Randomized Parallel Proof-Number Search

Jahn-Takeshi Saito[1], Mark H.M. Winands[1], and H. Jaap van den Herik[2]

[1] Department of Knowledge Engineering
Faculty of Humanities and Sciences, Maastricht University
{j.saito, m.winands}@maastrichtuniversity.nl
[2] Tilburg centre for Creative Computing (TiCC)
Faculty of Humanities, Tilburg University
h.j.vdnherik@uvt.nl

Abstract. Proof-Number Search (PNS) is a powerful method for solving games and game positions. Over the years, the research on PNS has steadily produced new insights and techniques. With multi-core processors becoming established in the recent past, the question of parallelizing PNS has gained new urgency. This article presents a new technique called Randomized Parallel Proof-Number Search (RP–PNS) for parallelizing PNS on multi-core systems with shared memory. The parallelization is based on randomizing the move selection of multiple threads, which operate on the same search tree. RP–PNS is tested on a set of complex Lines-of-Action endgame positions. Experiments show that RP–PNS scales well. Four directions for future research are given.

1 Introduction

Most computer programs for board games successfully employ $\alpha\beta$ search. For some games, however, $\alpha\beta$ search displays a weakness in the endgame that can currently neither be overcome by endgame databases nor by other $\alpha\beta$ extensions. To remedy the deficit, mate searches may be applied. One such alternative to $\alpha\beta$ search is Proof-Number Search (PNS). PNS enjoys popularity as a powerful method for solving endgame positions and full games. Since its introduction by Allis *et al.* [1] in 1994, PNS has developed into a whole family of search algorithms (e.g., PN^2 [1] and df-pn [12]) with applications to many games, such as Shogi [16], the one-eye problem in Go [9], Checkers [15], and Lines of Action [19].

A variety of parallel $\alpha\beta$ algorithms have been proposed in the past [3] but so far not much research has been conducted on parallelizing PNS (cf. Section 3). With multi-core processors becoming established as standard equipment parallelizing PNS has become an important topic. Pioneering research has been conducted by Kishimoto [8] who parallelized the depth-first PNS variant PDS. His algorithm is called ParaPDS and is designed for distributed memory systems. In this article we address the problem of parallelizing PNS and PN^2 for shared memory systems. The parallelization is based on randomizing the move selection of multiple threads, which operate on the same search tree. This method is called

Randomized Parallel Proof-Number Search (RP–PNS). Its PN^2 version is called RP–PN^2.

The article is organized as follows. Section 2 describes the PNS algorithm and two of its variants. Section 3 discusses the options for the parallelization of PNS in general terms. Section 4 introduces the new parallel PNS, RP–PNS. Section 5 shows and discusses the results of testing RP–PNS on a set of complex Lines-of-Action endgame positions. Section 6 provides a conclusion and gives four directions for future research.

2 Proof-Number Search

This section describes the sequential PNS algorithm (Subsect. 2.1) and two of its variants, PDS and PN^2 (Subsect. 2.2).

2.1 Sequential PNS

PNS is a best-first search for AND/OR trees. The search aims at proving or disproving a binary goal, i.e., a goal that can be reached by the first player or be refuted by the second player under optimal play by both sides. Each node N in the tree contains two numbers called the *proof number* ($pn(N)$) and the *disproof number* ($dn(N)$), respectively.

PNS iterates the best-first search cycle consisting of three steps.

1. Selection: starting at the root, a path P consisting of successor nodes is created until a leaf L is found; a heuristic guides the selection of successors;
2. Expansion: L is expanded and its children's proof and disproof numbers are set;
3. Back-up: the new values of the expanded node L are propagated back to the root.

Informally, the algorithm runs as follows. The selection step finds a leaf L of the tree. In PNS L, is called the *most-proving node*, i.e., the node that is expected to reach a proof (or disproof) with the fewest additional expansions. The most-proving node is found by heuristically descending a path P in the tree starting at the root. At every node N, the *best successor* ($bs(N)$) is selected and this $bs(N)$ becomes the new N. This procedure is repeated until a leaf L is reached. The best successor of N is determined differently in OR and AND nodes. (1) In OR nodes (when player 1 moves), the $bs(N)$ is the child that requires the fewest number of additional expansions to prove the goal. $pn(N)$ represents this number. (2) In AND nodes (when player 2 moves), $bs(N)$ is the child that requires the fewest additional expansions to disprove the goal. $dn(N)$ represents this number.

More formally we describe the best successor and the successor number as follows. Given a non-terminal node N, its children are denoted by $N_i, i = 1, ..., |N|$ where $|N|$ is the number of children of N. If N is an OR node, the N_i are sorted ascendingly by their proof number $pn(N_i)$. If N is an AND node, the N_i are sorted ascendingly by their disproof number $dn(N_i)$. The *successor number* of N for a child N_i is $sn(N_i) = pn(N_i)$ if N is an OR node and $sn(N_i) = dn(N_i)$ if

Rules for *AND* nodes:

$$pn(N) = \sum_{S \in successor(N)} pn(S)$$

$$dn(N) = \min_{S \in successor(N)} dn(S)$$

Rules for *OR* nodes:

$$pn(N) = \min_{S \in successor(N)} pn(S)$$

$$dn(N) = \sum_{S \in successor(N)} dn(S)$$

Fig. 1. Rules for updating proof and disproof numbers for a node N.

N is an AND node. The best successor of N is the child N_1. The *best successor number* $bsn(N)$ is $sn(N_1)$.

The expansion step of the cycle expands L and initializes its children's proof and disproof numbers. If a new child directly proves the goal, its proof number is set to 0 and its disproof number is set to infinity. Correspondingly, if a new child directly disproves the goal, its disproof number is set to 0 and its proof number is set to infinity. If the child neither proves nor disproves, the number of children of the leaf can be used to set these numbers.[3]

In the back-up step the newly assigned proof and disproof numbers are propagated back to the root changing the proof and disproof number in each node on the path P. The rules applied for updating proof and disproof numbers are given in Fig. 1.

After the root has been reached and its values have been updated, the cycle is complete. The cycle is repeated until the termination criterion is met. The criterion is satisfied if either the root's proof number is 0 and the disproof number is infinity, or vice versa. In the first case, the goal is proved. In the second case the goal is disproved.

Figure 2 (placed in Sect. 4 where it is also used for explanation) shows a search tree with proof and disproof numbers. The proof and disproof numbers of the interior nodes can be calculated from the children's numbers by applying the updating rules presented in Fig. 1.

2.2 PDS and PN^2

A weakness of PNS is its memory consumption. This problem arises because the whole tree is stored in memory. There are many variants of PNS that address the memory problem; two of them are PDS and PN^2.

PDS by Nagai [13] solves the memory problem by transforming the best-first search into a depth-first search. PDS applies multiple-iterative deepening at every node. PDS can only function successfully by means of a transposition table to speed up the re-searches.

Like PDS, PN^2 [1] reduces memory requirements of PNS by re-searching parts of the tree. PN^2 consists of two levels of PNS. The first level PNS (PN$_1$) calls a PNS at the second level (PN$_2$) for an evaluation of the most-proving node of the PN$_1$-search tree. This PN$_1$ search is bound by a maximum number

[3] We remark that other methods for estimating the proof and disproof numbers of newly expanded leafs have been proposed [2].

of nodes M to be stored in memory. Different ways have been proposed to set this bound [1, 2]. The PN_1 search is stopped when the number of nodes stored in memory exceeds M or the subtree is (dis)proved. After completion of the PN_1 search, the children of the root of the PN_1-search tree are preserved, but subtrees are removed from memory.

3 Parallelization of PNS

This section introduces some basic concepts for describing the behavior of parallel search algorithms (Subsect. 3.1), outlines ParaPDS, a parallelization of PDS (Subsect. 3.2), and explains parallel randomized search (Subsect. 3.3).

3.1 Terminology

Parallelization aims at reducing the time that a sequential algorithm requires for terminating successfully. The speedup is achieved by distributing computations to multiple threads executed in parallel.

Parallelization gains from dividing computation by multiple resources but simultaneously it may impose a computational cost. According to Brockington and Schaeffer [4] three kinds of overhead may occur when parallelizing a search algorithm: (1) search overhead, resulting from extra search not performed by the sequential algorithm; (2) synchronization overhead, created at synchronization points when one thread is idle waiting for another thread; (3) communication overhead, created by exchanging information between threads.

Search overhead is straightforward and can be measured by the number of additional nodes searched. Synchronization and communication overhead depend on the kind of information sharing used. There are two kinds of information sharing: (i) message passing and (ii) shared memory. Message passing simply consists of passing information between memory units exclusively accessed by a particular thread. Under shared memory all threads can access a common part of memory. With the advent of multi-core CPUs memory sharing has become common place.

An important property governing the behavior of parallel algorithms is scaling. It describes the efficiency of parallelization with respect to the number of threads as a fractional relation t_1/t_T between the time t_1 for terminating successfully with one thread and the time t_T for terminating successfully with T threads.

3.2 ParaPDS and the Master-Servant Design

The only existing parallelization of PNS described in the literature has so far been ParaPDS by Kishimoto and Kotani [8].[4] This pioneering work of parallel PNS achieved a speedup of 3.6 on 16 processors on a distributed memory

[4] Conceptually related to PNS is Conspiracy Number Search (CNS) by McAllester [11]. Lorenz [10] proposes to parallelize a variant of CNS (PCCNS). PCCNS uses a master-servant model ("Employer-Worker Relationship", ibid.).

machine. Therefore, processes are used instead of threads for parallelization. ParaPDS relies on a master-servant design. One master process is coordinating the work of several servant processes. The master manages a search tree up to a fixed depth d. The master traverses through the tree in a depth-first manner typical for PDS. On reaching depth d it assigns the work of searching further to an idle servant. The search results of the servant process are backed up by the master.

The overhead created by ParaPDS is mainly a search overhead. There are two reasons for this overhead: (1) lack of a shared-memory transposition table, and (2) the particular master-servant design. Regarding reason (1), ParaPDS is asynchronous, i.e., no data is passed between the proccesses except at the initialization and the return of a servant process. ParaPDS thereby avoids message passing. The algorithm is designed for distributed-memory machines common at the time ParaPDS was invented (i.e., 1999). Transposition tables are important to PDS, as this variation of PNS performs iterative deepening. An implication of using distributed-memory machines is that ParaPDS cannot profit from a shared transposition table and loses time on re-searching nodes. Regarding reason (2), the master-servant design can lead to situations in which multiple servant processes are idle because the master process is too busy updating the results of another processes or finding the next candidate to pass to a servant process.

One may speculate that the lack of a shared-memory transposition table in ParaPDS could nowadays be amended to a certain degree, at the expense of a synchronization overhead, by the availability of shared-memory machines. However, the second reason for the overhead of the master-servant design still remains.

3.3 Randomized Parallelization

An alternative to the master-servant design of ParaPDS for parallelizing tree-search is *randomized parallelization.* Shoham and Toledo [17] proposed the method for parallelizing *any* kind of best-first search on AND/OR trees. The method relies on a heuristic which may seem counterintuitive at first: the selection in any sequential best-first search is based on the heuristic evaluation of the children. (In PNS, the *selection heuristic* is based on proof and disproof numbers, as outlined in Sect. 2.) Instead of selecting the child with the best heuristic evaluation, a probability distribution of the children determines which node is selected. Shoham and Toledo call this a *randomization* of the move selection. Randomized Parallel Proof-Number Search (RP–PNS) as proposed in this contribution adheres to the principle of randomized parallelization. The specific probability distribution is obviously based on the selection heuristic.

The master-servant design of ParaPDS and randomized parallelization may be compared as follows. ParaPDS maintains a privileged master thread; only the master thread operates on the top level tree; the master thread selects the subtree in which the servant threads search; it also coordinates the results of the servant processes; each servant thread maintains a separate transposition table in memory. In randomized parallelization there is no master thread; each thread is guided by its own probabilities for selecting the branch to explore; there is

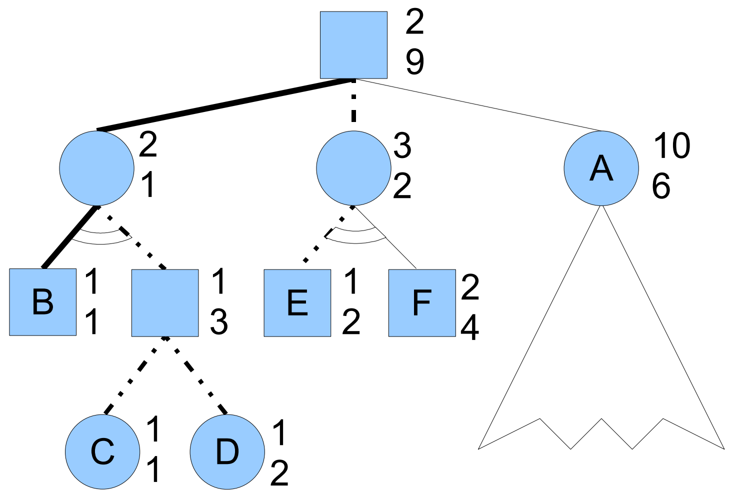

Fig. 2. Example of a PNS tree. Squares represent OR nodes; circles represent AND nodes. Depicted next to each node are its proof number at the top and its disproof number at the bottom.

no communication overhead but instead there is synchronization overhead; all threads can operate on the same tree which is held in shared memory.

4 RP–PNS

This section introduces RP–PNS. Subsection 4.1 explains the basic functioning of RP–PNS and describes how it differs from ParaPDS. Subsection 4.2 explains details of an implementation of RP–PNS.

4.1 Detailed Description of Randomized Parallelization for PNS

There are two kinds of threads in RP–PNS: (1) principal-variation (PV) threads, and (2) alternative threads. RP–PNS maintains one PV thread; all other threads operating on the search tree are alternative threads.

The PV thread always applies the same selection strategy as sequential PNS. It thereby operates on the PV, i.e., the path from root to leaf following the heuristic for finding the most-proving node. We call this selection strategy *PV selection strategy* and a child on the PV, a *PV node*.

The alternative threads select a node according to a modified selection strategy. Instead of minimizing the successor number, there is a chance that a suboptimal successor number is accepted. A probability distribution in the heuristic creates the desired effect: the expanded nodes are always close to the PV since nodes expanded in alternative threads would likely be on the PV at a later cycle. The alternative threads anticipate a possible future PV. The probability of a suboptimal node to be selected for an alternative thread depends on the degree by which it deviates from the PV. In the selection step, alternative threads consider only a subset of all children. The considered children have a successor number at most D larger than the best successor number. An alternative thread selects one of these children by a certain probability.

To account for the move selection more formally, we first introduce further notation. Similarly, for some positive natural number c, we can count the children N_i with successor number $sn(N)$ smaller than or equal to some positive integer c. This count is $cnt(N, c) = |\{N_i : bsn(N_i) \leq c, \text{for } i = 1, ..., |N|\}|$.

Let T be the number of threads used. For each thread $\theta_t, t = 1, ..., T$ and node N there is a probability distribution $P_{\theta_t,N}$ that assigns a probability $p(\theta_t, N, N_i)$ to each child N_i of N. This is the probability of node N_i to be selected as successor node. Equation 1 defines the probability for selecting N_i at N. θ_1 is the PV thread.

$$p(\theta_t, N_i) = \begin{cases} 0 & : \text{ if } \quad t = 1 \wedge sn(N_i) > min(N) \\ cnt(N, min(N))^{-1} & : \text{ if } \quad t = 1 \wedge sn(N_i) = min(N) \\ 0 & : \text{ if } \quad t \neq 1 \wedge sn(N_i) > min(N) + D \\ cnt(N, min(N) + D)^{-1} & : \text{ if } \quad t \neq 1 \wedge sn(N_i) \leq min(N) + D \end{cases} \tag{1}$$

The parameter D in Equation 1 regulates the degree to which the alternative threads differ from the PV. Setting $D = 0$ will result in the PV selection strategy.[5] Setting D too high results in threads straying too far from the PV.

Figure 2 illustrates the consequences of varying the parameter D. In this example, the PV is represented by the bold line and reaches leaf B. An alternative selection with $D = 1$ is represented by the bold, dotted line. It will select one of the leafs B, C, D, or E with equal probability. Setting $D = 2$ will result in also selecting F. We note that the subtree at A is selected only for $D \geq 8$.

In addition to these probabilities, for all alternative threads we assign a second probability of deviating from the PV. This is done by choosing with a probability of $2/d$ randomly from N_2 and N_3 (determined by trial-and-error) instead of choosing N_1 if so far the thread has not deviated from the PV. This choice is determined by the depth d of the last PV. The additional randomization is necessary because it enables sufficient deviation from the PV in case that D is not large enough to produce any effect.

We remark that RP–PNS differs from the original randomized parallelization with respect to three points. (1) Shoham and Toledo do not distinguish between PV and alternative threads. (2) The original randomized parallelization selects children with a probability proportionate to their best-first value while RP–PNS uses an equidistribution for the best candidates. (3) The original randomized parallelization does not rely on a second probability. The differences in point 2 and 3 are based on the desire to produce more deviations from the PV

[5] More precisely, this is true if there exists exactly one child N_i with $sp(N_i) = bsn(N)$. If multiple children have the same best successor number, the alternative threads can deviate from the PV which we assume to be selected deterministically in PNS.

in order to avoid that too many threads congest the same subtree. The selection in RP–PNS is similar to Buro's selection of a move from an opening book [5].

In RP–PNS multiple threads operate on the same tree. To facilitate the parallel access some complications in the implementation require our attention. The next subsection gives details of the actual implementation of RP–PNS.

4.2 Implementation

As pointed out in the previous subsection, all threads in RP–PNS operate on the same search tree held in shared memory. In order to prevent errors in the search tree, RP–PNS has to synchronize the threads. This is achieved in the implementation by a locking policy. Each tree node has a lock. It guarantees that only one thread at a time operates on the same node while avoiding deadlocks. The locking policy consists of two parts: (1) when a thread selects a node, it has to lock it; (2) when a thread updates a node N it has to lock N and its parent. The new values for N are computed. After N has been updated, it is released and the updating continues with the parent.

Each node N maintains a set of flags, one for each thread, to facilitate the deletion of subtrees. Each flag indicates whether the corresponding thread is in the subtree below N. A thread can delete the subtree of N only if no other thread has set its flag in N.

If a transposition table is used to store proof and disproof numbers, each table entry needs an additional lock. The number of locks for the transposition table could be reduced by sharing locks for multiple entries. Similar policies have been used in parallel Monte-Carlo Tree Search [7] (to which the master-servant design has also been applied [6]).

Synchronization imposes a cost on RP–PNS in terms of memory and time consumption. The memory consumption increases due to the additional locks (per node, 16 bytes for a spinlock and flags, cf. [7]) in each node.

The overhead is partially a synchronization overhead and partially a search overhead. The synchronization overhead occurs whenever a thread has to wait for another thread due to the locking policy or due to the transposition table locking. The search overhead is created by any path that would not have been selected by the sequential PNS and that at the same time does not contribute to find the proof. The following section describes experiments that also test the overhead of RP–PNS.

5 Experiment

This section presents experiments and results for RP–PNS. Subsection 5.1 outlines the experimental setup, Subsect. 5.2 shows the results obtained, and Subsect. 5.3 discusses the findings.

5.1 Setup

We implemented RP–PNS as described in the previous section and tested it on complex endgame positions of Lines of Action (LOA)[6]. We chose LOA because

[6] The test set is available at `http://www.personeel.unimaas.nl/m-winands/loa/`, "Set of 286 hard positions."

it is an established domain for applying PNS. The test set consisting of 286 problems has been applied before frequently [14, 18, 19].

The experiment tests two parallelization methods: RP–PNS and RP–PN^2 (an adaptation of RP–PNS for PN^2) for 1, 2, 4, and 8 threads. The combination of an algorithm with a specific number of threads is called a *configuration* and denoted by indexing the number of threads, e.g., RP–PNS$_8$ is RP–PNS using eight threads. We remark that PNS = RP–PNS$_1$ and RP–PN^2 = RP–PN$^2_1$.

The implementation of RP–PN^2 uses RP–PNS for PN$_1$ and PNS for PN$_2$. The size of PN$_2$ was limited to S^ϵ/T, where S is the size of the PN$_1$ tree, T is the number of threads used, and ϵ is a parameter. So, S^ϵ is the size $\sqrt[3]{S^4}$. This limit is a compromise between memory consumption and speed suitable for the test set. The compromise is faster than using the full S as suggested by Allis *et al.* [1]. Using S for the limit slows down RP–PN^2 disproportionally when many threads are used because the PN_1 tree grows faster in RP–PN^2 than in the sequential PN^2. Moreover, the size of PN_2 grows rapidly resulting in slowing down RP–PN^2. An advantage of using the above limit compared to Breuker's method [2] is that the former is robust to varying problem sizes. The values for the parameters of RP–PNS were set to $D = 5$ and $\epsilon = 0.75$ based on trial-and-error.

The experiments were carried out on a Linux server with eight 2.66 GHz Xeon cores and 8 GB of RAM. The program was implemented in C++.

5.2 Results

Two series of experiments were conducted. The first series tests the efficiency of RP–PNS; the second tests the efficiency of RP–PN^2.

For comparing the efficiency of different configurations, we selected a subset of the 143 problems for which PNS was able to find a solution in less than 30 seconds. This selection enabled us to acquire the experimental results for the series of experiments for RP–PN^2 in a reasonable time. We call the set of 143 problems the comparison set, S_{143}. PNS required an average of 4.28 million evaluated nodes for solving a problem of S_{143} with a standard deviation of 2.9 million nodes.

In the first series of experiments we tested the performance of RP–PNS for solving the positions of S_{143}. The results regarding time, nodes evaluated, and nodes in memory for 1, 2, 4, and 8 threads are given in the upper part of Table 1. We observe that the scaling factor for 2, 4, and 8 threads is 1.6, 2.5, and 3.5, respectively. Based on the results we compute that the search overhead expressed by the number of nodes evaluated is only ca. 33% for 8 threads. This means that the synchronization overhead is responsible for the largest part of the total overhead. Finally, we see that RP–PNS$_8$ uses 50% more memory than PNS.

In the second series of experiments we tested the performance of RP–PN^2. The results regarding time, nodes evaluated, and nodes in memory for 1, 2, 4, and 8 threads are given in the lower part of Table 1. We observe that the scaling factor for 2, 4, and 8 threads is 1.9, 3.4, and 4.7, respectively. Compared to RP–PNS the relative scaling factor of RP–PN^2 is better for all configurations.

Table 1. Experimental results for RP–PNS and PN^2 on S_{143}. The total time is the time required for solving all problems. "Nodes in memory" is the sum of all M_i, where M_i is the maximum number of nodes in memory used for test problem i. Nodes evaluated is the sum of all nodes evaluated all problems. For RP–PNS, this includes evaluations in the PN_2 tree and possible double evaluations when trees are re-searched.

	PNS	RP–PNS$_2$	RP–PNS$_4$	RP–PNS$_8$
Total Time (sec.)	1,679	1,072	682	478
Total scaling factor	1	1.6	2.5	3.5
Total nodes evaluated (million)	612	673	745	815
Total nodes in memory (million)	367	423	494	550

	PN^2	RP–PN$^2_2$	RP–PN$^2_4$	RP–PN$^2_8$
Total Time (sec.)	6,735	3,275	1,966	1,419
Total scaling factor PN^2	1	1.9	3.4	4.7
Total scaling factor compared to PNS	0.25	0.52	0.85	1.18
Total nodes evaluated (million)	2,271	2,426	2,534	2,883
Total nodes in memory (million)	68	68	70	73

The search overhead of RP–PN$^2_8$ is 27% which is comparable to the search overhead of RP–PNS$_8$ (33%, cf. above). At the same time the total overhead of RP–PN$^2_8$ is smaller. This means that the synchronization overhead is smaller for RP–PN$^2_8$ than for RP–PNS$_8$. The reason is that more time is spent in the PN_2 trees. Therefore, the probability that two threads simultaneously try to lock the same node of the PN_1 tree is reduced. Finally, we remark that in absolute terms, RP–PN$^2_8$ is slightly faster than PNS.

Despite the fact that RP–PN^2 has a better scaling factor than RP–PNS, RP–PNS is still faster than RP–PN^2 when the same number of threads is used. However, RP–PN^2 consumes less memory than RP–PNS.

5.3 Discussion

It would be interesting to compare the results of the experiments presented in Subsect. 5.2 to the performance of ParaPDS. However, the direct comparison between the results obtained for ParaPDS and RP–PNS is not feasible because of at least three difficulties.

First, the games tested are different (ParaPDS was tested on Othello, whereas RP–PNS is tested on LOA).

Second, the type of hardware is different. As described in Sect. 3, ParaPDS is designed for distributed memory whereas and RP–PNS is designed for shared memory.

Third, ParaPDS is a depth-first search variant of PNS whereas RP–PNS is not. ParaPDS is slowed down because of the transposition tables in distributed memory.

ParaPDS and RP–PN^2 both re-search in order to save memory. When comparing the experimental results for these two algorithms they appear to scale up in the same order of magnitude on a superfacial glance. On closer inspection, a direct comparison of the numbers would be unfair. ParaPDS and RP–PN^2 parallelize different sequential algorithms. Furthermore, RP–PN^2 parallelizes transposition tables while our implementation of RP–PNS does not not. Moreover, it can be expected that sequential PN^2 profits more from transposition tables than RP–PN^2 because the parallel version would suffer from additional communication and synchronization overhead.

In RP–PN^2 the size of the PN_2 tree determines how much the algorithm trades speed for memory. If the PN_2 is too large, the penalty for searching an unimportant subtree will be too large as well. In our implementation, we chose rather small PN_2 trees because the randomization is imprecise. Moreover, the PN_2 trees are bigger when less threads are used. This explains why RP–PN^2 (with a scaling factor of 4.7) scales better than RP–PNS (with a scaling factor of 3.5). A second factor contributing to the better scaling is the reduced synchronization overhead compared to RP–PNS. This effect is produced by the smaller relative number of waiting threads.

We may speculate that RP–PNS and RP–PN^2 could greatly profit from a more precise criterion for branching from the PV. To that end, it is desirable to find a quick algorithm for finding the k-best nodes in a proof-number tree. Thereby, the true best variations could be investigated.

6 Conclusion and Future Research

In this paper, we introduced a new parallel Proof-Number Search for shared memory, called RP–PNS. The parallelization is achieved by threads that select moves close to the principal variation based on a probability distribution. Furthermore, we adapted RP–PNS for PN^2, resulting in an algorithm we call RP–PN^2.

The scaling factor for RP–PN^2 (4.7) is even better than that of RP–PNS (3.5) but this is mainly because the size of the PN_2 tree depends on the number of threads used. Based on these results we may conclude that RP–PNS and RP–PN^2 are viable for parallelizing PNS and PN^2, respectively. Strong comparative conclusions cannot be made for ParaPDS and RP–PNS.

Future research will address the following four directions. (1) A combined parallelization at PN_1 and PN_2 trees of RP–PN^2 will be tested on a shared-memory system with more cores. (2) A better distribution for guiding the move selection, possibly by including more information in the nodes, will be tested to reduce the search overhead. For instance, the probability of selecting a child N_i could be set to $1-(bsn(N_i)/\sum_{j=1,..,|N|} bsn(N_j))$. (3) The concept of the k-most proving nodes of a proof-number tree and an algorithm for finding these nodes efficiently on a parallelized tree will be investigated. (4) The speedup of reducing the number of node locks by pooling will be investigated.

Acknowledgments. We thank the anonymous referees for their valuable advise. This work is financed by the Dutch Organisation for Scientific Research in the framework of the project Go for Go, grant number 612.066.409.

References

1. L.V. Allis, M. van der Meulen, and H.J. van den Herik. Proof-Number Search. *Artificial Intelligence*, 66(1):91–124, 1994.
2. D.M. Breuker. *Memory versus Search.* PhD thesis, Universiteit Maastricht, 1998.
3. M. Brockington. A Taxonomy of parallel game-tree search algorithms. ICCA Journal, Vol. 19(3):162–174, 1996.
4. M. Brockington and J. Schaeffer. APHID Game-Tree Search. In H.J. van den Herik and J.W.H.M. Uiterwijk, editors, *Advances in Computer Chess 8*, pages 69–92. Universiteit Maastricht, 1997.
5. M. Buro, Toward opening book learning. IJCAI-97 Workshop on Using Games as an Experimental Testbed for AI Research, pages 1–5, 1997.
6. T. Cazenave and N. Jouandeau. On the Parallelization of UCT. In H.J. van den Herik, J.W.H.M. Uiterwijk, M.H.M. Winands, and M. Schadd, editors, *Computer Games Workshop 2007 (CGW 2007)*, MICC Technical Report Series 07-06, pages 93–101, 2007. Universiteit Maastricht, The Netherlands, 2007.
7. G.M.J.B. Chaslot, M.H.M. Winands, and H.J. van den Herik. Parallel Monte-Carlo Tree Search. In *Conference on Computers and Games 2008 (CG 2008)*, volume 5131 of *LNCS*, pages 60–71, 2008.
8. A. Kishimoto. Parallel AND/OR tree search based on proof and disproof numbers. In *5th Games Programming Workshop*, volume 99(14) of *IPSJ Symposium Series*, pages 24–30, 1999.
9. A. Kishimoto and M. Müller. DF-PN in Go: Application to the one-eye problem. In H.J. van den Herik, H. Iida, and E. A. Heinz, editors, *Advances in Computer Games Conference (ACG-10)*, pages 125–141. Kluwer Academic, 2003.
10. U. Lorenz. Parallel controlled conspiracy number search. In M. Monien and R. Feldmann, editors, *Euro-Par*, volume 2400 of *LNCS*, pages 420–430. Springer, 2001.
11. D.A. McAllester. Conspiracy numbers for Min-Max Search. *Artificial Intelligence*, 35(3):287–310, 1988.
12. A. Nagai. A new depth-first search algorithm for AND/OR trees. In *Complex Games Lab Workhshop*, pages 40–45, ETL, Tsuruoka, Japan, 1998.
13. A. Nagai. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications.* PhD thesis, University of Tokio, 2002.
14. J. Pawlewicz and L. Lew. Improving depth-first pn-search: 1+ ϵ trick. In H. J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers, editors, *5th International Conference on Computers and Games*, volume 4630 of *LNCS*, pages 160–170. Computers and Games, Springer, Heidelberg, 2006.
15. J. Schaeffer, N. Burch, Y. Björnsson, N. Burch, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 5844(317):1518–1552, 2007.
16. M. Seo, H. Iida, and J.W.H.M. Uiterwijk. The PN-Search algorithm: application to tsume-shogi. *Artificial Intelligence*, 129(1-2):253–277, 2001.
17. Y. Shoham and S. Toledo. Parallel randomized best-first minimax search. *Artificial Intelligence*, 137(1-2):165–196, 2002.
18. H.J. van den Herik and M.H.M. Winands. Proof-Number Search and its Variants. In *Oppositional Concepts in Computational Intelligence*, pages 91–118. 2008.
19. M.H.M. Winands, J.W.H.M. Uiterwijk, and H.J. van den Herik. PDS-PN: A new proof-number search algorithm: Application to Lines of Action. In J. Schaeffer, M. Müller, and Y. Björnson, editors, *Computers and Games 2002*, volume 2883 of *LNCS*, pages 170–185. Computers and Games, Springer, Heidelberg, 2003.