.

Thus, we place a premium on reducing contention within the tree, especially higher up in the tree. To this end, we not only wish to run algorithms concurrently, but to distribute their work among different machines. This serves to both harness more processing power and cause fewer CAS failures. Different machines can have different memory spaces, so this means that we must break algorithms into discrete parts: parts that can be performed locally, on locations in memory to which a given processor has access, and messages to other processors to do work on other locations in memory to which a given processor does not have access. Thus, we can think of a distributed algorithm as an exercise in message passing between different processes, in which different processes instruct each other to execute different portions of the algorithm.

3.6 Distributed MCTS

One common approach to load balance between different processors in MCTS is transposition-driven scheduling (TDS). Here, we hash each possible board position, and assign ranges of hash values to different processors, thereby achieving load-balancing as each processor has responsibility for the same number of boards, which is equivalent to the number of nodes the processor has responsibility for. One of the most common hash functions used to achieve this load-balancing is the Zobrist hash. In [30], each processor is assigned a specific board; these boards correspond to nodes in the tree. Then, processors assume responsibility for selection down the tree and backpropagation up the tree for their respective nodes and assign new jobs to other processors based on the results of these selection and backpropagations. These jobs are maintained in a per-process queue.

In addition, to reduce the communication overhead inherent to the randomness of the Zobrist hash function, Yoshizoe et al. modify the MCTS algorithm from a best-first approach to a depth-first UCT approach, achieving a significant reduction in communication overhead [30]. In this approach, they do not have each thread begin each rollout from the root. Rather, a thread does rollouts locally in the best part of the tree. Threads only backtrack up the tree once it is possible that the part of the tree they are in has become worse than another part of the tree. Through this additional bookkeeping, Yoshizoe et al. reduces the communication between `MPI` processes as well as reduces some of the contention at the root node. In addition, Graf et al. explore breaking simulations into small "work packages", or discrete functions that need to be executed on the tree, and thereby split them up over multiple processes; they term this approach "UCT Tree-split" [11]. In addition to splitting up the memory as in [30], by splitting up the work and assigning it to different processes, they manage to improve on [30] as now all processors manage to be working around the same amount. In a later article, they explore improvements to the Zobrist hash and the TDS table in order to reduce communication in their distributed system. The Zobrist hash, in which we randomly generate hash values for different board states, maximizes communication, as different nodes likely belong to different processors.

When we implement this Zobrist based TDS distribution, we send two types of messages between nodes: trickle-up operations and trickle-down operations. These messages are packed into anonymous functions that need not be identified by any

particular name. We call these anonymous functions *lambda functions*.

One processor, the owner of the root node, signals the start of each rollout. When an MCTS node receives a trickle-down message, it determines whether it can select a child node, and if not, expands one of its child nodes. If it selects a child node, the ID of the process that owns the selected child node is computed, using the functionality of the `GlobalAddress` class, which is explored in Section 4.6. Then, it increments the number of visits of the child node it selects or expands in order to discourage other threads from picking the same child node.

These trickle-down select operations are passed between nodes using the `call` function in Dsys. Once an `MPI` process needs to expand a node remotely, we use the `call_return` function in Dsys. Both the `call` and `call_return` functions are explored in more depth in Section 4.2. This `call_return` function allows us to send a lambda that instructs another machine to create the child node in question locally and then return the `GlobalAddress` of that child node to the `MPI` process that owns the parent node. This `GlobalAddress` value returned through the `call_return` function allows the parent node to know where it can access its child on the remote machine. The ID of the remote machine is calculated through the Zobrist hash for the child node in question. Once the node has been expanded, it executes the simulation portion of the MCTS algorithm, and then signals the start of the trickle-up operation. This trickle-up operation is the backpropagation portion of the MCTS algorithm, and allows for all of the nodes that were traversed to the particular leaf to update the number of wins in their child nodes and thus serves to inform them which child they should pick when navigating the exploration exploitation trade-off. The trickle-up operations continue recursively up the tree until the root node of the tree is reached.

We know all of our rollouts have terminated when we have backpropagated to the root as many times as we perform rollouts in total. We keep track of the number of rollouts that have been executed through a `Synchronizer` object wrapped in a `GlobalAddress`; the semantics of the `Synchronizer` class are discussed in Section 4.4. We can then terminate the distributed MCTS algorithm either when a set number of rollouts have been executed or when a set amount of time has passed.

Thus, through these two functions, we see how we can effectively break the MCTS algorithm into discrete components that operate on singular nodes. We can then execute an MCTS search through passing these discrete portions of the MCTS algorithm between different processors which own different nodes and thus different sections of the tree, taking advantage of the computational power afforded to us by the multiple processors.

Chapter 4

Dsys

4.1 Motivation

In the era of big data, the problems that scientists face rely on vast amounts of data that cannot be stored in a single machine. One such problem is MCTS. Rather, scientists must use clusters of computers to hold the data requisite to solving their complex problems. In addition, some problems are so computationally intensive that they are impractical to run on one machine, further motivating the use of clusters of computers. Though distributed computing allows us to solve more computationally intensive problems with more data, it also makes programming significantly more difficult. Now, not only does one need to worry about an algorithm that is difficult to execute sequentially, but must also worry about communication between machines, communication that both complicates and occasionally slows down programs. Thus, the difficulties inherent to distributed computing are myriad: communication between processors is faulty, processors can corrupt shared data, communication takes time, and it is easy to under use the available resources as some processors sit idly while others shoulder the bulk of the work.

To ease these difficulties, many distributed libraries exist to facilitate scientists' development of distributed applications. However, some of these libraries, such as `MPI`, are so flexible that developers find themselves solving the same design questions again and again. `MPI` is an incredibly flexible, powerful, low-level library. That said, it requires an immense amount of effort to write even simple distributed programs; it is too low-level for efficient development and furthermore does not lend itself well to multi-threading. Developers often find themselves spending inordinate amounts of time parsing obscure documentation and working on small distributed bugs rather than thinking about ways to intelligently pass messages between machines in a manner that minimizes communication and maximizes resource usage.

Other libraries, such as GRAPPA, successfully ease some of the complexities inherent to distributed programming. They do so by providing safeguards against corrupting shared data or allowing developers to pass functions between processes and have them be executed remotely through some interface. However, in doing so, they provide such a rigid framework that programmers find themselves developing code to fit the requirements of the library in question rather than writing code with the application in mind [19]. Our Dsys library abstracts some of these same complexities inherent to performing computationally intensive tasks across many nodes while maintaining sufficient flexibility that developers need not conform to

an entirely new paradigm. We explore our abstractions of these complexities in the following sections.

4.2 Remote Calls

In distributed computing, we wish to pass messages between machines to coordinate the machines' efforts on a particular problem. For example, if we have two machines, A and B, and we wish to sort an array, a typical distributed program might work as follows. A begins the program, and takes responsibility for sorting the first half of the array. In addition, A sends a function to B telling B to sort the second half of the array. A sorts its half of the array and waits for B to send back its sorted half of the array. Once these two things have completed, A combines the two sorted arrays into one larger sorted array. A sending the function to B telling B to sort the second half of the array is called a remote call. If the array that is to be sorted is very short, there is little point to this sort of distribution, as the added communication between the two takes extra time and the code to sort an array like this is orders of magnitude more complicated than simple sequential sorting algorithms. However, if the array is very large, this cuts the amount of time it takes to sort the array almost in half. Thus, the ability to make remote calls of this sort is one of the most important building blocks for distributed computing.

Within this, there are broad paradigms: remote calls across processors and remote calls across threads. We call this multi-processing and multi-threading respectively. As we explored in Sections 3.3 and 3.5, multi-processing is a paradigm in which we coordinate the efforts of multiple CPUs with access to different virtual memories. Multi-threading is a paradigm in which multiple agents within the same processor and thus, with access to the same memory, coordinate efforts on a problem. We use a mixed or hybrid paradigm in which we take advantage of both multiple processors and multiple threads per processor to achieve high speeds and low delays.

Our library provides support for passing anonymous lambda functions between different threads within different processes. We use remote direct memory access (RDMA) to bypass CPUs and pass these lambdas between different threads on different machines. Decomposing a more complicated task into disparate parts that can be executed asynchronously is essential to distribution.

To that effect, our system allows for different threads within a machine as well as on different machines to send lambda functions to each other in order to efficiently distribute a given task. We build on the functionality of `MPI` to allow different threads the ability to send these lambdas. Ideally, the cores on each machine are working at full capacity at all times, with different cores working on the tasks associated with different threads at all times. The operating system is in charge of assigning these cores to do work, and thus creates, destroys, schedules and deschedules threads as needed. Creating and destroying threads is expensive; scheduling and descheduling, though less expensive than creating and destroying threads, are still costly operations. Thus, if we needlessly create and destroy threads or launch more threads in a machine than there are cores to do work, the efficiency of our distributed algorithm naturally suffers. It is our aim to create a reasonably flexible system that allows developers to easily launch a given number of threads to do work in each `MPI` process. This allows for developers to easily optimize the amount of time that cores

spend doing the work of the program and minimize the amount of time that the cores spend doing the expensive bookkeeping associated with multithreading.

Dsys launches one receiver or sentinel thread in each process. This receiver spins in a while loop, listening for messages that take the form of discrete lambdas from other processes. The receiver does not needlessly use resources, as we use the `MPI_Probe` function to detect incoming messages from other processes; `MPI_Probe` is a call that blocks the execution of the receiver until it receives a message, in contrast with `MPI_IProbe` which executes asynchronously and does not block the execution of that thread which performs the `MPI_IProbe` call. In general, `MPI` calls that take the form `MPI_Func` block that thread which called them until the call has completed, i.e., they execute synchronously. Conversely, `MPI` calls that take the form `MPI_IFunc` execute asynchronously and do not block that thread which called them; that thread cannot assume that the call has completed successfully unless so signaled through an `MPI_Wait` call on the outstanding asynchronous operation. Once the receiver gets a message from another processor with `MPI_Probe`, it executes the lambda that has been sent by the other process. We execute the lambda locally using the std::async functionality of c++11. On many operating systems, the std::async call takes advantage of a thread pool. In a thread pool, the operating system does not create and destroy threads; creating and destroying threads are extremely expensive operations. Instead, using std::async and its thread pool allows the operating system to create many threads at the beginning of program execution. Then, the threads are scheduled and descheduled by the operating system as needed to perform asynchronous calls, a significant improvement over creating and destroying them. If there are more tasks than there are threads, c++ simply increases the size of the thread pool. Thus, the std::async call is preferable to launching a thread to asynchronously execute every function. However, if the async call fails, we lazily launch additional threads to perform the function. The receiver thread does not wait for these new child threads to finish execution before continuing to do this work; we call this child threads *detaching* from the receiver thread. Thus, whether through std::async or by launching and detaching a thread, the receiver can always immediately return to listening for additional messages from other processes.

Dsys requires minimal maintenance, as one of our goals in designing this system is to ease distributed development. A example program is below in Listing 4.1. We call dsys:: init_thread_handler with the usual argc and argv arguments to initialize the distributed system. We expect the second argument to the command line to be the number of threads which the programmer wishes to launch per process. In addition to this init call, there is a minor amount of bookkeeping necessary; we expect the user to initialize the number of threads per process and the total number of threads manually. Once all of the threads have started and completed their tasks, as shown in the two for loops with an arbitrary function called *task*, the user only needs to call dsys:: finalize_thread_handler in order to finalize the distributed system. This ensures that we have not lost track of dynamically allocated memory, i.e. no resources were leaked by the system. We do not check to make sure that the user did not leak resources in their application - the user alone is responsible for the correctness of the user's program.

Listing 4.1: An example Dsys program

```
vector<thread> thread_list;
if(argc != 2) {
```

```
        exit(EXIT_FAILURE);
    }
    number_threads_process = atoi(argv[1]);

    init_thread_handler(argc, argv);

    number_threads = number_threads_process * number_processes;

    for(int i = 0; i < number_threads_process; i++) {
        thread_list.push_back(thread(task, i));
    }
    for(int i = 0; i < number_threads_process; i++) {
        thread_list[i].join();
    }
    finalize_thread_handler();
```

Once dsys has been initialized, we allow for a great deal of flexibility with remote calls to other processes and threads. These remote calls are essential to passing lambdas between different threads. Without the ability to pass these small, discrete functions around, it is incredibly difficult to write distributed algorithms quickly. The flexibility of these remote calls is paramount to easy development of distributed code. These functions are built using `MPI`, abstracting many of the difficulties inherent to using `MPI`; `MPI` is built to send discrete chunks of data between processors. This does not lend itself to either passing messages between specific threads on different machines or quickly and efficiently passing functions to be executed on a remote process. To this end, we design a wide variety of functions in order to allow the developer to pass whatever type of function they wish between threads quickly and efficiently. We template these calls so that c++ can automatically deduce the type of each function. One can then simply create a lambda using c++'s built-in lambda notation, which combined with the auto type allows for c++ to automatically deduce the type of these functions. Once the function has been created, we provide the function signatures in Listing 4.2 in an attempt to ease facilitated development.

Listing 4.2: The signatures of some different call functions

```
void call(int destination_thread_id, F &function);
auto call_return(int destination_thread_id, F &&function);
auto call_return(int destination_thread_id, F &&function,
    GlobalAddress<Synchronizer> &synchronizer);
auto call_return(GlobalAddress<T> address, F &&function);
void call(GlobalAddress<T> address, F &&function);
void call_thread(GlobalAddress<T> address, F &&function);
void call_thread(int destination_thread_id, F &&function);
void call_everyone(F &&function);
void call_everyone(F &&function, void *data, uint64_t size);
void call_register(int destination_thread_id, F &&function,
    GlobalAddress<Synchronizer> synchronizer);
void call_register(int destination_thread_id, F &&function,
    G &&notify_function, GlobalAddress<Synchronizer> synchronizer);
void call_buffer(int destination_thread_id, F &&function,
    U *data, uint64_t size);
```

`call` is the simplest function. This allows us to send a specific function to a specific thread, which will then be executed by that thread. For any call in dsys, if the function's destination is the same processor as the sender, it is simply executed

locally, without making a call to `MPI`. In addition to this simple call function, we support calls which return a value, which we export as the function `call_return`. This function calls a lambda remotely. The original function is wrapped with a call from the original destination to the sender, in which a `GlobalAddress` is populated with the return value of the function passed to `call_return` - we explore the `GlobalAddress` class in more detail in 4.6. Once this `GlobalAddress` has been populated, the function returns, thus `call_return` is a blocking function.

Such a simple idea as a blocking function that returns across multiple processes is difficult to implement. Sequentially, one would simply specify the return type of a given function and then have that same function return something. Distributed, the code to execute a typical `call_return` is as shown in Listing 4.3.

Listing 4.3: Typical distributed code for a function that returns from another process

```
template<typename F>
auto call_return(int destination_thread_id, F &&function) {
int destination_process_rank = (destination_thread_id
                                / dsys::number_threads_process);

if(destination_process_rank == process_rank) {
    return function();
}

Synchronizer synchronizer(1);
GlobalAddress<Synchronizer> global_synchronizer(synchronizer);

decltype(function()) result;
GlobalAddress<decltype(result)> global_result(result);

auto lambda = [function, global_synchronizer, global_result]() {
    decltype(function()) result = function();

    auto notify_function = [global_synchronizer, global_result, result] {
        *(global_result.get_address()) = result;

        global_synchronizer->decrease();
    };
    int dest_thread = global_synchronizer.get_thread_id();
    dsys::call(dest_thread, notify_function);
};
dsys::call(destination_thread_id, lambda);

global_synchronizer->wait();

return *(global_result.get_address());
}
```

This is heinous, especially when compared to typical sequential code. First, we check to see if our process id is the same as the destination process so as to avoid the overhead of making an `MPI` call unnecessarily. Then, so that the calling thread knows when the remote process returns, we create a `Synchronizer` object wrapped in a `GlobalAddress` thread that allows the called thread to signal on our process that the function has returned. Otherwise, the calling thread might return prematurely. Then, we create an object of the type which the function will return, and again wrap it in a `GlobalAddress`. Wrapping the `Synchronizer` and return objects

in `GlobalAddresses` mean that other threads on the same processor will be able to populate the return object and signal the population of the return object on the `Synchronizer` object. Thus, rather than sending a lambda that constitutes just the function parameter to the destination thread, we send a lambda that first executes the function, then takes the return value of the function. It then sends that result back to the original calling thread, populating the result in the result object and then signalling that the result has been populated on the `Synchronizer` object. This results in the original thread releasing from the wait() call and returning the result of the function call. What in sequential code is trivial clearly becomes orders of magnitude more difficult when we write distributed code, so we see that exporting these succinct functions that abstract these difficulties results in a huge burden being lifted from distributed developers. Though we will not explore the other functions in similar depth, it is needless to say that the other `call` functions that we export are similarly convoluted. In addition, it is worth noting that the `Synchronizer`, `GlobalAddress` and `call` functions that we use in `call_return` are similarly elaborate; we do not discuss them here in an attempt to spare the reader.

As some developers might wish to use `call_return` asynchronously, we include another version of `call_return` which takes a `GlobalAddress<Synchronizer>` object as a parameter indicate that they have returned using the wrapped `Synchronizer` object; the `Synchronizer` class is discussed in more detail in 4.4. This way, one thread can continue doing other work, and then, when it needs the result of the `call_return` call, it can wait on the `Synchronizer` object, allowing for developers to take more full advantage of parallelism in their algorithms. `call_thread` functions allow developers to force the thread on which they are calling the given function to spawn another thread to execute the function in question, allowing the destination thread to continue its work once it spawns the child thread.

As oftentimes, a developer may wish to call the same function on every thread, we abstract this difficulty to our class of functions `call_everyone`, in which every thread on every processor executes the same function.

In addition to supporting functionality for simple sends with discrete lambdas, we consider the necessity of sending lambdas which operate on arrays of data, especially massive arrays of data that are time-consuming and expensive to send between processes. We export this functionality in the functions labeled `call_buffer_...` and pass serialized versions of lambda functions using `FunctionWrapperBuffer`. In this case, the function and the buffer are sent in two separate `MPI` sends. Separating these two calls allows us to send a function of one type and a buffer of another type. However, separating these two calls means that the receiving thread cannot simply execute the function immediately, as the buffer arrives some time after the function. The receiver thread therefore stores the functions in a map. As each `MPI` send allows us to use a tag, the key for this map is the `MPI` tag and the value is the function. When the buffer is received, it is matched to the function via the `MPI` tag. Then and only then is the lambda executed on the array of data in question. Finally, we allow for functions which register their result with a `Synchronizer` object in the family of functions `call_register`. We explore the `Synchronizer` object and its uses more fully in Section 4.4. The `Synchronizer` object allows the developer in question to register the number of calls which have been completed. For example, if one function can only be executed once another has completed or once several other functions have completed, the thread that will execute that dependent function

can wait on the `Synchronizer` object. Thus, these allow for points of synchronicity between different processors and allow for better interleaving of the work of different threads.

Thus, we see how we distill distributed tasks that would otherwise be exasperating at best into small, easy to use, `call` functions. We attempt to anticipate the different needs of distributed developers, hence the broad range of types of functions and flexibility within each of these types of `calls`.

4.3 MPI Thread Helper

In addition to this family of remote calls, we allow for collective operations between threads. `MPI` is the gold standard for a multi-processor communication system [12]. More recently, `MPI` has begun to provide support for hybrid multi-processor multi-threaded implementations. The highest level of multi-threading support, `MPI_THREAD_MULTIPLE`, allows multiple threads to simultaneously call `MPI` functions from different processes [2].

However, while this support is provided, many of the collective operations used by `MPI` are ill-defined when accessed by multiple threads; documentation is practically nonexistent and `MPI` does not provide a natural extension from passing messages between processors to passing messages between threads. For example, one of the simplest collective operations in `MPI` is broadcasting. Consider four processes which we label $\{P_0, P_1, P_2, P_3\}$. P_0 has an array $\{1, 2, 3, 4, 5, 6, 7, 8\}$ which it wishes to broadcast to the other three, so it calls `MPI_BROADCAST`. So long as every other process calls `MPI_BROADCAST` with an array with empty space allocated, they receive the array $\{1, 2, 3, 4, 5, 6, 7, 8\}$. However, if multiple threads all call `MPI_BROADCAST` with references to the same shared memory on different machines, the result is corruption of shared data or a fault. As such, we naturally extend the `MPI` interface through the MPI_THREAD_HELPER module. We will consider the example of scattering to understand how MPI_THREAD_HELPER serves to naturally extend the `MPI` standard. Scattering in the `MPI` standard traditionally involves one process distributing data to each other process. For example, consider a four-processed program in which we denote each separate process $\{P_0, P_1, P_2, P_3\}$, as above. P_0 has an array $\{1, 2, 3, 4\}$ which it wishes to scatter. If it and all of the other processes call `MPI_SCATTER`, the result is that process P_0 has an array with only 1 in it, P_1 has an array with 2 in it, P_2 has an array with 3 in it, and P_3 has an array with 4 in it. Now consider the same program, except each process has two threads. We denote P_j i-th thread as P_j^i. Thread P_0^0 has an array $A = \{0, 1, 2, ..., 14, 15\}$ which it wishes to scatter. So long as every thread in our distributed setting calls `MPI_Thread_Helper_SCATTER`, each thread will acquire two elements from the array. For example, after the call, thread P_0^0 will have an array $A' = \{0, 1\}$, P_0^1 will have an array $A' = \{2, 3\}$, and so on and so forth up to thread P_3^1 which will have ownership over $A' = \{14, 15\}$. Thus, the MPI_THREAD_HELPER module uses `MPI` as a foundation for collective operations between multiple processors, and naturally extends the functionality of `MPI` so as to better facilitate hybrid operations between multiple threads and multiple processes. We provide support for many of the collective operations that serve as the backbone of any distributed system implemented using `MPI`, notably, gather, scatter, allGather, broadcast, allToAll, allToAllV, and barrier. The multi-threaded extension of these collective operations

Table 4.1: `MPI_THREAD_Helper_gather`

P ID	T ID	T Rank	SendBuffer	RecvBuffer
0	0	0	[0]	[0 1 2 3 4 5 6 7]
0	1	1	[1]	[ ]
1	2	0	[2]	[ ]
1	3	1	[3]	[ ]
2	4	0	[4]	[ ]
2	5	1	[5]	[ ]
3	6	0	[6]	[ ]
3	7	1	[7]	[ ]

Table 4.2: `MPI_THREAD_Helper_scatter`

P ID	T ID	T Rank	SendBuffer	RecvBuffer
0	0	0	[0 1 2 3 4 5 6 7]	[0]
0	1	1	[ ]	[1]
1	2	0	[ ]	[2]
1	3	1	[ ]	[3]
2	4	0	[ ]	[4]
2	5	1	[ ]	[5]
3	6	0	[ ]	[6]
3	7	1	[ ]	[7]

are depicted in Tables 4.1,4.2, 4.3, 4.4, 4.5, and 4.6, except for barrier. The barrier extension, much as the `Barrier MPI` function does, suspends each thread in each machine that calls a specific barrier until all threads in all processes have reached that same barrier object, thereby providing a point of synchronicity. Then and only then does it release the threads to continue program execution.

4.4 Synchronizer

We consider providing points of synchronicity within a program essential to effective distribution of a program. As mentioned in Section 4.2, when a function depends on the successful completion of another function first, we need some sort of method to allow for different threads and different processes to know when certain discrete parts of the distributed algorithm have run to completion. C++'s standard has a `condition_variable`, which allows for a thread to signal other threads when it has completed a given function. This `condition_variable` is not thread-safe, and therefore must be protected by a `mutex`. This makes `condition_variable`'s difficult to use safely as the `mutex` allows for deadlocks between different threads without careful management. Therefore, we abstract these difficulties to a more easily man-

Table 4.3: `MPI_THREAD_Helper_allGather`

P ID	T ID	T Rank	SendBuffer	RecvBuffer
0	0	0	[0]	[0 1 2 3 4 5 6 7]
0	1	1	[1]	[0 1 2 3 4 5 6 7]
1	2	0	[2]	[0 1 2 3 4 5 6 7]
1	3	1	[3]	[0 1 2 3 4 5 6 7]
2	4	0	[4]	[0 1 2 3 4 5 6 7]
2	5	1	[5]	[0 1 2 3 4 5 6 7]
3	6	0	[6]	[0 1 2 3 4 5 6 7]
3	7	1	[7]	[0 1 2 3 4 5 6 7]

Table 4.4: `MPI_THREAD_Helper_broadcast`

P ID	T ID	T Rank	SendBuffer	RecvBuffer
0	0	0	[0 1 2 3 4 5 6 7]	[0 1 2 3 4 5 6 7]
0	1	1	[ ]	[0 1 2 3 4 5 6 7]
1	2	0	[ ]	[0 1 2 3 4 5 6 7]
1	3	1	[ ]	[0 1 2 3 4 5 6 7]
2	4	0	[ ]	[0 1 2 3 4 5 6 7]
2	5	1	[ ]	[0 1 2 3 4 5 6 7]
3	6	0	[ ]	[0 1 2 3 4 5 6 7]
3	7	1	[ ]	[0 1 2 3 4 5 6 7]

Table 4.5: `MPI_THREAD_Helper_alltoall`

P ID	T ID	T Rank	SendBuffer	RecvBuffer
0	0	0	[0 1 2 3 4 5 6 7]	[0 8 16 24 32 40 48 56]
0	1	1	[8 9 10 11 12 13 14 15]	[1 9 17 25 33 41 49 57]
1	2	0	[16 17 18 19 20 21 22 23]	[2 10 18 26 34 42 50 58]
1	3	1	[24 25 26 27 28 29 30 31]	[3 11 19 27 35 43 51 59]
0	4	0	[32 33 34 35 36 37 38 39]	[4 12 20 28 36 44 52 60]
1	5	1	[40 41 42 43 44 45 46 47]	[5 13 21 29 37 45 53 61]
1	6	0	[48 49 50 51 52 53 54 55]	[6 14 22 30 38 46 54 62]
1	7	1	[56 57 58 59 60 61 62 63]	[7 15 23 31 39 46 55 63]

Table 4.6: `MPI_THREAD_Helper_allToAllV`

P ID	T ID	T Rank	SendBuffer	RecvBuffer
0	0	0	[0_0 1_0 2_0 3_1 4_1 5_3 6_3 7_7]	[0 1 2 8 24 40 41 48 56]
0	1	1	[8_0 9_1 10_2 11_2 12_2 13_4 14_4 15_5]	[3 4 9 25 26 27 28 42 43 49]
1	2	0	[16_5 17_5 18_5 19_5 20_6 21_7 22_7 23_7]	[10 11 12 44 45]
1	3	1	[24_0 25_1 26_1 27_1 28_1 29_4 30_4 31_7]	[5 6 46 47 50 51 52 57]
0	4	0	[32_4 33_4 34_4 35_5 36_5 37_6 38_6 39_6]	[29 30 32 33 34]
1	5	1	[40_0 41_0 42_1 43_1 44_2 45_2 46_3 47_3]	[16 17 18 19 35 36 58 59]
1	6	0	[48_0 49_1 50_3 51_3 52_7 53_7 54_7 55_7]	[20 37 38 39 60]
1	7	1	[56_0 57_3 58_5 59_5 60_6 61_7 62_7 63_7]	[7 21 22 23 31 53 54 55 61 62 63]

Note that we use the notation n_t to denote an element n which is to be sent to thread rank t.

aged `Synchronizer` class, which we guarantee will result in no deadlocks between threads. The `Synchronizer` takes the number of operations which must be completed. Any thread can signal that they have completed a task on the `Synchronizer` object through a call to `decrease`. When a thread completes the last of the operations that have been assigned, it uses the `Synchronizer` object to release every thread that has been waiting for these operations to complete. Finally, we allow for developers to reuse `Synchronizer` objects. One must call the reset function between separate uses; without the reset call in-between uses, a `Synchronizer` object will not work as expected. Thus, we see how we can use the `Synchronizer` object to keep track of the amount of work that has been completed by many different threads and how it abstracts the difficulties inherent to using some of the lower-level concurrent functions available from C++.

4.5 Chimera

Occasionally, when we call lambdas on remote machines on static addresses, developers may want different recipient threads to interpret this address differently. For instance, if you have unique local copies of certain types of objects, we may want to map a specific function onto these different local copies. One trivial example is a developer wishes to sum the elements in an array. To this end, the developer assigns a distinct part of the array to each thread in question. Each thread then sums the part of the array for which they are responsible, and then a master thread sums the result of these sums. The `chimera` package is designed to allow developers to more easily complete tasks like this, and have different processes or different threads enact some function on an instance of an object that is different depending on which process or thread is executing.

The `chimera` package is a group of classes that provide support for having different machines have different objects stored in the same location in memory. Here, we provide a class for pointers and objects that can be declared as unique per-process or unique per-thread within each process. We build on the barrier functionality of `MPI_THREAD_HELPER` to ensure that a pointer or an object has been created or referenced across all threads or processes, respectively, before declaring that particular instance of a class to be safe to use. As the barrier provides a point of synchronicity, if all threads have called our `Barrier`, we know that the local object has been created by all threads. An individual thread can detect whether that particular class is safe to use through the wait call, which only returns when every thread or process has checked in and initialized the class in question. This class abstracts much of the difficulty of sharing memory between different threads and processes and thereby is particularly useful to developers who often wish to use the same object or reference to an object across multiple different agents within a distributed system. We support four different classes; the `SymmetricThreadPointer`, `SymmetricThreadObject`, `SymmetricProcessObject` and the `SymmetricProcessPointer`. As the names suggest, each guarantees that they will be in the same location in memory; a per thread pointer or object means that each thread has a unique copy, whereas a per process pointer or object means that each machine will have a unique copy of an object and that threads that operate within the same machine will share access to that copy. Classes that we designate as pointers mean that the static location in memory refers to a pointer; we overload the $->$ operator in order to facilitate idiomatic

development. Similarly, classes in chimera that we designate as objects mean that the static location in memory refers to an object. Between these four classes, we provide great flexibility to developers.

4.6 GlobalAddress

Finally, a backbone of our system is Remote Direct Memory Access, or RDMA. Sometimes, we wish to access an object on a remote machine that is not necessarily in the same, static place in memory. The `GlobalAddress` wrapper class allows us to not only store the address of a given object, but also which thread and machine that process is stored on, so different machines know where to access the different objects. A `GlobalAddress` object only has two fields.

```
uint64_t address; //16 bits of flags, 48 bits of address
uint64_t process_thread; //32 bits of process, 32 bits of threads
```

Thus, we see how it is a lightweight object with only two `uint64_ts`. The last 48 bits of the address field refer to the actual location in memory of a given object; the first 32 bits of the process_thread field let us know which machine a given object is stored on and the last 32 bits similarly let us know which thread on a given machine a given object is stored on. It follows that dsys supports 2^{32} distinct machines and 2^{32} threads per machine; more machines than this would result in overflows in our `GlobalAddress` objects. To facilitate easy, idiomatic development, we overload the typical pointer operators, i.e. $->$ and *. This allows developers the ability to treat a `GlobalAddress` as though it were a pointer.

Thus, we see how dsys forms a flexible but lightweight distributed system. While we expect the developer of any given distributed application to ensure the correctness of their algorithm, we ease their development significantly. We do this through abstracting complexities appropriately and providing classes that anticipate common needs in distributed computing. As we saw in 3.6, many of these classes and functions were central to our development of a distributed MCTS agent, serving to further motivate their utility as a framework for complex distributed algorithms.

Chapter 5

Experiments and Results

5.1 Experiments

To illustrate the importance of distributed computing and the impact distributed computing can have on computationally intensive algorithms such as MCTS, we developed three MCTS agents, `Seq`, `Conc`, and `Dist`. Our `Seq` agent uses the traditional sequential MCTS outlined in Section 3.2. Our `Conc` agent avails itself of multithreading but not multiprocessing, as outlined in Section 3.4. Our `Dist` agent uses the distributed schema outlined in Section 3.6. We use the framework provided by Dsys to ease the development of this `Dist` agent. We implement all agents in c++14. Though there are many ways to tune the MCTS algorithm, all of our agents use UCB1 for Selection with $c \approx \sqrt{2}$. All agents expand nodes randomly, and evaluate the results using 20 random playouts. Our `Seq` agent backpropagates normally, whereas our `Conc` and `Dist` agents use the virtual loss paradigm explored in Section 3.4. All of our tests were run on a cluster with four cores and the ability to run 32 threads per core. All microprocessors are Intel(R) Xeon(R) CPU E5-2620 v4s.

We ran two types of experiments. In the first, we hold rollouts constant at 100000, and see how long each agent takes to perform a given rollout. In these experiments, our results are based on the average time it took to complete a rollout for a given agent over 20 games of Hex on an 11×11 board.

In the second type of experiment, we have two agents play each other in a game of Hex. We give each agent 5 seconds of computational time to make each move. Here, we evaluate the win rates for different agents over 40 games of Hex on an 11×11 board. As explored in Section 2.1, we know that move order is of particular importance in Hex, so each agent moves first in half of the games in order to get a balanced estimate of agent strength.

5.2 Results

Based on the results of 20 games of Hex with 100000 rollouts per move in each game, our `Seq` agent takes 51.03 microseconds on average to perform a rollout. We use this as a benchmark by which to judge the amount of time it takes our `Conc` and `Dist` agents to perform rollouts.

As we see in 5.1, the number of threads of execution has a dramatic impact on how long it takes our `Conc` agent to perform a given rollout. A `Conc` agent with a single thread of execution takes longer to perform a single rollout than our `Seq`

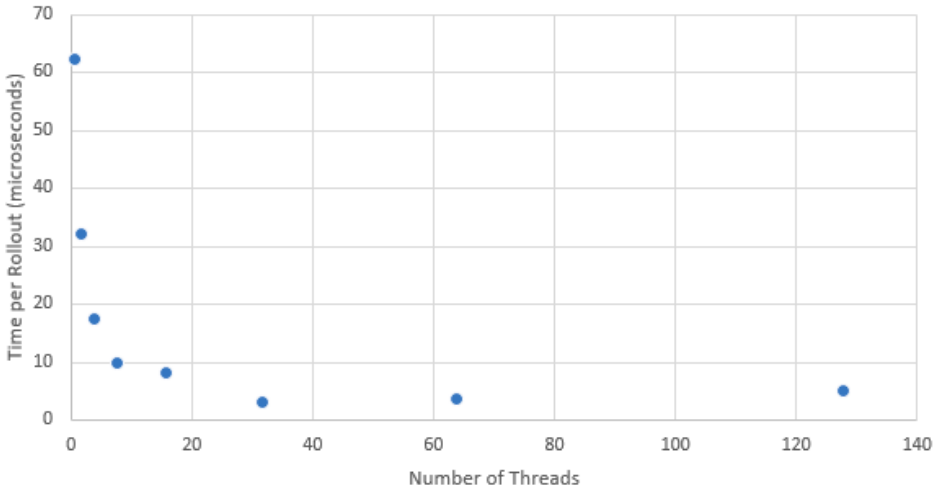

Figure 5.1: The efficiency of `Conc` at performing rollouts as a function of the number of threads of execution afforded to `Conc`

agent. This $\approx 22\%$ increase in the amount of time it takes a single-threaded `Conc` agent when compared to the `Seq` agent to perform rollouts is due to the overhead of using atomic operations, even though no CAS operations fail when there is only a single thread performing rollouts. We see a dramatic decrease in the amount of time it takes to perform rollouts as we increase the number of threads from 1 to 2 to 4 to 8 to 16 to 32. This decrease is very steep at first: two threads take almost half the amount of time to perform rollouts compared to one thread. However, as we increase the number of threads, the dropoff becomes less pronounced. This is due to increased contention in the MCTS tree as more threads attempt to access the same nodes. As CAS operations fail, it naturally takes longer to perform rollouts.

We do not expect the efficiency of the MCTS algorithm to increase beyond 32 threads of execution, which the results in Figure 5.1 confirm. This is because our cluster only has capacity for 32 threads per core. Thus, we see a slight slowdown as we increase the number of threads of execution beyond 32, as now we pay for context switches between threads, wasting CPU cycles when they could be working towards performing MCTS rollouts.

As we see in Figure 5.2, the cost of the framework for the distribution is significant: a single threaded version of `Dist` run on one `MPI` process takes almost three times as long to perform a rollout as `Seq`. In addition, we see that as we increase the number of `MPI` processes on which we run `Dist` from 1 to 2 to 3 to 4, there is a slight increase in the amount of time it takes to perform a rollout. Though now multiple `MPI` processes are coordinating their computational resources on the same problem, the communication between processes means that the time per rollout slightly increases with the number of `MPI` processes. Similarly to `Conc`, as we increase the number of threads on which `Dist` performs rollouts, we see a dramatic decrease in the amount of time it takes to perform a given rollout in Figure 5.3. In this test, we held the number of `MPI` processes constant at 4 and simply varied the number of threads of execution per process. As we double the number of threads, the amount of time it takes to perform a rollout is almost cut in half. Of particular note is that this trend does not continue past 16 threads per `MPI` process. This suggests that there are bottlenecks in the distributed algorithm and that the 128 threads acting

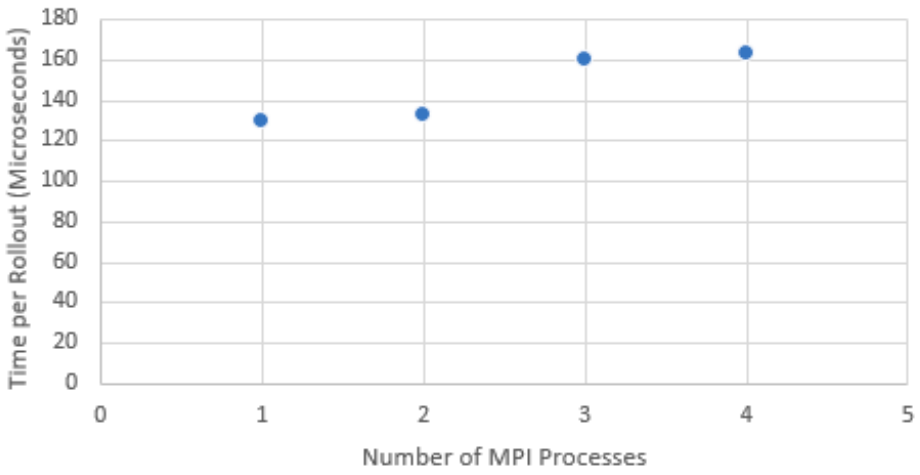

Figure 5.2: The efficiency of `Dist` at performing rollouts as a function of the number of `MPI` processes

on the same MCTS tree resulted in extremely high contention for particularly interesting nodes or that receiver threads became overwhelmed with the amount of communication between different `MPI` processes.

We see the direct impact the additional rollouts any agent can perform has on its strength in Figure 5.4. Where a single-threaded `Conc` agent only wins against `Seq` 37.5% of the time, a `Conc` agent run with 32 threads wins against `Seq` 82.5% of the time. Though there is some noise in the data - for example, the 16-threaded `Conc` agent underperforms the 8-threaded `Conc` agent by a few percentage points, this is likely due to the relatively small sample size of the data, and the broad trend that more rollouts result in an increase in agent strength holds. Of course, additional rollouts do not guarantee that an agent wins a given game, but it does increase the amount of information an agent can get about the arms of the multi-armed bandit. This increase in information increases the probability with which the agent finds better moves. Thus, we see the sizeable impact multithreading can have on agent performance, motivating the need to take advantage of concurrency in these computationally intensive algorithms.

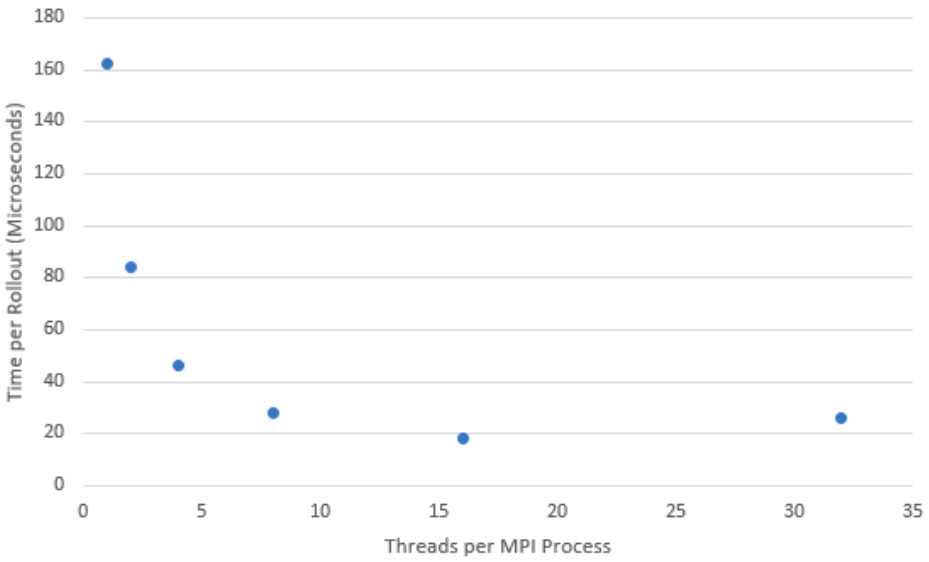

Figure 5.3: The efficiency of `Dist` at performing rollouts as a function of the number of threads per `MPI` process

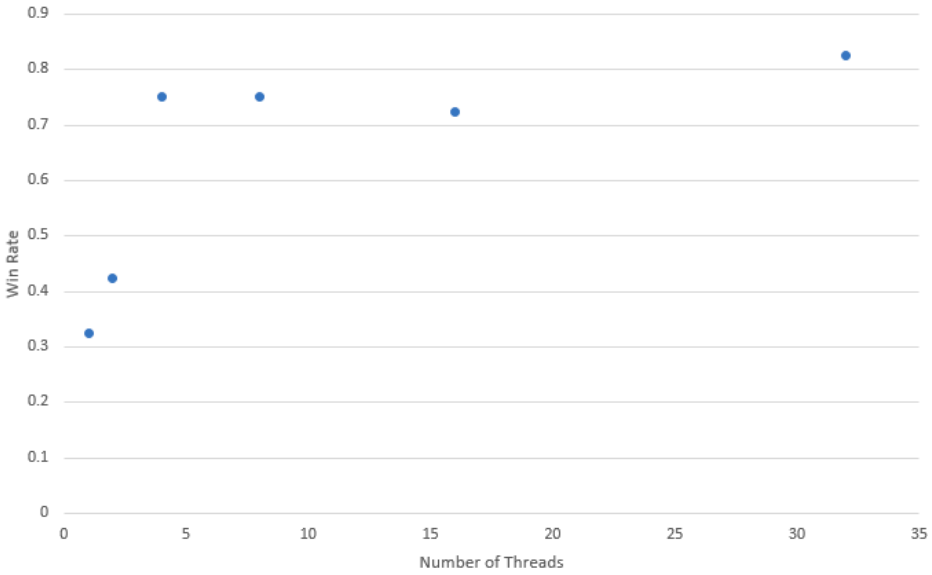

Figure 5.4: The percentage `Conc` wins games of Hex against `Seq` as a function of the number of threads of execution afforded to `Conc`

Chapter 6

Conclusion

6.1 Conclusion

In summation, we developed a distributed MCTS agent that plays the board game Hex. Through harnessing the computational power afforded to us by multi-threading and multi-processing paradigms, we increased the strength of our agent significantly. However, the difficulty of implementing an MCTS agent increases exponentially when we wish to distribute the agent across multiple machines. Distributing MCTS serves to highlight some of the myriad difficulties presented by multi-threading and multi-processing. To ease the burden of building distributed programs, we created a framework in dsys that abstracts many of the difficulties inherent to distributed programming while not forcing developers into an entirely new programming paradigm. Our distributed MCTS agent therefore not only highlights the impact of harnessing multiple machines to the same problem, but validates the utility and flexibility of this distributed framework.

Unfortunately, the time it takes our `Dist` agent to perform a rollout increases as we increase the number of `MPI` processes. In addition, there is a slight increase in the time it takes for our `Dist` agent run on 4 `MPI` processes to perform a rollout when we increase the number of threads per process from 16 to 32. This shows us that we are not sufficiently taking advantage of the power of distributed computing. In particular, we must do more to reduce contention and avoid expensive CAS failures and should consider new schemes to reduce communication between nodes. This communication between nodes creates a costly overhead for the distributed schema and shows that, even if correct, distributed algorithms can execute more slowly than we would expect given the additional resources they are afforded.

Thus, our work is ongoing. Using Dsys and our existing distributed MCTS agent, we are currently exploring additional means by which we can more intelligently distribute the work and take advantage of parallelism in the MCTS algorithm.

As communication between processes takes time, we place a premium on reducing the number of messages that we need to pass between different processes. In addition, as CAS operations failing is expensive, as seen in 5.2, we wish to reduce contention as much as possible in this new distributed MCTS paradigm. Thus, we explore means by which we can minimize this contention around the root node of the tree. We associate nodes in the search tree with particular machines (processors), as pictured in 6.2. This association is done at node creation through a Zobrist hash as described in 3.6, which means there is little rhyme or reason as to which processor

is associated with which tree nodes.

Similarly to the classic producer-consumer problem, we consider two classes of threads and types of work that we might do: threads that we refer to as worker threads and threads that we refer to as service threads. Messages between processors can be understood as requests for work to be done on specific parts of the tree.

There is one service thread per processor and as many worker threads as needed. The service thread responsibilities include the following classes of tasks.

1. Receiving messages from other processors.
2. Sending messages to other processors.
3. Batching messages into work orders.
4. Distributing and scheduling work orders to worker threads.

In addition, if the service thread becomes overwhelmed, which we understand as reaching a certain unacceptable combined number of unread incoming messages and unsent outgoing messages, it can submit a work order to a worker thread that tells the worker thread to perform some of the batching and scheduling. However, the service thread never submits work orders to worker threads telling them to batch messages, as this would be in violation of our guarantee that each worker will have only one producer, the service thread, and one consumer, the worker thread in question, for its queue. This is an important guarantee, as concurrent queues are extremely fast as long as there is only one producer and one consumer, but slow down if there are multiple producers or multiple consumers due to increased contention.

Worker threads are responsible for actually performing the operations in the tree. Each worker thread has its own private work queue. As noted above, a workers' queue has one producer: the service thread, and one consumer: the worker thread, so we expect operations on this queue to take place quickly. Worker threads responsibilities include the following classes of tasks.

1. Performing the UCT selection (selection).
2. Generating new pointers with a new hash (expansion).
3. Running simulations at leaf nodes (simulation).
4. Scheduling work orders.

Finally, in addition to the service and worker threads, there is one thread that signals the start of a rollout.

To illustrate our algorithm, we trace through one rollout in the MCTS algorithm. The service thread in charge of the root node pushes a job to a worker's queue of performing some number of UCT selections on the root's children. The worker in question pops jobs from their queue until they reach the UCT selection task they received from their service thread, and then dutifully perform the requisite selections on the node. They then send a message to the service threads which own each of the selected nodes letting these service threads know that their children have been selected; this process repeats recursively until we reach a leaf node in our MCTS tree.

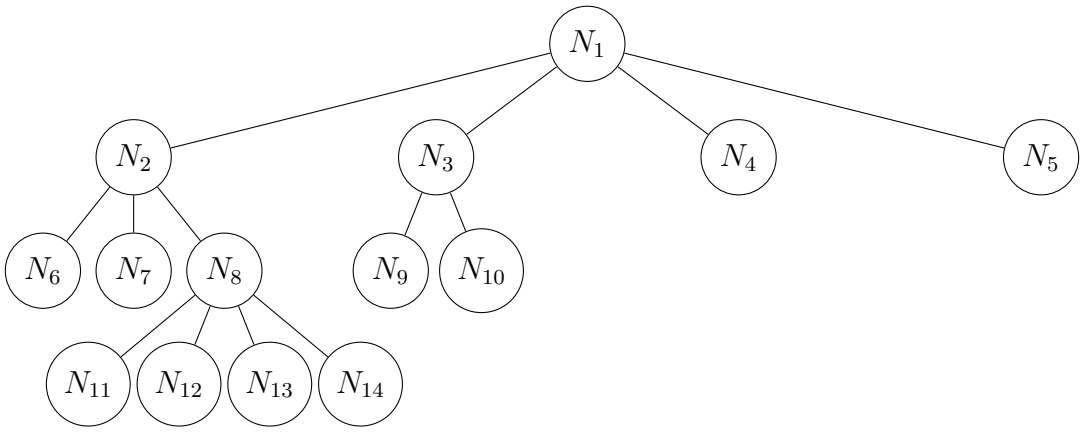

Figure 6.1: A hypothetical Monte Carlo Search Tree

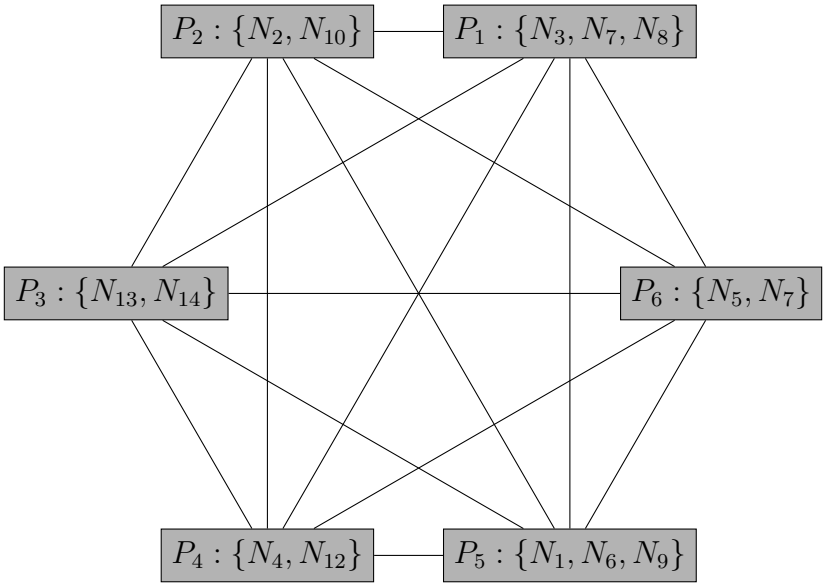

Figure 6.2: An example six-processor network

At this point, the service thread, instead of pushing a job to a worker's queue of performing a UCT selection and traversing down the tree, pushes an expansion job to the worker's queue. This expansion job means that the worker generates a new node in the tree with a new hash and assigns it to a new service thread. When a worker thread completes this task, it signals the service thread owner of this freshly expanded node, who in turn pushes a job to a worker's queue to run a set number of simulations to evaluate this new leaf node. Based on the results of these simulations, service threads communicate with each other back up the tree to re-evaluate the goodness of their respective nodes. Naturally, throughout this process, service threads batch requests for selection and backpropagation. This batching serves to both minimize communication between processors and to reduce the number of times worker threads need to update or query nodes in the tree.

As we saw in our agent `Conc`, CAS operations failing is expensive, and thus we place a premium on reducing contention in the tree so that CAS operations do not fail and we do not waste computation. Thus, it is the responsibility of the service thread to intelligently schedule work orders such that different workers operate on different parts of the tree. Ideally, we thereby avoid the problem of multiple threads

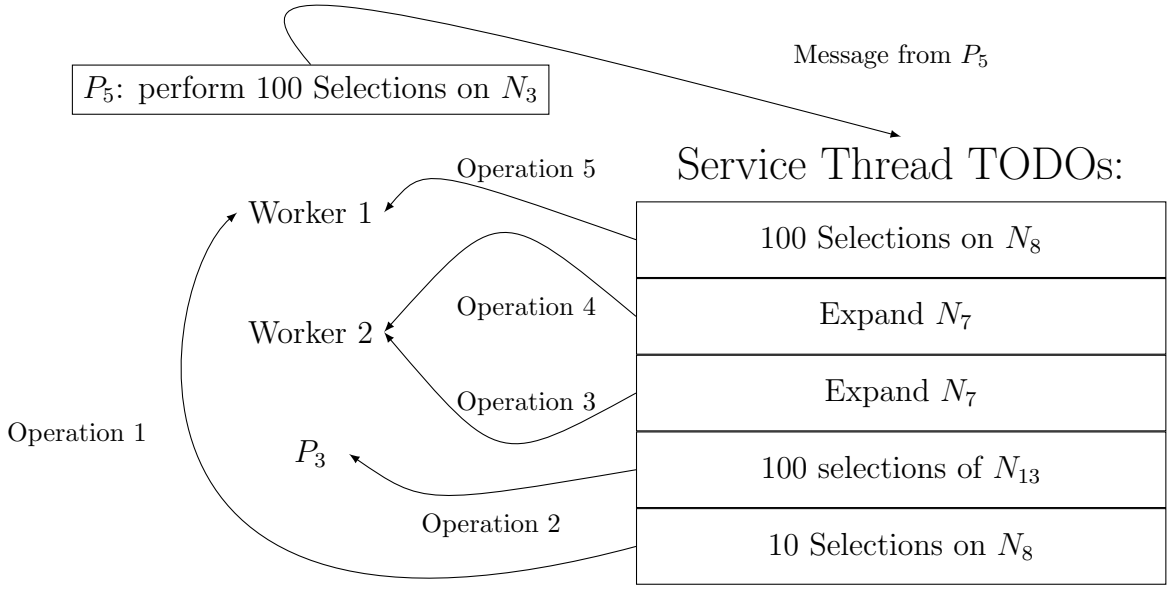

Figure 6.3: Example Service Thread interactions from processor P_1 in 6.2

attempting to modify the same fields of the same node and reduce contention in the tree. That said, our algorithm is still sound even if multiple workers operate on the same node within the tree and there is a minor amount of contention. This is because we use atomic variables to flush caches and ensure that that other threads can see changes to the tree. Without these atomic variables, the changes one thread makes in a given node might not be seen by another thread with the same fields in its cache, and our algorithm would suffer as workers operate on erroneous, old values. Thus, though a CAS operation might fail if multiple workers attempt to change the same node concurrently, our tree and MCTS search algorithm will not become corrupted.

Through avoiding failed CAS operations, our algorithm largely takes full advantage of the resources available to it, so long as the worker queues are reasonably similar in size. Every thread is either working on intelligently scheduling and distributing the work, or performing some work on a given rollout. However, this is not true if the service threads do not maintain queues of similar size. Without similarly sized worker queues, our algorithm's performance suffers as we do not take full advantage of our resources. To ensure that queues are of relatively similar size, if a service thread becomes overwhelmed, it can broadcast a message to other machines asking for another processor to take responsibility for one or more of its nodes. Similarly, if a given service thread has no work, it can broadcast a message to other machines asking for ownership of a node so that it can do some work on the tree. This dynamic load-balancing is particularly important for a successful, efficient MCTS algorithm, as the MCTS search tree can become very imbalanced, as illustrated in 6.1. Different nodes can have vastly different UCT values and thus be visited much more frequently. For example, if we have four computers as in 6.1, and one of them is assigned the node set $S_1 = \{N_1, N_2, N_3, N_8\}$, it clearly will

have significantly more traffic than another processor which is assigned the node set $S_2 = \{N_4, N_5, N_6, N_7\}$. Though we cannot avoid imbalances occurring at the time of node assignment; some processors will necessarily have more interesting nodes than others, we attempt to mitigate these concerns though the dynamic load-balancing as described above. Consider the scenario above. The first processor has one service thread; this service thread may soon become overwhelmed by requests to do work on their part of the tree. Once this service thread's workload reaches some unacceptable threshold, it will broadcast a request that some other processor gain ownership over one of it's nodes. The second processor, with comparatively little to do, will accept ownership of say, N_8 and thereby equalize the amount of work. In addition, we allow for the second processor to steal work from the first processor if it realizes that it is under-worked before the first processor realizes that it is over-worked.

Disparities in the amount of traffic different nodes see are due to the fact that certain game states and their associated nodes can be much more interesting than others, and it is easy for a single processor to gain ownership over the nodes associated with some very good moves while other processors gain ownership over the nodes associated with some quite bad moves. However, through the work-stealing and load-balancing algorithm described above, we avoid many of the problems caused by this imbalance in the tree.

As our system relies on message passing between different processors, rather than simply wasting the time used for message passing, we can overload the processors, as described in [21]. Here, we schedule more rollouts than our nodes have time to schedule, because we know that some of the time will be spent not performing actual computation but rather in limbo, waiting for messages. The amount we overload our processors is therefore a hyperparameter we must tune.

Thus, future work includes an even more efficient distributed MCTS agent that uses a producer-consumer paradigm to ensure that all threads are constantly working towards the shared goal of performing rollouts, with reduced communication between processors, fewer CAS fails, and batched updates to nodes so as to reduce operations on the tree.

Though there is much work that can be done in order to improve on our distributed MCTS agent, our existing distributed MCTS agent successfully leverages the processing power of multiple machines and multiple threads on different machines, and thus is significantly stronger than its sequential counterparts. In addition, it serves to validate our Dsys framework as a powerful tool for distributed developers.

Bibliography

[1] Broderick Arneson, Ryan B Hayward, and Philip Henderson. "Solving hex: beyond humans". In: *International Conference on Computers and Games*. Springer. 2010, pp. 1–10.

[2] Pavan Balaji et al. "Fine-grained multithreading support for hybrid threaded MPI programming". In: *The International Journal of High Performance Computing Applications* 24.1 (2010), pp. 49–57.

[3] Yngvi Björnsson et al. "Dead cell analysis in Hex and the Shannon game". In: *Graph Theory in Paris*. Springer, 2006, pp. 45–59.

[4] Bruno Bouzy and Bernard Helmstetter. "Monte-carlo go developments". In: *Advances in computer games*. Springer, 2004, pp. 159–174.

[5] Cameron B Browne et al. "A survey of monte carlo tree search methods". In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43.

[6] Giuseppe Burtini, Jason Loeppky, and Ramon Lawrence. "A survey of online experiment design with the stochastic multi-armed bandit". In: *arXiv preprint arXiv:1510.00757* (2015).

[7] Tristan Cazenave and Nicolas Jouandeau. "On the parallelization of UCT". In: *proceedings of the Computer Games Workshop*. Citeseer. 2007, pp. 93–101.

[8] Guillaume MJ-B Chaslot, Mark HM Winands, and H Jaap van Den Herik. "Parallel monte-carlo tree search". In: *International Conference on Computers and Games*. Springer. 2008, pp. 60–71.

[9] David B Fogel. *Blondie24: Playing at the Edge of AI*. Elsevier, 2001.

[10] David Gale. "The game of Hex and the Brouwer fixed-point theorem". In: *The American Mathematical Monthly* 86.10 (1979), pp. 818–827.

[11] Tobias Graf et al. "Parallel Monte-Carlo tree search for HPC systems". In: *European Conference on Parallel Processing*. Springer. 2011, pp. 365–376.

[12] William Gropp et al. "A high-performance, portable implementation of the MPI message passing interface standard". In: *Parallel computing* 22.6 (1996), pp. 789–828.

[13] Philip Henderson, Broderick Arneson, and Ryan B Hayward. "Hex, braids, the crossing rule, and XH-search". In: *Advances in Computer Games*. Springer. 2009, pp. 88–98.

[14] Philip Henderson, Broderick Arneson, and Ryan B Hayward. "Solving 8x8 Hex". In: *Twenty-First International Joint Conference on Artificial Intelligence*. 2009.

[15] Shih-Chieh Huang et al. "MoHex 2.0: a pattern-based MCTS Hex player". In: *International Conference on Computers and Games*. Springer. 2013, pp. 60–71.

[16] Levente Kocsis and Csaba Szepesvári. "Bandit based monte-carlo planning". In: *European conference on machine learning*. Springer. 2006, pp. 282–293.

[17] Michael McCarver and Rob LeGrand. "Evolving a Hex-Playing Agent". In: *CRIUS* 4 (2018).

[18] John F Nash. "Some games and machines for playing them". In: (1952).

[19] Jacob Nelson et al. "Latency-Tolerant Software Distributed Shared Memory". In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, July 2015, pp. 291–305. ISBN: 978-1-931971-225. URL: https://www.usenix.org/conference/atc15/technical-session/presentation/nelson.

[20] Pascutto, Gian-Carlo and Linscott, Gary. *Leela Chess Zero*. Version 0.21.0. Mar. 8, 2019. URL: http://lczero.org/.

[21] Lars Schaefers, Marco Platzner, and Ulf Lorenz. "UCT-treesplit-parallel MCTS on distributed memory". In: *Proc. 21st Int. Conf. Automat. Plan. Sched., Freiburg, Germany*. Citeseer. 2011.

[22] Claude E Shannon. "XXII. Programming a computer for playing chess". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41.314 (1950), pp. 256–275.

[23] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), p. 484.

[24] *Stockfish Github*. https://github.com/official-stockfish/Stockfish.

[25] István Szita, Guillaume Chaslot, and Pieter Spronck. "Monte-carlo tree search in settlers of catan". In: *Advances in Computer Games*. Springer. 2009, pp. 21–32.

[26] Guy Van den Broeck, Kurt Driessens, and Jan Ramon. "Monte-Carlo tree search in poker using expected reward distributions". In: *Asian Conference on Machine Learning*. Springer. 2009, pp. 367–381.

[27] H Jaap Van Den Herik, Jos WHM Uiterwijk, and Jack Van Rijswijck. "Games solved: Now and in the future". In: *Artificial Intelligence* 134.1-2 (2002), pp. 277–311.

[28] François Van Lishout, Guillaume Chaslot, and Jos WHM Uiterwijk. "Monte-Carlo tree search in backgammon". In: (2007).

[29] Wikipedia contributors. *Hex (board game)*. [Online; accessed 26-October-2019]. 2019. URL: https://en.wikipedia.org/wiki/Hex_(board_game)#Automatons.

[30] Kazuki Yoshizoe et al. "Scalable distributed monte-carlo tree search". In: *Fourth Annual Symposium on Combinatorial Search*. 2011.

[31] Kenny Young, Gautham Vasan, and Ryan Hayward. "Neurohex: A deep q-learning hex agent". In: *Computer Games*. Springer, 2016, pp. 3–18.