Scalable Distributed Monte-Carlo Tree Search

Kazuki Yoshizoe⋆, Akihiro Kishimoto†, Tomoyuki Kaneko⋆, Haruhiro Yoshimoto⋆, Yutaka Ishikawa⋆
⋆The University of Tokyo, †Tokyo Institute of Technology and Japan Science and Technology Agency

Abstract

Monte-Carlo Tree Search (MCTS) is remarkably successful in two-player games, but parallelizing MCTS has been notoriously difficult to scale well, especially in distributed environments. For a distributed parallel search, transposition-table driven scheduling (TDS) is known to be efficient in several domains.

We present a massively parallel MCTS algorithm, that applies the TDS parallelism to the Upper Confidence bound Applied to Trees (UCT) algorithm, which is the most representative MCTS algorithm. To drastically decrease communication overhead, we introduce a reformulation of UCT called Depth-First UCT. The parallel performance of the algorithm is evaluated on clusters using up to 1,200 cores in artificial game-trees. We show that this approach scales well, achieving 740-fold speedups in the best case.

Introduction

The $\alpha\beta$ algorithm (Knuth and Moore 1975) achieves strength at least comparable to a human champion for most two-player games. However, $\alpha\beta$ is ineffective for the game of Go because making a good evaluation function for Go is difficult. Monte-Carlo Tree Search (MCTS) has quickly become famous for its remarkable success in developing strong Go programs (Coulom 2006; Gelly et al. 2006; Enzenberger et al. 2010). The Upper Confidence bound applied to Trees (UCT) algorithm (Kocsis and Szepesvári 2006) best represents MCTS, effective not only in other games (Lorentz 2008; Sturtevant 2008) but also in security evaluations of biometric authentication systems (Tanabe, Yoshizoe, and Imai 2009).

UCT can improve the quality of outcomes by constructing a larger game tree (called a *UCT tree*) and increasing the number of Monte-Carlo simulations (called *playouts*) performed at leaf nodes of the UCT tree. A playout is performed in the most valuable leaf determined by traversing the current UCT tree in a best-first manner, and the result of the playout is propagated back to update the UCT tree. In this way only the promising part of the UCT tree, which contains the best line of plays for both players, grows deeper. Having a deeper tree in the most promising part is crucial for the quality of the outcomes of search algorithms.

UCT must respond in real-time in many applications, including games. Parallelization is one way to improve the quality of solutions yielded by UCT in a fixed amount of time. In fact, promising results have been reported for when UCT is parallelized on shared-memory systems with a small number of CPU cores (up to 16), where a UCT tree can be efficiently shared among processors (Chaslot, Winands, and van den Herik 2008; Enzenberger and Müller 2010).

Distributed computing enables UCT to gain much better performance because of the availability of larger CPU and memory resources. For efficient parallelization, the UCT tree should be shared among processors because UCT decides on a promising search direction based on previous computations recorded in the tree. However, efficiently sharing the UCT tree is notoriously difficult due to communication delays over the network. Existing work on distributed UCT (Gelly et al. 2008) focuses only on increasing the number of playouts, at the cost of performing useless playouts. The existing methods therefore fail in growing the UCT tree compared to its sequential counterpart, making these methods difficult to scale well with a large number of cores. For example, Segal's sequential simulation of parallel UCT, in which the UCT tree can be shared without any overhead among processors, showed the promise of massive parallelism with 512 CPU cores. However, his implementations of distributed algorithms reported in the previous literature did not achieve any meaningful scaling behavior beyond 32 cores in practice (Segal 2010).

We present a distributed UCT algorithm that can drastically grow both the UCT tree size and the number of playouts. Our approach incorporates transposition-table driven scheduling (TDS) (Romein et al. 1999), which has been successfully applied to various search algorithms (Kishimoto and Schaeffer 2002; Kishimoto, Fukunaga, and Botea 2009). We achieved both efficient sharing of a UCT tree and almost uniform load balancing by using the TDS idea. However, this straightforward combination of UCT with TDS suffers from severe communication overhead incurred by the processor managing the root node of the UCT tree. We therefore introduce Depth-First UCT (df-UCT), a reformulation of UCT in a depth-first manner that practically relaxes the dependency of each search inside the UCT tree. In this way, our distributed df-UCT synthesized with TDS can drastically reduce the communication overhead compared to the version combining UCT with TDS.

The performance of our distributed algorithms was evaluated for up to 1,200 CPU cores in artificial game trees, a representative domain used to investigate the performance of game-tree search algorithms. Our results show that the depth-first reformulation enables our distributed df-UCT to scale very well, while a naive combination of UCT and TDS has limited scalability.

Related Work

Monte-Carlo Tree Search and UCT

MCTS (Kocsis and Szepesvári 2006) has recently had notable success, especially for the game of Go, where $\alpha\beta$ has failed to perform well. MCTS has become the *de facto* standard technique for developing the strongest computer Go programs (Coulom 2006; Gelly et al. 2006; Enzenberger et al. 2010).

MCTS yields solutions by aggregating the results of Monte-Carlo simulations, called *playouts*, performed at the leaf nodes of a search tree. In a playout starting at a node p, each player keeps playing one of the legal moves with randomness [1] until reaching a terminal position where the winner of the game is determined by the rules of the game. The outcome of the playout at p is usually defined as the value (win or loss) of that terminal position. For simplicity, draws are ignored in this paper.

Better solutions tend to be obtained when more playouts are performed at important leaf nodes, while the importance of each branch itself must be estimated by playouts. UCT handles such trade-offs with a simple formula (Auer, Cesa-Bianchi, and Fischer 2002). It performs a playout at the most valuable leaf, decided by traversing the current search tree in a best-first manner, and updates the winning probabilities of the branches in the current path with the result of the playout (see next section for details). Thus, UCT gradually grows its search tree where important branches tend to have deep subtrees.

UCT is guaranteed to converge to an optimal policy with an infinite number of iterations (Kocsis and Szepesvári 2006). However, due to many enhancements adopted for empirical strength (Gelly and Silver 2009; Coulom 2007; Gelly et al. 2006), practitioners often accept the possibility of breaking the assumptions of such a theoretical guarantee. A variety of enhancements are available, but plain UCT with no enhancements was used in our reported experiments for simplicity. Additionally, UCT does not always return the same solution for an identical problem with different random seeds. In fact, this phenomenon is often observed in UCT-based computer Go programs.

Parallel Monte-Carlo Tree Search

Many researchers have parallelized MCTS to improve the quality of outcomes yielded in a fixed amount of time. To achieve parallelism, CPUs must start a search before knowing the results of the latest search. If all the CPUs wait for the latest results, no place could be executed in parallel. Therefore, note that any parallel MCTS may return different outcomes to the sequential version. Parallelizing MCTS is still important because its performance is improved in practice.

In shared-memory environments, MCTS is effectively parallelized with small modifications to the sequential version. *Tree parallelization* is the most promising approach incorporated into many computer Go programs (Chaslot, Winands, and van den Herik 2008; Enzenberger and Müller 2010; Gelly et al. 2006). Each processor independently traverses a shared tree in parallel to find a leaf node to perform a playout. On the basis of the playout result, the statistical information stored in the tree is updated. Lock-free structures have been used (Enzenberger and Müller 2010) to reduce the overhead of mutex locks, which is usually unavoidable in tree search. We parallelize UCT on the basis of tree parallelization for distributed-memory environments because of the popularity of this approach.

Coulom's virtual loss is crucial for making processors traverse different but promising parts of the tree in tree parallelization (Chaslot, Winands, and van den Herik 2008). A processor selecting a branch to trace down to find a leaf pessimistically decreases the upper confidence bounds (UCB) value of the branch by assuming that one playout results in a loss, thus encouraging other processors to select other moves. The UCB value is corrected after the actual playout result is propagated back.

In distributed-memory environments, efficient implementation of tree parallelization is a serious issue (Segal 2010; Bourki et al. 2010). Processors do not share memory, so one processor cannot access the trees managed by the others without incurring communication overhead. The most successful approach so far uses tree parallelization where memory can be shared and periodically broadcasts only an "important" part of the local tree among machines where memory is distributed (Gelly et al. 2008). However, in the rest of the tree, playout results are independently accumulated on different machines and useless playouts are performed. This approach also cannot share virtual losses efficiently, possibly resulting in limited scalability. Our approach, in contrast, can efficiently share both a UCT tree and virtual losses over distributed machines.

Other reported approaches include *root parallelization* and *leaf parallelization* (Chaslot, Winands, and van den Herik 2008). However, both these approaches cannot build a larger search tree compared to sequential MCTS. In root parallelization, the performance gain is less successful than tree parallelization with an extensive analysis of up to 64 processors (Soejima, Kishimoto, and Watanabe 2010). The performance improvement of leaf parallelization is less than that of root parallelization (Cazenave and Jouandeau 2007).

Transposition-table Driven Scheduling

TDS (Romein et al. 1999) simultaneously addresses the issues of load balancing and sharing of the search results, which are crucial in many state-of-the-art algorithms including iterative deepening A* (IDA*) (Korf 1985). Romein

[1] Certain moves are rarely considered as candidates for playouts. For example, in Go, a move filling in an eye point is prohibited in playouts because it is usually an extremely bad move.

et al.'s IDA* implementation utilizes a *transposition table*, which is a large hash table that preserves results of previous search efforts by mapping nodes to entries. Thus, information on nodes repeatedly visited can be reused by simply retrieving it from the corresponding table entries.

In TDS, each processor manages a disjoint subset of the transposition table entries. A hash function maps each table entry exactly to one processor. That is, one large transposition table can be seen as being divided among all the processors. This distribution indicates that the size of the table scales well with the number of distributed machines by increasing the total amount of available memory.

Whenever searching a node n, TDS distributes the job n to the processor that keeps the table entry of n so that the processor receiving n can locally check the entry. Each processor periodically checks if a new node has arrived. TDS might suffer from performance degradation caused by a large communication overhead, which can be intuitively understood because generated nodes are almost always sent to other processors. However, Romein et al. showed that this is not a significant concern, because all communications are *asynchronous*. Once a processor sends out a generated node, it can immediately work on another task without waiting for messages sent by other processors, thus overlapping communication and computation.

The hash function plays an important part in uniformly distributing the work across processors. In two-player games, the Zobrist function (Zobrist 1970) is commonly used to achieve good load balancing.

Romein et al.'s IDA* implementation based on TDS achieved almost linear or even super-linear speedups with 128 processors. The TDS idea was later applied to other domains such as $\alpha\beta$ search (Kishimoto and Schaeffer 2002), and optimal planning (Kishimoto, Fukunaga, and Botea 2009).

Sequential UCT and Depth-First UCT

This section briefly explains the sequential UCT algorithm and then introduces our Depth-First UCT (df-UCT) algorithm that turns UCT into a depth-first search.

UCT

UCT is an anytime algorithm that performs as many iterations as possible until the time has come to play a move for a given position. In each iteration, UCT performs a playout, where a game is played out by selecting moves with randomness, as well as incrementally building a search tree in memory. UCT first traverses a tree starting from the root by selecting the most important move until it reaches a leaf node. The importance of a move is evaluated according to its *UCB value*, explained in the next paragraph. UCT then expands that leaf if necessary and performs a playout there. Next, the playout result is propagated along the path back from the leaf to the root and the UCB values of all the branches on the path are updated.

The UCB value of a branch i (i.e., a move in games) is defined as $\mathrm{UCB}(i) = \dfrac{w_i}{s_i} + c\sqrt{\dfrac{\ln t}{s_i}}$ where c is a constant, s_i is the number of visits to a child i, t is the total number of visits to all children, and w_i is the number of wins accumulated from the results of playouts at child i and its descendants.

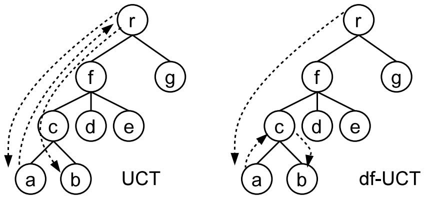

Figure 1: Leaf node selection and result propagation

Move i with high winning ratio $\frac{w_i}{s_i}$ must be selected to have a higher chance to win a game (exploitation). However, move j with a lower winning ratio should be selected if the winning ratio is inaccurate due to small s_j (exploration). The exploration is controlled by the second term of the UCB value. The constant c is $\sqrt{2}$ in theory, but it is usually adjusted in a domain-dependent way.

The behavior of UCT is illustrated in Fig. 1. Branches are sorted such way that the left-most branch is the best. When UCT finishes a playout at leaf a, it increments w_i (if necessary) and s_i of all the branches located between the root r and a (i.e., backed up $a \rightarrow c$, $c \rightarrow f$, and $f \rightarrow r$). Then, from the root node, UCT recursively follows downward the branch with the highest UCB value until it reaches a leaf node, where a new playout will be performed. A leaf is expanded when the number of visits reaches a preset *threshold*. In our experiments, the threshold was set to 4, since typical computer Go programs set it to larger than 1.

Finally, when it is time to play a move, the move with the largest number of visits is typically selected in computer Go. At first glance, selecting a move either with the largest UCB value or with the highest winning probability seems to be the best choice, but this selection is often unreliable if there is a much smaller number of visits to the selected move.

To improve performance, it is essential to build a larger tree by repeating the above steps and concentrating on performing playouts at more promising parts of the search space. For games, performing more playouts contributes to more accurate estimation of the winning ratios of nodes.

Depth-First UCT

A best-first search can often be turned into a depth-first search to preserve the merits of both approaches. Our df-UCT algorithm reformulates UCT in a depth-first manner. Df-UCT has the advantage of revisiting interior nodes less frequently. This reduced revisiting contributes to drastically reducing communication overhead in distributed environments for parallel df-UCT (see the following section and experimental results).

The idea behind df-UCT is based on an observation that once UCT finds a significantly promising part of the search space, playouts are most frequently performed around that part. However, UCT always propagates back a playout result to the root in each iteration to restart the procedure of selecting a new leaf from the root (left of Fig. 1). Df-UCT, in contrast, immediately traces down a tree at an interior node n without propagating back the playout result to the root if n is still on the most promising path. Thus, it explores the most promising search space as long as possible, while it can select the same leaf nodes chosen by UCT in most cases. On the right of Fig. 1, the left-most branch in each level has a larger UCB value than other sibling branches. When df-UCT backtracks to c from a, it checks whether both UCB$(r \rightarrow f) \geq$ UCB$(r \rightarrow g)$ and UCB$(f \rightarrow c) \geq$ UCB$(f \rightarrow d)$ hold. If both conditions still hold, df-UCT immediately moves to b. Unlike UCT, df-UCT does not unnecessarily update the UCB values on path $c \rightarrow f \rightarrow r$ or trace down path $r \rightarrow f \rightarrow c$ again. Returning to r is delayed until $r \rightarrow g$ becomes the best.

Df-UCT manages a local stack as in the standard depth-first search. Assume that df-UCT is currently at node n, and let the best and second-best moves be m_p and m_q, respectively. When df-UCT moves to n's child reached by m_p, it pushes to the local stack the numbers of wins for m_p and m_q, the numbers of visits to m_p and m_q, and the number of visits to n. They are used to check if df-UCT is required to backtrack to an ancestor by recomputing the UCB values.

When a playout is finished at node x, df-UCT examines its local stack. If UCB$(m_p) <$ UCB(m_q) holds for any ancestor n of x that is saved in the local stack, df-UCT proves that the most promising move is changed to m_q from m_p at node n. Therefore, it must return to n to select m_q by popping the appropriate information from the stack.

In our current implementation, moves that are inferior to the second-best one are ignored and not saved in the local stack. In theory, such inferior moves might become the best after the UCB values are recomputed because of the second term in UCB(i). In principle, if df-UCT's local stack manages all the branches of nodes located on the current path, this problem can be safely avoided at the cost of extra memory and computation overhead for managing the stack. However, such a scenario does not occur frequently in our experiments on artificial game trees. Investigating a trade-off in practice (e.g., storing the second- and third-best moves to further reduce such an error) is an important direction for future work.

Distributed UCT

We first explain a distributed algorithm integrating TDS with UCT and then replace UCT with df-UCT.

Distributed UCT Based on TDS

Our distributed UCT leverages TDS to efficiently share a UCT tree among processors over the network. In other words, UCT trees are partitioned over processors in a disjoint way and can be seen as one large UCT tree.

Our distributed UCT inherits the TDS strategy in which the work must always be moved to the processor where its corresponding data is located. In our implementation, each disjoint subset of the entire UCT tree is preserved as a hash table of each processor. The Zobrist hash function (Zobrist 1970) calculates the hash key of a node in the tree. When generating a node n, our distributed UCT must find the processor that has the corresponding hash table entry of n (called the *home processor*) and then move n there. The ID of the *home processor* is generated by a simple calculation using the hash key. Once n is sent there, its table entry can be accessed locally with n's hash key.

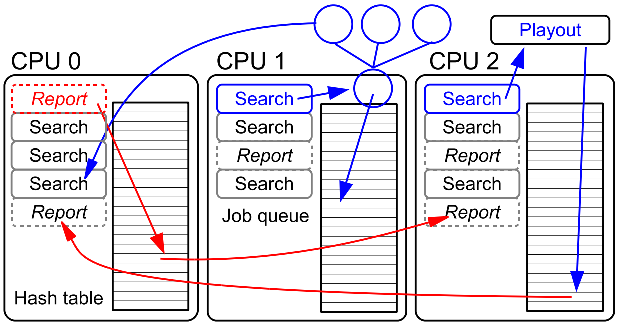

Figure 2: Distributed UCT combined with TDS

Unlike TDS, each processor in our distributed UCT creates *search jobs* and *report jobs* (Fig. 2). A search job is created when the algorithm determines the best move at the current node n and generates its child n_c. A search job for n_c must be sent to n_c's home processor. This step is recursively applied until leaf node is found where a playout is performed locally. A report job is created to propagate back a playout result to update the UCB value. The home processor of n_c sends a report job to n's home processor P. P receives the result of the job, updates the UCB value with the result, and sends another report job upward. This step is repeated until the root receives the job and one iteration (in the sense of sequential UCT) is completed.

The procedures of selecting a leaf and back-propagating its playout result are represented as a recursive function in sequential UCT. In our distributed UCT, each processor has a job queue to manage search and report jobs. It (1) periodically checks if new jobs have arrived, (2) pushes a received job to the queue, and (3) pops one job from the queue to do an appropriate task (i.e., search or report). These procedures are implemented as simple a while-loop (see TDS_UCT in Fig. 3). Our distributed UCT inherits all the advantages of TDS, including asynchronous communication that enables overlap of communication and computation, and almost-uniform load balancing.

For all the processors to be busy, the number of jobs J spawned at the root must be greater than the number of processors N. J is set to $20N$ in our implementation. Segal's sequential simulation of parallel UCT (Segal 2010) shows the strength degradation for large J. We believe that our distributed UCT behaves similarly in theory, although the parallel overheads should be taken into account in practice.

Virtual losses are incorporated into our distributed UCT because they play an important role in distributing search jobs near the promising search spaces explored in shared-memory parallel environments (Segal 2010). Due to the advantage of TDS, the virtual loss information of node n is always available at n's home processor. Thus, by accessing n's hash table entry only locally, our distributed UCT can

```
DistributedUCT(tree_node root, time allotted, int nparallel) {
  start TDS_UCT() in all processors
  foreach (1..nparallel) {
    nj = Job(Type:SEARCH, Node:root)
    SendJob(HomeProcessor(root), nj)
  }
  while (elapsed() < allotted) {
    if (ReceiveJob()!=null) {
      nj = Job(Type:SEARCH, Node:root)
      SendJob(HomeProcessor(root), nj)
    }
  }
  stop all running jobs and gather all results to root node
}

TDS_UCT () {
  while (!Finished()) {
    if ((job = ReceiveJob())!=null) EnqueueJob(job)
    // received a job from another processor
    if ((job = PopJobQueue())==null) continue
    tree_node n = job.Node
    entry e = LookUpHashTable(n)
    // statistics are stored in local memory
    if (job.Type == SEARCH) {
      if (e.t < threshold) {
        player winner = doPlayout()
        e.t+=1; WriteToHashTable(e)
        nj = Job(Type:REPORT, Winner:winner, Node:n.parent, Child:n)
        SendJob(HomeProcessor(n.parent), nj)
      } else {
        int i = find move id with the highest UCB value
        e.t+=1; WriteToHashTable(e) // virtual loss
        nj = Job(Type:SEARCH, Node:n.child[i])
        SendJob(HomeProcessor(n.child[i]), nj)
      }
    }
    else if (job.Type == REPORT) {
      player winner = job.Winner
      int i = n.indexOfChild(job.Child)
      e.s_i += 1, e.w_i += (n.turn == winner) ? 1 : 0
      WriteToHashTable(e)
      nj = Job(Type:REPORT, Winner:winner, Node:n.parent, Child:n)
      SendJob(HomeProcessor(n.parent), nj)
    }
  }
}
```

Figure 3: Pseudo code of distributed UCT

obtain the exact number of processors currently marking the virtual losses.

Assume that the number of search jobs descending from child i is d_i. The original UCB value definition is modified as follows, where $d_T = \sum_{j \in i's}\text{siblings}\ d_j$. Considering virtual loss, UCB(i) decreases as more processors select i, which makes other processors prefer other variations.

$$\text{UCB}(i) = \frac{w_i}{s_i + d_i} + c\sqrt{\frac{\ln(t + d_T)}{s_i + d_i}} \qquad (1)$$

Distributed Depth-First UCT

One drawback of combining UCT with TDS is the excessive amount of communication, especially to the home processor of the root. This issue becomes more serious for a larger number of processors. For example, assume that the average search depth of the UCT tree is d in N iterations, and that search and report jobs must always be sent to another processor whenever they are created. This assumption is reasonable for massive parallelism, since the probability of the TDS strategy assigning node n's child to n's home processor is low with a large number of processors. The total number of messages received by all the processors is approximately $2dN$. Moreover, since each iteration always starts at the root, the home processor of the root must send or receive at least N jobs. When N becomes large (6–20 million in our experiments), the number of messages exchanged is very unbalanced among processors.

Df-UCT overcomes this problem by reducing the frequency of propagating back a playout result. On average, df-UCT has a shorter path traced back to update the UCB value than UCT. Additionally, the frequency of df-UCT backtracking to the root is much smaller than the number of iterations N. Compared to distributed UCT, distributed df-UCT greatly reduces the number of messages exchanged at the home processor of the root with large N.

Unlike distributed UCT, explained in the last subsection, the distributed df-UCT algorithm based on TDS must decide whether to trace down the most promising child of node n by spawning a search job for the child or to send a report job to the home processor of n's parent, when a report job for n is received. Df-UCT makes this decision by checking its local stack. Therefore, when a search job is created, distributed df-UCT must send the complete information on the local stack as well. This additional information can increase the communication overhead. In our implementation, the job packet size for naive distributed UCT was approximately 100 bytes, while it increased to 2 or 3 kilobytes for distributed df-UCT. However, this is a small price to pay to obtain drastic speedups (see the next section).

Several nodes are packed into one message to send in (Romein et al. 1999), but this is not the case in TDSAB, a parallel $\alpha\beta$ search based on TDS (Kishimoto and Schaeffer 2002), due to the importance of preserving the search order such that the most promising nodes must be expanded as soon as possible. Our scenario is very similar to that of TDSAB. Promising search jobs must be executed immediately, while the UCT tree must be updated to the latest as quickly as possible. It is therefore difficult to leverage Romein et al.'s technique.

One small modification specific to our distributed df-UCT is that only the second-best move takes the virtual loss information into account when each processor recomputes the UCB values of the best and second-best moves saved in the local stack of df-UCT. This technique slightly decreased the number of report jobs.

Experimental Results

Setup of Experiments

To show the scalability of our distributed algorithms, experiments were conducted on artificial game trees called P-game (Smith and Nau 1994). We prepared three algorithms: (1) UCT1: UCT + TDS, (2) UCT2: a version of UCT1 modified by assigning one special processor working only on exchanging messages on the root node, and (3) DFUCT: Df-UCT + TDS. For a P-game tree, we use virtual playouts that merely wait without doing anything for a certain time period. We tested two kinds of playout granularity (0.1 and 1.0 millisecond (ms)), corresponding to those performed by typical Go programs with the board sizes of 9×9 and 19×19, respectively. We measured the elapsed time required until each version finished performing a fixed number of playouts per problem (20 million for the 0.1-ms playout and 6 million for the 1-ms playout). Since DFUCT does not always return to the root due to its depth-first reformulation, it does not have the latest information on the total number of playouts performed by all the processors. We therefore ran DFUCT until each processor performed at least $\lceil 1.05 \times \frac{K}{N} \rceil$ playouts, where K is the number of playouts and N is the number of processors. This resulted in DFUCT performing more playouts than the other two versions, which slightly underperforms the expected speedup value of DFUCT[2].

We tested three cases of uniform branching factors (8, 40, and 150). The branching factors of 40 and 150 are similar to the average branching factors in Go with the board sizes of 9×9 and 19×19, respectively. We tested two playout granularities for each branching factor, resulting in a total of six categories. Ten problems were generated, and the runtime was averaged over all problems in each category.

All experiments were run on a cluster of machines where each compute node was interconnected by QDR Infiniband x 2 (80 Gbps) and had two hexa core Xeon X5670 processors at 2.93 GHz (12 CPU cores) with 54 GiB memory. We used at most 100 compute nodes. All the algorithms were parallelized using an implementation of the Message Passing Interface library. When N CPU cores were used, N MPI processes were invoked and communicated by MPI operations even if they were running on the same compute node.

Artificial Game Tree

P-game models two-player games and is commonly used to evaluate search algorithms (Kocsis and Szepesvári 2006). A random value is assigned to each branch of P-game's tree, and the score of a node is defined as the sum of the branch values in the path from the root to that node. In our implementation, a uniform distribution was used to compute each branch value range of $[-255, 0]$ for the first player and $[0, 255]$ for the second. For all nodes, the value of 0 was assigned to one branch to set the true min-max value of the tree to 0. In P-game, more promising moves tend to be available for a player once that player gains the advantage, as occurs in actual games. In our implementation, after a specified elapsed time, a playout at node N returned either 1 or 0 with a probability defined by the score of N via sigmoid function $1/(1 + e^{-score(N)/128})$. The probability of the playout result of 1 becomes higher for high scores.

Table 1: Sequential search execution time (seconds)

	0.1-ms playout			1.0-ms playout		
Br. factor	8	40	150	8	40	150
DFUCT	2,108	2,129	2,243	6,027	6,034	6,068
UCT	2,303	2,365	2,847	6,068	6,074	6,200

Sequential Search Performance

We compared the performances of sequential UCT and DFUCT, since the best sequential algorithm must be chosen as a baseline when measuring speedups of parallel algorithms. Table 1 shows the execution time, which includes the time for playout execution and for other operations such as management of a UCT tree. Both UCT and DFUCT spent the same amount of time in playout execution: 2,000 s for the 0.1-ms playouts and 6,000 s for the 1.0-ms playouts. As the table shows, DFUCT was faster than UCT in all the categories. The performance difference increased with a larger branching factor and faster playout speed. For example, when the playout time was set to 0.1 ms, 15–40 % of the execution time spent by UCT was related to managing the UCT tree, such as its hash table accesses and traversing the tree. Due to its unnecessary backtracking, UCT was slower than DFUCT by 22% in the worst case.

As explained previously, DFUCT occasionally performs a playout at a different leaf than UCT. Such cases occur when a move with the UCB value that is smaller than the second-best one becomes the best move after a playout. Table 2 shows that only 1–2 % of the UCT backups were such a case.

To check whether the above difference in behavior between DFUCT and UCT is acceptable, we evaluated the strengths of these algorithms by holding 400-game matches for branching factors of 8, 40, and 150. Each 400-game match was played as follows. First, 20 P-game trees were prepared with limited depths. For branching factors of 8, 40, and 150, the depths were limited to 30, 20, and 10, respectively. On each P-game tree, DFUCT played games against UCT with ten different seeds and with either player moving first. This resulted in a total of 40 games on each tree.

The results with 100,000 playouts are shown in Table 3. DFUCT was very slightly stronger than UCT with the same number of playouts. Additionally, since DFUCT was faster than UCT, it could perform more playouts than UCT in a fixed amount of time. Hence, DFUCT's modified move selection did not hurt its playing strength compared to UCT.

Distributed Search Performance

Table 4 gives observed speedups for up to 1,200 cores compared to sequential DFUCT, which is a better baseline than UCT as we discussed in the previous subsection. The average speedup for ten problems is shown. Our distributed

[2] This is not a serious issue because UCT will be controlled by the time limits rather than the number of total playouts in practical situations such as real tournament games.

Table 2: Frequency of third-best or less-promising move becoming best after a playout

#playouts	Br. 8	Br. 40	Br. 150
6 million	1.27%	2.33%	1.98%
20 million	0.995%	1.78%	1.74%

Table 3: P-game strength: UCT vs. df-UCT (Wins and Losses are for the first player)

Br.	Dp.	1st player	2nd player	Win	Loss	Draw
8	30	UCT	df-UCT	80	119	1
8	30	df-UCT	UCT	84	116	0
40	20	UCT	df-UCT	157	39	4
40	20	df-UCT	UCT	159	36	5
150	10	UCT	df-UCT	58	140	2
150	10	df-UCT	UCT	65	130	5

DFUCT not only scaled well, but its observed speedups were significantly larger than the two versions based on UCT + TDS, especially with a large number of cores. For example, with a branching factor of 8 and the playout time of 1 ms, DFUCT achieved a 741-fold speedup on 1,200 cores, while UCT1 and UCT2 were limited to at most 70-fold speedups. UCT1 and UCT2 modestly scaled up to a limit of around 36 cores with the 1-ms playout. However, the performance improvements with additional cores became much smaller than that of DFUCT. Additionally, compared to UCT1, UCT2 did not have noticeably improved speedup, indicating that an ad hoc modification reducing the communication overhead at the root is not sufficient and the depth-first reformulation technique incorporated into DFUCT is essential to achieve scalability.

It became significantly more difficult to achieve good speedups with a shorter playout time (0.1 ms). This is not surprising because a longer playout time can hide the overheads incurred by distributed tree search, such as communication and managing the job queue. However, DFUCT still yielded over 250-fold speedup in all cases, while UCT1 and UCT2 were limited to around 10-fold speedup each.

Table 5 shows the maximum and average numbers of report jobs (max node and avg., respectively) and the number of report jobs received per playout (RP/pl). The differences between the maximum and average in DFUCT were small, while they were extremely large in both UCT1 and UCT2. The report jobs were almost uniformly distributed in DFUCT, while they were concentrated on a specific processor, (the one in charge of the root for most cases) for UCT1 and UCT2. In UCT1 and UCT2, one processor that was not the home processor of the root received over 21 million report jobs with the branching factor of 8 because it was home to several nodes that spawned search jobs very frequently. The RP/pl field indicates that both UCT1 and UCT2 propagated a playout result all the way back to an ancestor node about 20 levels up in the tree, while DFUCT propagated back 2–3 levels. Using DFUCT greatly reduced the number of report jobs.

Table 4: Average speedup (best cases in bold)

Branching factor 8

	0.1-ms playout			1.0-ms playout		
Cores	UCT1	UCT2	DFUCT	UCT1	UCT2	DFUCT
12	3.29	3.13	**8.68**	7.26	6.92	**10.0**
36	5.15	5.50	**25.6**	18.7	19.0	**30.1**
72	5.78	6.39	**45.1**	27.1	27.8	**59.5**
144	5.96	6.44	**84.4**	36.8	37.6	**120.7**
300	6.78	7.36	**123.7**	47.0	51.3	**250.4**
600	9.88	9.13	**258.7**	62.2	62.9	**473.1**
1200	9.93	9.95	**327.8**	67.7	69.5	**741.2**

Branching factor 40

	0.1-ms playout			1.0-ms playout		
Cores	UCT1	UCT2	DFUCT	UCT1	UCT2	DFUCT
12	5.33	5.26	**9.20**	9.11	8.75	**10.3**
36	8.08	8.77	**26.3**	23.7	24.6	**29.1**
72	8.18	8.61	**50.7**	35.9	38.8	**58.7**
144	8.88	9.26	**89.9**	50.9	57.1	**114.2**
300	9.30	9.89	**140.5**	68.4	78.2	**229.4**
600	11.2	11.8	**236.8**	80.4	98.4	**442.4**
1200	11.5	12.1	**257.6**	89.2	102.2	**635.6**

Branching factor 150

	0.1-ms playout			1.0-ms playout		
Cores	UCT1	UCT2	DFUCT	UCT1	UCT2	DFUCT
12	4.74	5.46	**9.33**	9.01	8.71	**10.3**
36	7.31	7.64	**25.1**	21.6	22.3	**29.7**
72	7.29	7.49	**46.1**	30.8	33.3	**52.2**
144	7.29	7.53	**87.0**	38.5	44.2	**110.8**
300	7.35	7.45	**147.7**	45.0	48.5	**210.7**
600	8.39	8.29	**239.3**	51.4	57.7	**416.0**
1200	9.08	8.93	**251.6**	64.4	65.7	**600.8**

Our current implementation specific to DFUCT sends out messages to all cores to force all jobs to return to the root at the end of the search. A considerable amount of time was required when a large number of cores was involved (e.g., 1–3 s, which was approximately 10-30% of the execution time with 1,200 cores). This overhead can be ignored with longer time settings. Table 6 presents speedups without such overhead that reached nearly 1,000-fold using 1,200 CPU cores, showing the potential of DFUCT.

Conclusions and Future Work

We parallelized UCT in a distributed environment. When UCT was combined with TDS in a straightforward way, we obtained discouraging results in artificial game trees due to the large communication overhead centralized on a specific processor. However, by combining Depth-First UCT with TDS, we achieved drastic improvements. The speedup obtained by our distributed DFUCT was over 700-fold with 1,200 CPU cores in the best case. This is encouraging, since achieving reasonable scalability for parallel MCTS is notoriously challenging.

One important direction for future work is to investigate the performance of our distributed DFUCT in actual games. We are currently implementing it on top of one of

Table 5: Number of received report jobs for 20 million playouts (0.1-ms playout, 600 cores)

	Branch 8			Branch 40			Branch 150		
	Max node	Avg.	RP/pl	Max node	Avg.	RP/pl	Max node	Avg.	RP/pl
UCT1	21,712,516	685,065	20.6	20,398,836	242,752	7.28	20,054,697	183,474	5.50
UCT2	21,802,853	697,210	20.9	20,000,000	240,174	7.21	20,000,000	186,873	5.61
DFUCT	100,738	82,016	2.64	118,725	66,895	2.01	158,115	58,039	1.74

Table 6: Speedup values obtained by excluding time that distributed df-UCT spent in forcing all jobs to return to root at end of parallel search

	0.1-ms playout			1.0-ms playout		
Cores	Br.8	Br.40	Br.150	Br.8	Br.40	Br.150
12	8.69	9.20	9.34	10.0	10.3	10.4
36	25.6	26.3	25.2	30.2	29.2	29.7
72	45.2	50.8	46.2	59.6	58.8	52.2
144	84.7	90.1	87.2	120.8	114.4	111.0
300	126.2	143.1	149.0	252.2	234.9	216.5
600	277.2	264.5	266.5	510.7	485.4	450.8
1,200	445.3	393.6	364.1	997.1	920.9	839.4

the strongest Go programs. There are numerous obstacles to improving the playing strength of computer Go based on distributed df-UCT, such as non-uniform branching factors, varied playout granularity, and enhancements added to sequential UCT. Ideas to overcome them are topics of current research.

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multi-armed bandit problem. *Machine Learning* 47:235–256.

Bourki, A.; Chaslot, G.; Coulm, M.; Danjean, V.; Doghmen, H.; Hoock, J.-B.; Hérault, T.; Rimmel, A.; Teytaud, F.; Teytaud, O.; Vayssière, P.; and Yu, Z. 2010. Scalability and parallelization of Monte-Carlo tree search. In *Proc. Computers and Games (CG'2010)*.

Cazenave, T., and Jouandeau, N. 2007. On the parallelization of UCT. In *the Computer Games Workshop*, 93–101.

Chaslot, G. M. J.-B.; Winands, M. H. M.; and van den Herik, H. J. 2008. Parallel Monte-Carlo tree search. In *Proc. Computers and Games (CG'2008)*, 60–71.

Coulom, R. 2006. Efficient selectivity and backup operators in Monte-Carlo tree search. In *Proc. 5th International Conf. on Computer and Games*, 72–83.

Coulom, R. 2007. Computing elo ratings of move patterns in the game of Go. In *Computer Games Workshop 2007*.

Enzenberger, M., and Müller, M. 2010. A lock-free multithreaded Monte-Carlo tree search algorithm. In *Advances in Computer Games 12 (ACG'2009)*, 14–20.

Enzenberger, M.; Müller, M.; Arneson, B.; and Segal, R. 2010. FUEGO - an open-source framework for board games and Go engine based on Monte-Carlo tree search. *IEEE Trans. on Computational Intelligence and AI in Games* 2(4):259–270.

Gelly, S., and Silver, D. 2009. Combining online and offline knowledge in UCT. In *the 24th International Conf. on Machine Learning*, 273–280.

Gelly, S.; Wang, Y.; Munos, R.; and Teytaud, O. 2006. Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA.

Gelly, S.; Hoock, J. B.; Rimmel, A.; Teytaud, O.; and Kalemkarian, Y. 2008. The parallelization of Monte-Carlo planning. In *Proc. 5th International Conf. on Informatics in Control, Automation, and Robotics*, 198–203.

Kishimoto, A., and Schaeffer, J. 2002. Distributed game-tree search using transposition table driven work scheduling. In *Proc. 31st Inter. Conf. on Parallel Processing*, 323–330.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2009. Scalable, parallel best-first search for optimal sequential planning. In *Proc. ICAPS-2009*, 201–208.

Knuth, D. E., and Moore, R. W. 1975. An analysis of alpha–beta pruning. *Artificial Intell.* 6(4):293–326.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *17th European Conf. on Machine Learning (ECML 2006)*, 282–293.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intell.* 27(1):97–109.

Lorentz, R. 2008. Amazons discover Monte-Carlo. In *Proc. Computers and Games (CG'2008)*, 13–24.

Romein, J. W.; Plaat, A.; Bal, H. E.; and Schaeffer, J. 1999. Transposition table driven work scheduling in distributed search. In *Proc. AAAI'99*, 725–731.

Segal, R. 2010. On the scalability of parallel UCT. In *Proc. Computers and Games (CG'2010)*.

Smith, S., and Nau, D. 1994. An analysis of forward pruning. In *12th National Conf. on Artificial Intell.*, 1386–1391.

Soejima, Y.; Kishimoto, A.; and Watanabe, O. 2010. Evaluating root parallelization in Go. *IEEE Trans. on Computational Intelligence and AI in Games* 2(4):278–287.

Sturtevant, N. R. 2008. An analysis of uct in multi-player games. In *Proc. Computers and Games (CG'2008)*, 37–49.

Tanabe, Y.; Yoshizoe, K.; and Imai, H. 2009. A study on security evaluation methodology for image based biometrics authentication systems. In *IEEE 3rd International Conf. on Biometrics: Theory, Applications and Systems (BTAS09)*.

Zobrist, A. 1970. A new hashing method with applications for game playing. Technical report, Department of Computer Science, University of Wisconsin.