Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)

Parallel Depth First Proof Number Search

Tomoyuki Kaneko
Department of Graphics and Computer Sciences
University of Tokyo, Japan
kaneko@acm.org

Abstract

The depth first proof number search (df-pn) is an effective and popular algorithm for solving and-or tree problems by using proof and disproof numbers. This paper presents a simple but effective parallelization of the df-pn search algorithm for a shared-memory system. In this parallelization, multiple agents autonomously conduct the df-pn with a shared transposition table. For effective cooperation of agents, virtual proof and disproof numbers are introduced for each node, which is an estimation of future proof and disproof numbers by using the number of agents working on the node's descendants as a possible increase. Experimental results on large checkmate problems in shogi, which is a popular chess variant in Japan, show that reasonable increases in speed were achieved with small overheads in memory.

Introduction

Proof number search and its variations have been extensively researched to efficiently identify the game-theoretical value in and-or trees. The depth first proof number search (df-pn) is an effective algorithm and has many applications, including checkers (Schaeffer et al. 2007). Other applications of df-pn contain checkmate problems in shogi which is a popular chess variant in Japan (Nagai 2001), and tactical problems in Go (Kishimoto 2005; Yoshizoe, Kishimoto, and Müller 2007). They are not only interesting by themselves, and also real problems that computer players must solve to win in shogi or Go. The parallelization presented here is designed to be suitable for such situations.

For example, in shogi, most strong programs use (a variant of) the df-pn in addition to a variant of the standard alpha-beta search. In shogi, an endgame is usually a race between two players. Each player wants to force their opponent's king into a checkmate sequence first (Iida, Sakuta, and Rollason 2002). Professional players often win by finding a forced checkmate sequence with more than 30 plies, where the branching factor in the endgame often increases beyond 200. The size of the search space explains why the df-pn is used to perform a deep and selective search, specialized for finding a checkmate sequence.

Figure 1 shows a simple search code which incorporates the df-pn for checkmate search. SearchRoot() and SearchInternal() are standard alpha-beta search functions for the root node and for a non-root node respectively. The lines for checkmate search are shaded in gray. In root nodes, a checkmate sequence is carefully searched; otherwise, a checkmate search is performed only when a β cutoff does not occur. Resources used for a checkmate search depend on the estimation of criticality of the position.

```
SearchRoot(position) {
    if (Checkmate(position, the opponent)) return Win
    usual iterative deepening is performed here
}
SearchInternal(position, α, β) {
    run PVS or PV Splitting and return β if cut off happened
    if (Checkmate(position, the opponent)) return Win
    return α
}
```

Figure 1: Two types of invocation of df-pn (shaded in gray) inside PVS or PV Splitting

As multi-core processors become popular, the efficiency of the df-pn search becomes important in combination with parallel search methods such as Principal Variation (PV) splitting (Marsland, Member, and Popowich 1985). There are two major issues for improvement: the parallelization of the df-pn and the use of interim results using the algorithm in different threads. For root nodes, the parallelization of the df-pn is beneficial because additional CPU resources will be available. For other nodes, all CPU resources are used for execution of SearchInternal in different nodes. Thus, instead of the parallelization of the df-pn, the sharing of information will improve efficiency because the same position is frequently visited by using the df-pn in different threads. Note that similar situations will also occur in Go when one combines a parallel Monte-Carlo tree search (Chaslot, Winands, and Herik 2008) and a tactical search (Cazenave and Helmstetter 2005). This paper first introduces a framework of sharing information among virtual agents conducting the df-pn and presents a method for parallel cooperation of agents based on the framework.

The rest of this paper is organized as follows: The next section reviews related research. After a brief introduction of the df-pn search algorithm, our parallelization is

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

presented. Then, the experimental results in shogi are discussed, followed by concluding remarks.

Related Work

Proof number search (PNS) is an effective and-or tree search method that uses proof and disproof numbers (Allis, van der Meulen, and van den Herik 1994). The details of this method are explained in the next section. Since PNS is a type of best first search, all nodes must be kept in memory. Thus, PNS is not suitable for solving large problems, and there are actually many checkmate problems in shogi that PNS cannot solve. To solve such difficult problems, several depth-first variants that use proof numbers and disproof numbers were developed (Seo, Iida, and Uiterwijk 2001; Nagai 1998; Ueda et al. 2008). Among these variations, the df-pn (Nagai 2001) is the most promising. The behavior of this algorithm is equivalent to that of PNS on condition that the state space is actually a "tree". It also achieves memory efficiency by introducing garbage collection (GC) of subtrees when needed. Note that the state space of some popular games are not trees and contain cycles. In such cases, additional techniques are required to avoid complicated problems such as graph history interaction (GHI) (Kishimoto 2005) and estimation problems (Kishimoto 2010).

Many methods have been developed for parallelization of a game tree search based on an alpha-beta search (Marsland, Member, and Popowich 1985; Brockington 1997). However, there has been little research on the parallelization of and-or tree search methods based on proof and disproof numbers. The difficulty is in a task decomposition method since one can hardly expect the width and depth of the tree to be searched in these methods based on proof numbers. Pioneering work is ParaPDS which increases the speed with a scaling factor of 3.6 over a sequential PDS using 16 distributed processors (Kishimoto and Kotani 1999). Recently, parallelization based on randomization was applied to PNS (Saito, Winands, and van den Herik 2009). In this work, a best-first version called RP-PNS achieved a reasonable increase in speed in a "Lines of Action" game; however, the memory overheads reached almost 50%. We focused on the parallelization of the df-pn which has advantages over PNS, and similar increases in speed were achieved with much less overheads.

Recently, parallelization of a Monte-Carlo tree search has been intensively investigated in Go. A notable idea on parallelization of a Monte-Carlo tree search is the notion of "virtual wins" or "virtual losses" (Chaslot, Winands, and Herik 2008), which was effective in assigning processors to Monte-Carlo playouts. Virtual wins and losses have an interesting similarity with virtual proof numbers introduced in this paper, though there are substantial differences in the behavior of agents in parallelization.

Depth First Proof Number Search

This section briefly reviews the df-pn algorithm based on proof and disproof numbers defined over an and-or tree. The purpose of an and-or tree search is to identify the game-theoretical value (proof or disproof) of the root node of and-or trees. In the cases of checkmate search, proof is defined as win for the attacker, and disproof is defined as an evasion of the defender into positions where no check move exists. An OR-node is for a position of the attacker's turn where he/she tries to find a forced win with the checkmate sequence. An AND-node is for a position of the defender's turn.

Proof Numbers and Disproof Numbers

A pair of heuristic numbers called proof numbers ($\widehat{pn}(n)$) and disproof numbers ($\widehat{dn}(n)$) are used in the df-pn as well as in PNS. Proof and disproof numbers are defined for each node n. The $\widehat{pn}(n)$ is an estimation of the difficulty in proving a node n in the current search tree. Its value is defined as the minimum number of leaf nodes in the tree that have to be newly proven to prove n. Similarly, disproof numbers $\widehat{dn}(n)$ are the minimum number of leaf nodes that have to be newly disproven to disprove n. Consequently, $(\widehat{pn}(n), \widehat{dn}(n)) = (1, 1)$ if n is a frontier node in the current tree. Also, $(\widehat{pn}(n), \widehat{dn}(n))$ is $(0, \infty)$ or $(\infty, 0)$ if n is already proven or disproven, respectively.

When the search space is a directed acyclic graph, it is computationally infeasible to identify the exact $(\widehat{pn}(n), \widehat{dn}(n))$ defined above for internal nodes that are not proven nor disproven yet. As a result, an approximation of $(\widehat{pn}(n), \widehat{dn}(n))$ by $(pn(n), dn(n))$ are used in the df-pn, calculated by

$$pn(n) = \begin{cases} \min_{c \in child(n)} pn(c) & (n \textit{ is OR node}) \\ \sum_{c \in child(n)} pn(c) & (n \textit{ is AND node}) \end{cases}$$
$$dn(n) = \begin{cases} \min_{c \in child(n)} dn(c) & (n \textit{ is AND node}) \\ \sum_{c \in child(n)} dn(c) & (n \textit{ is OR node}) \end{cases} \quad (1)$$

based on the assumption that the search space is a tree. This assumption causes overestimation of proof and disproof numbers, see (Kishimoto 2005; 2010) for practical cures.

Df-pn and PNS

PNS repeatedly expands a frontier node, called the "most-proving node", in a best-first manner until the game-theoretical value of the root is determined (Allis, van der Meulen, and van den Herik 1994). The most proving node is identified by recursively selecting a child from the root that has the least pn (dn) among its siblings if the parent is an OR (AND) node. The df-pn expands the same frontier node in a depth-first manner guided by a pair of thresholds (th_{pn}, th_{dn}), which indicates whether the most-proving node exists in the current subtree. This difference improved runtime efficiency in the df-pn and also made the incorporation of GC available.

The pseudo code of the df-pn is shown in Figure 2. The lines shaded in gray are enhancements used in the df-pn$^+$, which are explained in the next subsection. OrNode() is a function for OR-nodes, in which the df-pn searches descendants of a given node n while $(pn(n), dn(n))$ are under the given threshold (th_{pn}, th_{dn}). The most proving child n_1 with the least proof number is visited next by calling the

```
OrNode(n,thpn,th_dn) {
  Mark(n)
  while (true) {
    foreach children c {
      (pn(c), dn(c)) = Lookup(c) or (H_pn(c),H_dn(c))
      pn(c) += Cost_pn(n, c)
      dn(c) += Cost_dn(n, c)
    }
    compute pn(n) and dn(n) by Equation (1)
    if (pn(n) ≥ thpn or dn(n) ≥ th_dn) break
    identify two children (n1, n2) with the two least pn
      s.t. pn(n1) ≤ pn(n2) ≤ pn(others)
    np = min(thpn, pn(n2) + 1) - Cost_pn(n, n1)
    nd = th_dn − dn(n) + dn(n1)
    AndNode(n1,np, nd)
  }
  Store(n, pn(n), dn(n))
  Unmark(n)
}
```

Figure 2: Pseudo code of df-pn and df-pn$^+$ (shaded in gray).

function AndNode with a new threshold computed using the thresholds for n as well as proof and disproof numbers of all children. The proof and disproof numbers for children c are retrieved from a transposition table. When the proof and disproof numbers are not stored in the table, they are initialized with $(1, 1)$. This is a case when c is a never visited fresh node or when the information of c is discarded by GC.

When either $pn(n)$ or $dn(n)$ becomes equal to or greater than its threshold, the df-pn leaves n and returns to the parent in order to expand the most proving node outside the subtree of n. The df-pn may visit n again when a frontier node inside the subtree of n becomes the most proving node.

Similarly, the most proving child with the least disproof number is visited in AND-nodes. There is a clear symmetry between proof (disproof) numbers in OR-nodes and disproof (proof) numbers in AND-nodes, and the function AndNode() is defined similarly.

Enhancements in Df-pn

Many enhancements have been developed for the df-pn. The df-pn$^+$ was largely improved by introducing heuristic evaluation functions into the df-pn (Nagai and Imai 1999). The differences between the df-pn$^+$ and df-pn are shaded in gray in Figure 2. Proof and disproof numbers are no longer initialized with $(1, 1)$, instead, they are initialized using the evaluation functions, $\mathbf{H}_{pn}$ and $\mathbf{H}_{dn}$, which are designed to predict future proof (disproof) numbers. $\mathbf{Cost}_{pn}$ and $\mathbf{Cost}_{dn}$ are also evaluation functions. While the effect of **H** vanishes once the node is expanded, the effect of **Cost** persists until a game-theoretical value is identified. In the experiments discussed in "Experimental Results" section, **H** are computed with a two-ply fixed-depth search (Kaneko et al. 2005), and a penalty for sacrificing moves is given as **Cost**.

Many games have special rules about the repetition of the same position, such as the SuperKo rule in Go. To detect repetitions, a flag for each node indicating whether the node is currently visited or not is maintained by Mark() and Unmark() functions called at the beginning and ending of OrNode(). Note that one should carefully treat a repetition, to avoid graph history interaction (GHI) problems. See (Kishimoto 2005) for details.

```
OrNodePar(n,thpn,th_dn, tid ) {
  Mark(n, tid )
  while (true) {
    foreach children c {
      lock(c)
      (pn(c), dn(c)) = Lookup(c) or (H_pn(c),H_dn(c))
      pn(c) += Cost_pn(n, c) + T(n,c) /* vpn */
      dn(c) += Cost_dn(n, c)
      unlock(c)
    }
    compute pn(n) and dn(n) by Equation (1)
    if (pn(n) ≥ thpn or dn(n) ≥ th_dn) break
    identify two children (n1, n2) with the two least pn
      s.t. pn(n1) ≤ pn(n2) ≤ pn(others)
    np = min(thpn, pn(n2) + 1) - Cost_pn(n, n1)
    nd = th_dn − dn(n) + dn(n1)
    AndNodePar(n1,np, nd, tid )
    if (stopped by other agents) break
  }
  lock(n)
  unless (n was (dis)proven by other agents)
    Store(n, pn(n), dn(n))
  unlock(n)
  Unmark(n, tid )
  if (pn(n)=0 or dn(n)=pn0) stop agents in descendants(n)
}
```

Figure 3: Pseudo code of an agent in parallel df-pn$^+$ (differences with sequential codes are shaded in gray)

Another important enhancement is a cure for overestimation problems caused by DAG. The basic idea is that one should sometimes take the maximum instead of the total in computation of proof and disproof numbers in Equation (1). See (Nagai 2001; Kishimoto 2010) for precise conditions and discussions.

Parallelization of Df-pn

This section presents our parallelization of the df-pn. In a popular parallelization, decomposed tasks are executed in parallel. However, such a top-down assignment of tasks is difficult because one can hardly estimate the search tree needed to be expanded to solve a problem before finding a solution. Thus, we propose a multi-agent framework that gradually integrates, in a bottom-up way, the interim results of the df-pn. Figure 3 presents codes of a df-pn agent for parallel cooperation. OrNodePar() is a function for OR-nodes, where the lines shaded in gray are the differences with the codes of the sequential df-pn$^+$. The function AndNodePar() is defined similarly.

In this parallelization, agents share a transposition table

for proof and disproof numbers, to use the results of subtrees expanded by other agents. Thus, lock() and unlock() are added around Lookup() and Store(). The lock overhead was small and about 1% in the experiments. Before Store(), an agent tests whether a proof or disproof is already found by other agents, so as not to overwrite it with an interim result. Also, Mark() and Unmark() functions are extended with an additional argument of a thread id (*tid*) to determine who is currently visiting that node. These modifications are sufficient for using agents in parallel.

Note that some enhancements not discussed here use path-dependent information. Such information should be kept in thread local storage when an agent works on different root nodes. See (Kishimoto 2005) for details on such enhancements and implementations of path-dependent information.

Balancing of Search Trees and Cut-offs

This section addresses the issues of balancing and cut-offs, to make agents suitable for harmoniously solving a common problem with a simple driver shown in Figure 4.

For each agent to expand search trees to be slightly different from each other, a virtual proof number $vpn(n, c)$ is introduced as $vpn(c) = pn(c) + \mathit{Cost}_{pn}(n, c) + T(n, c)$. $T(n, c)$ acts as a pheromone in multi-agent systems (Blum and Roli 2003) and represents the congestion of agents at c. Since the node with the least proof number is selected at OR-nodes, a positive value of $T(n, c)$ makes c less visited than cases for $T(n, c) = 0$. If T is always 0, the same tree is expanded as the sequential df-pn$^+$. Although the ideal shape of $T(n, c)$ is not known, a simple function returning the number of agents visiting the descendants of c is sufficient for the experiments described in the next section. The number of such agents can easily be obtained as a set of *tid*s maintained by Mark() and Unmark() functions. Similarly, a virtual disproof number $vdn(n, c)$ is introduced for AND-nodes.

Once a node is proven or disproven, search on the node's descendants is no longer valuable because it does not affect the game-theoretical value of the root node. It is similar to beta-cut in alpha-beta searches. Thus, to stop search in such cases, two modifications are applied at the end of AndNodePar() and the end of Unmark(). For an efficient implementation of this facility, a stack is allocated for each thread. In Mark() and Unmark() functions, the Zobrist hash of n is pushed into and popped from the stack, respectively. When an agent stops other agents at the next line of Unmark(), it asynchronously notifies the Zobrist hash of n to the agents. Each agent will test whether a hash is notified by another agent at the next line of AndNodePar(). When notified, it then examines its stack to see whether the notified hash is still one of the ancestors of a node currently visiting.

Experimental Results

To show the effectiveness of the parallelization presented in the previous section, experiments in shogi were conducted. A computer with two quad-core CPUs with AMD Opteron Processor 2376 running 64-bit linux was used for the experiments. The author's implementation is available as part

ParallelCheckmateSearch(n) {
  parallel-for each thread (*tid*)
    OrNodePar(n, ∞, ∞,*tid*)
  return $(pn(n), dn(n))$ stored in table
}

Figure 4: Pseudo code of parallel df-pn$^+$ driver.

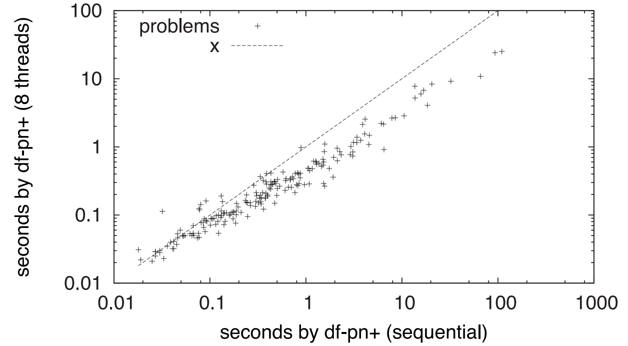

Figure 5: Comparison of solution time for each problem

of open source software at `http://gps.tanaka.ecc.u-tokyo.ac.jp/gpsshogi/`. While both parallel and sequential versions worked well with GC, the experiments were conducted without GC to measure search overheads in terms of expansion of irrelevant nodes.

First, the sequential df-pn$^+$ was implemented with state-of-the-art techniques (Kawano 1996; Nagai 2001; Kishimoto 2005; Kaneko et al. 2005). The performance of this df-pn$^+$ is expected to be better or equal to other methods, though it is difficult to show precise comparisons. For example, PN* search required 550,939,913 nodes to be searched for more than 20 hours (Seo, Iida, and Uiterwijk 2001) to solve "Microcosmos," which is a famous checkmate problem whose solution consists of 1,525 steps. Now, the df-pn$^+$ solved it with less than 50,000,000 nodes within three minutes. The reduction in the number of nodes searched suggests the advantage of the df-pn$^+$, while better hardware can contribute to reducing solution time.

Two problem sets, Shogi-Zuko and Shogi-Muso, were used in these experiments as in (Seo, Iida, and Uiterwijk 2001). While they were created during the Edo period in Japan more than 200 years ago, they are still good test-sets with relatively large problems whose solution steps often exceed one hundred. Because they contain some defective problems, 99 and 94 complete problems out of 100 problems were used. All the problems were successfully solved with all implementations. Figure 5 shows that the parallel df-pn$^+$ with 8-threads solved problems faster than the sequential df-pn$^+$. The horizontal axis plots the solution time in seconds by the sequential one, and the vertical axis plots those for the parallel one. Almost all problems were solved in a shorter time with the parallel df-pn$^+$. Larger increases in speed was observed for problems where the sequential df-pn$^+$ required more solution time, and the scaling factor for the four largest problems was more than 4.0.

Figure 6 shows scaling factors in speed for 2, 4, and 8

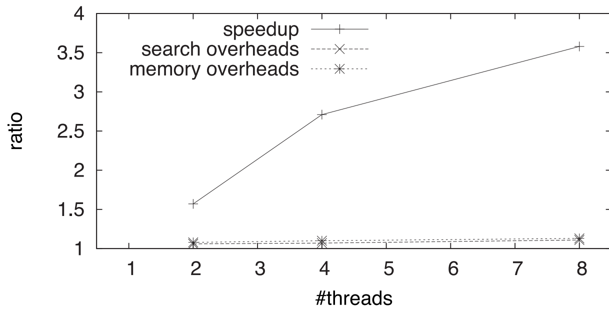

Figure 6: Scaling factors in 2, 4, and 8 threads for Hard problems

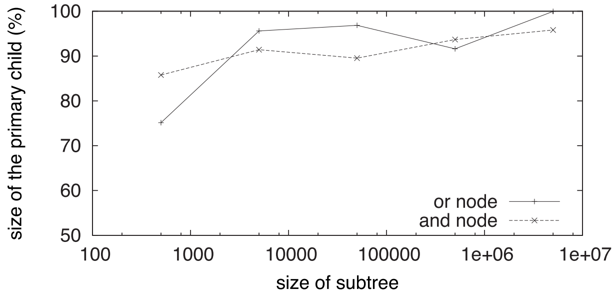

Figure 7: Imbalance in game tree sizes measured for nodes in proof tree for 100-th problem in Shogi-Muso.

threads, as well as the overheads measured by the number of visited and stored nodes in the parallel search divided by that in the sequential one. These statistics are measured using a subset of the problems called Hard, which consists of 29 such problems the sequential df-pn$^+$ required more than 1,000,000 nodes to solve. The scaling factors were 2.71 and 3.58 for 4 and 8 threads, respectively, while the search overhead was about 11% at 8 threads. While the scaling factors were similar, the overhead was much less than that reported with RP-PNS in Lines of Action (2009).

The scaling factors observed seem modest compared to those in parallel search for multi-value game trees. The reason can partially be explained by the imbalance in proof trees, where a proof tree is a subset of a searched game tree that consists of nodes required to prove the root position. Here, we show statistics of the proof tree identified by the sequential df-pn$^+$ for the 100-th problem of Shogi-Muso, where the solution consists of 163 steps. Let a primary child p in node n be a child with the largest searched subtree if n is an AND-node and be a proven child if n is an OR-node. The vertical axis in Figure 7 plots the ratio of the subtree size of p divided by the tree size of n, and the horizontal axis plots the subtree size of n. The ratio is often over 90%, even in AND-nodes as well as in OR-nodes. This indicates that one has to allocate most resources to one child for each node, and explains the difficulty in task decomposition. On the other hand in cases of multi-value search, subtrees are expected to be balanced at nodes called "ALL nodes", where β cut-off did not occur.

Detailed performance of the parallel df-pn$^+$ are summarized in the upper half of Table 1. For each problem set, the number of nodes visited, that of nodes stored, and solution time are listed. The search trees may vary for each execution, if the number of threads ≥ 2, at nodes that are irrelevant to construct the proof of the root. Therefore, an average of ten executions is presented for each cell, as well as the standard deviation (σ) for solution time. The lower half of Table 1 lists the same data for the df-pn where the evaluation functions (**H** and **Cost**) implemented in the df-pn$^+$ are switched off. The nodes visited in the df-pn$^+$ includes those using a two-ply fixed depth search serving as **H**. Because the fixed depth search is much faster than search in the df-pn, the df-pn$^+$ solved problems quicker than df-pn even when larger numbers are reported in "#node visited" for the df-pn$^+$. In summary, the df-pn$^+$ and df-pn both significantly increased speed, and the improvement in the df-pn$^+$ is greater than that of the df-pn. This can be explained by the fact that more nodes tend to have similar proof and disproof numbers if evaluation functions are absent.

Concluding Remarks

This paper presented a simple but effective parallelization of the df-pn search algorithm for a shared-memory system. The parallelization is based on asynchronous cooperation of agents, where each agent independently conducts a df-pn with a shared transposition table. The implementation is much simpler than other parallel search methods and requires little modification from its sequential implementation of the df-pn. For effective cooperation of agents, virtual proof and disproof numbers are introduced for each node, which is an estimation of future proof and disproof numbers by the number of agents working on the node's descendants.

Experimental results on large checkmate problems in shogi showed that reasonable increases in speed were achieved with small overheads in memory. The scaling factor was about 3.6 in 8 threads, and search overheads were less than 15% for large problems. This is the first result of parallelization in the df-pn, and to the best of the author's knowledge, the performance was better than or at least equal to those in current research on parallelization of proof number search variants.

Acknowledgments

The author would like to thank the anonymous referees for their beneficial comments.

References

Allis, L. V.; van der Meulen, M.; and van den Herik, H. J. 1994. Proof-number search. *Artificial Intelligence* 66:91–124.

Blum, C., and Roli, A. 2003. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.* 35(3):268–308.

Brockington, M. G. 1997. *Asynchronous Parallel Game-Tree Search*. Ph.D. Dissertation, Department of Computing Science, University of Alberta.

Table 1: Increase in speed in df-pn$^+$

	#thread	#node visited	overhead	#table	overhead	seconds	σ	speedup
Muso	1	68,114,896.0	-	11,491,308.0	-	235.0	-	1.00
	2	77,684,522.2	1.14	13,027,766.5	1.13	159.7	13.8	1.47
	4	79,551,231.2	1.16	13,257,241.6	1.15	95.9	5.2	2.45
	8	83,587,512.6	1.22	13,694,800.8	1.19	74.9	6.1	3.13
Zuko	1	87,411,686.0	-	14,360,182.0	-	317.8	-	1.00
	2	96,082,504.3	1.09	16,014,805.7	1.11	213.2	23.4	1.49
	4	97,858,084.2	1.11	16,376,611.5	1.14	126.2	7.9	2.51
	8	105,888,878.9	1.21	17,354,390.5	1.20	102.5	6.4	3.01
Hard	1	132,403,071.0	-	21,004,381.0	-	493.0	-	1.00
	2	141,306,495.1	1.06	22,873,823.6	1.08	313.8	29.6	1.57
	4	141,917,849.5	1.07	23,121,211.8	1.10	181.4	10.5	2.71
	8	147,729,337.7	1.11	23,757,377.6	1.13	137.5	5.7	3.58

Increase in speed in df-pn

	#thread	#node visited	overhead	#table	overhead	seconds	σ	speedup
Muso	1	49,068,219.0	-	16,239,257.0	-	259.1	-	1.00
	2	50,283,856.5	1.02	16,894,628.1	1.04	166.9	10.3	1.55
	4	50,947,624.0	1.03	17,475,295.5	1.07	103.9	5.9	2.49
	8	55,438,399.7	1.12	18,832,165.2	1.15	92.2	5.0	2.81
Zuko	1	71,873,553.0	-	24,319,554.0	-	463.9	-	1.00
	2	73,589,641.6	1.02	25,465,728.8	1.04	279.8	45.4	1.65
	4	71,261,572.7	0.99	25,060,111.5	1.03	164.3	13.3	2.82
	8	84,160,639.6	1.17	29,327,590.0	1.20	158.4	16.8	2.92
Hard	1	103,166,296.0	-	33,477,914.0	-	644.0	-	1.00
	2	103,688,841.7	1.00	34,455,449.8	1.02	386.6	42.0	1.66
	4	99,371,797.2	0.96	33,845,832.7	1.01	223.3	17.3	2.88
	8	112,912,943.8	1.09	38,439,427.4	1.14	203.9	16.7	3.15

Cazenave, T., and Helmstetter, B. 2005. Combining tactical search and monte-carlo in the game of go. In *Proceedings of CIG*, 171–175. IEEE.

Chaslot, G. M.; Winands, M. H.; and Herik, H. J. 2008. Parallel monte-carlo tree search. In *CG '08: Proceedings of the 6th international conference on Computers and Games*, 60–71. Berlin, Heidelberg: Springer-Verlag.

Iida, H.; Sakuta, M.; and Rollason, J. 2002. Computer shogi. *Artificial Intelligence* 134(1–2):121–144.

Kaneko, T.; Tanaka, T.; Yamaguchi, K.; and Kawai, S. 2005. Df-pn with fixed-depth search at frontier nodes. In *The 10th Game Programming Workshop*, 1–8. (In Japanese).

Kawano, Y. 1996. Using similar positions to search game trees. In Nowakowski, R. J., ed., *Games of No Chance*, volume 29 of *MSRI Publications*, 193–202. Cambridge University Press.

Kishimoto, A., and Kotani, Y. 1999. Parallel AND/OR tree search based on proof and disproof numbers. In *Game Programming Workshop in Japan*, number 14, 24–30.

Kishimoto, A. 2005. *Correct and Efficient Search Algorithms in the Presence of Repetitions*. Ph.D. Dissertation, University of Alberta.

Kishimoto, A. 2010. Dealing with infinite loops, underestimation, and overestimation of depth-first proof-number search. In *AAAI*, to appear.

Marsland, T. A.; Member, S.; and Popowich, F. 1985. Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 7:442–452.

Nagai, A., and Imai, H. 1999. Application of df-pn$^+$ to Othello endgames. In *Game Programming Workshop in Japan '99*, 16–23.

Nagai, A. 1998. A new AND/OR tree search algorithm using proof number and disproof number. In *Complex Games Lab Workshop*. Electrotechnical Laboratory, Machine Inference Group, Tsukuba, Japan.

Nagai, A. 2001. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. Ph.D. Dissertation, the University of Tokyo.

Saito, J.-T.; Winands, M.; and van den Herik, H. J. 2009. Randomized parallel proof-number search. In *Advances in Computer Games (ACG 2009)*, to appear.

Schaeffer, J.; Burch, N.; Bjornsson, Y.; Kishimoto, A.; Muller, M.; Lake, R.; Lu, P.; and Sutphen, S. 2007. Checkers is solved. *Science* 1144079+.

Seo, M.; Iida, H.; and Uiterwijk, J. W. 2001. The PN*-search algorithm: Application to tsume-shogi. *Artificial Intelligence* 129(1-2):253–277.

Ueda, T.; Hashimoto, T.; Hashimoto, J.; and Iida, H. 2008. Weak proof-number search. In *CG '08*, 157–168. Berlin, Heidelberg: Springer-Verlag.

Yoshizoe, K.; Kishimoto, A.; and Müller, M. 2007. Lambda depth-first proof number search and its application to go. In Veloso, M. M., ed., *IJCAI*, 2404–2409.