Massively Parallel Proof-Number Search for Impartial Games and Beyond

Tomáš Čížek[1], **Martin Balko**[1], **Martin Schmid**[1,2]

[1]Department of Applied Mathematics, Charles University, Prague, Czech Republic,
[2]EquiLibre Technologies, Inc.
{cizek, balko, schmid}@kam.mff.cuni.cz

Abstract

Proof-Number Search is a best-first search algorithm with many successful applications, especially in game solving. As large-scale computing clusters become increasingly accessible, parallelization is a natural way to accelerate computation. However, existing parallel versions of Proof-Number Search are known to scale poorly on many CPU cores. Using two parallelized levels and shared information among workers, we present the first massively parallel version of Proof-Number Search that scales efficiently even on a large number of CPUs. We apply our solver, enhanced with Grundy numbers for reducing game trees of impartial games, to the Sprouts game, a case study motivated by the long-standing Sprouts Conjecture. Our algorithm achieves 332.9× speedup on 1024 cores, significantly improving previous parallelizations and outperforming the state-of-the-art Sprouts solver GLOP by four orders of magnitude in runtime while generating proofs 1,000× more complex. Despite exponential growth in game tree size, our solver verified the Sprouts Conjecture for 42 new positions, nearly doubling the number of known outcomes.

Code — https://github.com/cizektom/spots

Introduction

Game-solving is a well-established and difficult task in artificial intelligence, which involves computing outcomes under perfect play of all players, often requiring a deep traversal of vast game trees. Despite these challenges, several classical games have already been solved, including Connect Four (Allis 1988), Gomoku (Allis, van den Herik, and Huntjens 1996), Checkers (Schaeffer et al. 2007), and Othello (Takizawa 2023). *Proof-Number Search* (Allis, van der Meulen, and van den Herik 1994) is among the most successful game-solving algorithms. It is a best-first tree search algorithm designed to compute game outcomes efficiently. This algorithm has been widely used due to its ability to focus on the most promising parts of the game tree, making it particularly effective in domains with large branching factors or unbalanced trees. Beyond games, its variants and related algorithms have been applied in various other settings, for example, in chemistry (Franz et al. 2022), graphical models (Dechter and Mateescu 2007), medicine (Heifets and Jurisica 2012), and theorem proving (Lample et al. 2022).

With increasing computational power, it is natural to pursue parallel versions of Proof-Number Search; yet, despite many efforts, existing variants have failed to scale efficiently across many CPUs. Several studies have suggested using parallel Proof-Number Search or its variants in diverse domains (Franz et al. 2022; Kishimoto, Marinescu, and Botea 2015), but scalability remains a challenge. These limitations led Kishimoto et al. (2012) to pose the problem of developing a well-scaling, massively parallel Proof-Number Search.

Many combinatorial games remain far from being solved, including Chess, Shogi, Go, Cram, or Sprouts. In *Sprouts*, two players alternately connect n given spots in the plane according to simple rules, where the last player unable to make a move loses; see Čížek and Balko (2021) for the implementation. We use this game as our experimental domain, as it features highly unbalanced game trees with large branching factors, making it well-suited for Proof-Number Search. Designed by Conway and Paterson to resist computer analysis (Roberts 2015), Sprouts poses a challenging benchmark: the complexity of its game tree surpasses Chess and Go for relatively small values of n; see Figure 1. The computation of outcomes of Sprouts positions is further motivated by the long-standing *Sprouts Conjecture* (Applegate, Jacobson, and Sleator 1991), which states that the *n-spot position*, consisting of n given spots, is winning for the first player if and only if n is congruent to 3, 4, or 5 modulo 6.

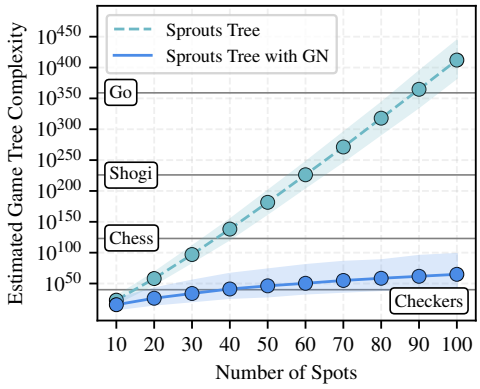

Figure 1: Estimated game tree complexity of Sprouts (with and without Grundy numbers) compared to other games.

Our Contributions

We address the problem posed by Kishimoto et al. (2012) and develop a new variant of Proof-Number Search that scales efficiently on many CPUs. Building on this, we implement a parallel solver for combinatorial games. Together with a new memory-efficient algorithm for impartial games, we achieve numerous new results for the game Sprouts.

Well-Scaling Parallel Proof-Number Search. We introduce *PNS-PDFPN*, a massively parallel Proof-Number Search for distributed-memory systems combining two parallelized levels with shared information among workers. It achieves 208.6× speedup on 1024 cores over sequential DFPN, significantly outperforming prior methods, whose best scaling factor in the same setting was 21.3. Using minimal domain knowledge, the speedup increases to 332.9×.

DFPN Search with Grundy Numbers. Proof-Number Search is designed to solve problems that are represented as AND/OR trees. We formalize the notion of game trees with Grundy numbers to reduce the complexity of game trees of the so-called impartial games, as introduced by Lemoine and Viennot (2012). We adapt *Depth-First Proof-Number Search with Grundy numbers*, a popular and memory-efficient variant of Proof-Number Search, to work with such trees.

Solving Sprouts Positions. To demonstrate the efficiency of our new algorithms, we implemented a solver for the Sprouts game, and used it to determine many new outcomes. Our solver outperforms the state-of-the-art solver *GLOP* by Lemoine and Viennot (2015), achieving a four-order-of-magnitude speedup and generating proofs 1,000 times more complex. Despite the exponential growth of the game trees, our solver verifies the Sprouts Conjecture for 42 new positions, nearly doubling the number of known outcomes.

Related Work

A considerable amount of work has been devoted to designing parallel versions of Proof-Number Search. These include *RP-PNS*, introduced by Saito, Winands, and van den Herik (2010), a parallel variant of DFPN by Kaneko (2010), and *SPDFPN* algorithm by Pawlewicz and Hayward (2014). All of these were designed for shared-memory systems, which typically offer only a limited number of CPU cores. To leverage the greater computational power of clusters with many cores, algorithms tailored to distributed-memory systems are required. Such algorithms include *ParaPDS* by Kishimoto and Kotani (1999), *Job-Level Proof-Number Search* by Wu et al. (2011, 2013), and *PPN*^{2} search by Saffidine, Jouandeau, and Cazenave (2012). A distributed-memory parallel Proof-Number Search was also used by Schaeffer et al. (2005, 2007) in solving Checkers.

As noted by Kishimoto et al. (2012), the scalability of existing algorithms was evaluated only on a relatively small number of cores, and, despite various successes in game solving, their efficiency declined rapidly as core counts increased. While parallel DFPN by Kaneko (2010) and SPDFPN have speedups of 3.58 on 8 cores and 11.8 on 16 cores, respectively, they are limited to shared-memory systems. The best scaling factors for distributed-memory systems are reported for PPN^2 search, which achieved speedups of 7.2, 11.1, and 18.3 on 16, 32, and 64 cores, respectively.

Determining the outcomes of n-spot Sprouts positions has a long history dating back to the 1960s. Conway himself determined the outcomes for $n \leq 3$, and later Mollison extended the results to $n = 6$ by hand (Gardner 1967). The first computer-based solver was developed by Applegate, Jacobson, and Sleator (1991), who used it to prove outcomes for $n \leq 11$. In 2006, Purinton created the program *AuntBeast*, which solved positions up to 14 spots. A major advance was made by the state-of-the-art solver GLOP (Lemoine and Viennot 2015) who used Alpha-Beta Pruning combined with Grundy number theory to solve all positions for $n \leq 32$. These results were later extended to all values of $n \leq 44$ and selected cases up to $n = 53$, by incorporating a basic variant of Proof-Number Search combined with Grundy numbers. All known outcomes agree with the Sprouts Conjecture.

Preliminaries

We now briefly review relevant preliminaries from combinatorial game theory; for more details, see Kishimoto et al. (2012) and Berlekamp, Conway, and Guy (2003). A *combinatorial game* is a two-player, deterministic game with perfect information that ends in a finite number of moves. Under *normal play convention*, the last player able to move wins. A combinatorial game is *impartial* if the available moves from any given position are the same for both players, regardless of whose turn it is. Combinatorial games include Chess, Checkers, Go, Othello, and Shogi, whereas impartial games include Nim, Sprouts, Quarto, and Cram.

Game Trees in Negamax Form

Game trees of combinatorial games can be represented by the so-called *AND/OR trees*. Here, we instead work with AND/OR trees in negamax fashion using *NAND trees*, which are easier to work with, as they eliminate the unnecessary distinction between the AND and OR nodes in subsequent definitions. The nodes at odd and even levels of a NAND tree correspond to positions where the first and second player, respectively, is on the move. In particular, the root corresponds to the initial position with the first player on the move. A *terminal* is a node with no children, which represents a position where the game ends. In a partially expanded NAND tree, a non-terminal node is *internal* if some of its children are generated, and a *leaf* if none of them are.

Each node is associated with one of the following *values*: win, loss, or unknown. The value of a node is *unknown* if the outcome of the associated position is not implied by the currently expanded subtree of the node. Otherwise, it is *win* or *loss*, depending on whether the player on the move at the associated position has a winning strategy or not. Thus, the value of every terminal is loss in any combinatorial game under the normal play convention. The value of an internal node is win if at least one of its children is loss, and it is loss if all its children are wins; otherwise, the value is unknown. The value of a leaf is always unknown. A node is *proved* or *disproved* if its value equals to win or loss, respectively.

Given a position P, the goal is to find a *proof* of P, which is a NAND tree rooted in P whose value is known. These

proofs correspond to the so-called *weak solutions*, as they give the outcome of P and a winning strategy for one of the players. In contrast, *strong solutions* provide outcomes and winning strategies for all positions reachable in the game.

Proof-Number Search

We first describe a basic variant *PNS* of the Proof-Number Search for NAND trees. In each step, PNS selects and expands a *most proving node* (*MPN*)—a leaf in a currently expanded tree whose solution would contribute the most to the solution of the root. To find an MPN, each node v in a NAND tree is associated with a *proof number* $pn(v) \in \mathbb{N}_0 \cup \{\infty\}$ and a *disproof number* $dn(v) \in \mathbb{N}_0 \cup \{\infty\}$, representing the lower bounds on the minimum number of leaves in the subtree of v that have to be solved to prove or disprove v, respectively. Both these values are computed in a bottom-up manner. For leaves, they are initialized as $pn(v) = dn(v) = 1$. For terminals in games under the normal play convention, $pn(v) = \infty$ and $dn(v) = 0$. The values $pn(v)$ and $dn(v)$ of an internal node v are defined as $pn(v) = \min_c dn(c)$ and $dn(v) = \sum_c pn(c)$, where the minimum and the sum are taken over the children c of v. At the start, the algorithm initializes $pn(r)$ and $dn(r)$ of the root r. At each step, it then finds an MPN by descending from the root and always selecting the child with the lowest disproof number, with ties broken arbitrarily. PNS then expands the selected MPN by generating all its children c and initializing their $pn(c)$ and $dn(c)$. At the end of each step, it updates the proof and disproof numbers on the path back to the root.

Depth-First Proof-Number Search. One drawback of PNS is that it stores the entire expanded tree in memory, which is usually consumed very quickly. Therefore, Nagai (2002) introduced *Depth-first Proof-Number Search* (*DFPN*), a space-efficient variant of PNS.

DFPN selects the MPN in a depth-first search manner. Let $\mathcal{P}$ be a currently explored path consisting of internal nodes leading to the MPN. The key idea of DFPN allows one to store only the nodes along $\mathcal{P}$ together with their children while preserving the properties of PNS and staying deep in the tree as long as the MPN is guaranteed to occur there. To achieve that, for each node v of $\mathcal{P}$, DFPN maintains two thresholds $pt(v) \in \mathbb{N}_0 \cup \{\infty\}$ and $dt(v) \in \mathbb{N}_0 \cup \{\infty\}$ for its proof and disproof numbers. The MPN occurs in the subtree of v if and only if $pn(v) < pt(v)$ and $dn(v) < dt(v)$, where $\infty < \infty$ is interpreted as false. Otherwise, DFPN backtracks along $\mathcal{P}$ until it reaches v' that satisfies $pn(v') < pt(v')$ and $dn(v') < dt(v')$, updating the proof and disproof numbers along $\mathcal{P}$. DFPN then resumes MPN selection from v'.

First, set $pt(r) = dt(r) = \infty$ for the root r. Let v be the currently visited node, w and w' be its children with the lowest and second lowest disproof number, where w is the next node to be selected. The thresholds of w are then set as

$$\begin{aligned} pt(w) &= dt(v) - dn(v) + pn(w), \\ dt(w) &= \min\{pt(v), dn(w') + 1\}. \end{aligned} \tag{1}$$

The decreased memory consumption of DFPN comes at the cost of increased time complexity since the proof and disproof numbers of some of the previously visited nodes are lost and potentially need to be recomputed. To balance this trade-off, DFPN is often combined with a transposition table that maintains previously computed values.

Grundy Numbers

As shown by Lemoine and Viennot (2012) and by Beling and Rogalski (2020), the *Sprague–Grundy Theorem* (Grundy 1939; Sprague 1935) can be applied to effectively simplify game trees of impartial games under the normal play convention; see Figure 1. Since we build on these techniques, we briefly recall the necessary background.

Nim is a game that is played on h heaps with $n_1, \ldots, n_h \in \mathbb{N}_0$ objects. With each move, a player must remove at least one object from a single heap, and the first player with no move loses the game. Nim is a strongly solved game, as Bouton (1901) proved that the outcome of Nim is loss if and only if $n_1 \oplus \cdots \oplus n_h = 0$, where $\oplus$ is a bitwise exclusive or.

Let $P_1, \ldots, P_k$ be positions of an impartial game under the normal play convention. Then, the *combination* of $P_1, \ldots, P_k$, denoted by $P_1 + \cdots + P_k$, is a position in which, in each turn, the players decide to move in one of the positions $P_1, \ldots, P_k$ while leaving the other positions untouched. The first player with no move in any of $P_1, \ldots, P_k$ loses in the combination $P_1 + \cdots + P_k$. A position P is *atomic* if it cannot be expressed as a combination of at least two non-empty position; otherwise, it is *decomposable*, in which case $P = P_1 + \cdots + P_k$ for non-empty atomic positions $P_1, \ldots, P_k$ and $k \geq 2$. Two positions P and Q are *equivalent* if, for any position R, the combinations $P + R$ and $Q + R$ have the same outcome.

The *Grundy number* $gn(P)$ of a position P (also called the *nimber* of P) is defined recursively as follows. If P is a terminal position, then $gn(P) = 0$. Otherwise, $gn(P)$ is equal to $\min \mathbb{N}_0 \setminus G$ where G is the set of the Grundy numbers of the children of P. Using $*n$ to denote a Nim position with a single heap of n objects, we formulate the *Sprague–Grundy Theorem* (Grundy 1939; Sprague 1935), which states that each equivalence class of impartial games under the normal play convention can be represented by a unique Grundy number.

Theorem 1 (The Sprague–Grundy Theorem)**.** *Each position P of an impartial game under the normal play convention is equivalent to $*gn(P)$.*

It follows from Theorem 1 that the outcome of $P_1 + \cdots + P_k$ is loss if and only if $gn(P_1) \oplus \cdots \oplus gn(P_k) = 0$. Thus, we can compute the outcome of a decomposable position $P = P_1 + \cdots + P_k$ more efficiently using the Grundy numbers $gn(P_1), \ldots, gn(P_k)$ (Lemoine and Viennot 2012).

Methods

First, we formalize extended NAND trees with Grundy numbers that were implicitly used by Lemoine and Viennot (2012) to efficiently determine the outcomes of decomposable positions. Then, we describe and improve PNS for extended NAND trees by Lemoine (2011) and introduce DFPN for such trees. Finally, we describe our massively parallel version PNS-PDFPN of Proof Number Search.

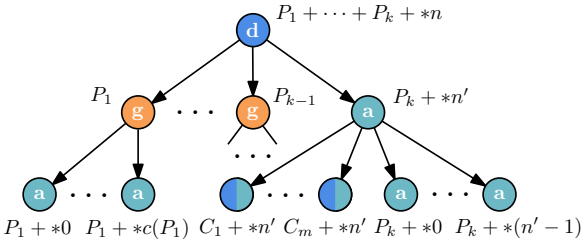

Figure 2: A scheme of a NAND tree with Grundy numbers. We use $C_1, \ldots, C_m$ for the children of P_k and (d), (a), (g) for decomposable, atomic, and Grundy nodes, respectively, and (d/a) for nodes that can be atomic or decomposable.

NAND Trees with Grundy Numbers

To apply the Sprague–Grundy Theorem, we need to determine the Grundy number $gn(P)$ of a position P in an impartial game G under the normal play convention. We efficiently compute $gn(P)$ using the following recursive procedure introduced by Lemoine and Viennot (2012): start with $n = 0$ and increment n until the outcome of the combination $P + *n$, called a *couple*, is loss. Then $gn(P) = n$, because $gn(P) = n$ if and only if the outcome of $P + *n$ is loss.

If P is atomic, then the outcome of the couple $P + *n$ is obtained as before from the outcomes of its children, which are the couples of the form $P' + *n$ and $P + *n'$ where P' is a child of P and $0 \leq n' < n$. The outcome of $P + *n$ with decomposable $P = P_1 + \cdots + P_k$ is obtained by first computing $gn(P_1), \ldots, gn(P_{k-1})$ using the above procedure and then by computing the outcome of the couple $P_k + *n'$, where $n' = n \oplus gn(P_1) \oplus \cdots \oplus gn(P_{k-1})$ and P_k is now atomic.

The nodes of a NAND tree with Grundy numbers correspond either to atomic positions Q of G or to couples $P + *n$ consisting of a position P of G and a Nim position $*n$ with $n \in \mathbb{N}_0$. We have three types of nodes: decomposable, atomic, and Grundy; see Figure 2. A node v corresponding to a couple $P + *n$ is *decomposable* if P is decomposable and *atomic* if P is atomic. Each node u corresponding to an atomic Q is a *Grundy* node. The children of v correspond to the children of $P + *n$ if P is atomic and to Grundy nodes $P_1, \ldots, P_{k-1}$ and the couple $P_k + *n'$ if P is decomposable. The children of u correspond to $Q + *0, \ldots, Q + *c(Q)$, where $c(Q)$ is the number of children of Q.

The value of a node v corresponding to $P + *n$ is again either win, loss, or unknown. For terminals, which correspond to positions $\emptyset + *0$, where $\emptyset$ is the empty position in G, the value is loss. If v is atomic, then its value is determined in the same way as in the original NAND tree with respect to the children of $P + *n$. If v is decomposable, then the value is unknown if v is a leaf, and is equal to the value of the node corresponding to $P_k + *n'$ if v is internal. For a Grundy node u corresponding to Q, the value of u is either a Grundy number $gn(Q)$ or unknown. The value of u equals n if there is a losing child $Q + *n$ of Q, and is unknown otherwise.

Proof-Number Search with Grundy Numbers

To extend PNS to NAND trees with Grundy numbers, we only need to adapt the proof and disproof numbers and specify the next node w to be selected. The resulting algorithm *PNS with GN* then proceeds exactly as PNS.

We define the proof and disproof numbers so that they give lower bounds on the size of the proof of a given position, which guarantees that PNS finds a minimal proof. For leaves, we initialize them to 1. For atomic nodes, the proof and disproof numbers and the next node w are defined as in the original setting. If u is a Grundy node, then it corresponds to an atomic position Q and we use v_i to denote the child of u representing $Q + *i$. The children of u are atomic and are generated incrementally with increasing i, each only after the previous one has been proved, as prescribed by the recursive procedure for the calculation of $gn(Q)$. If v_j is the last child of u generated so far, then we set $pn(u) = dn(u) = \min\{pn(v_j), dn(v_j)\}$ and $w = v_j$. Finally, let v be a decomposable node. Then v corresponds to a couple $P + *n$ where $P = P_1 + \cdots + P_k$ and $k \geq 2$. We use $u_1, \ldots, u_k$ to denote the nodes where each u_i is a Grundy node corresponding to P_i and v'_k is an atomic node corresponding to $P_k + *n'$. Note that only $u_1, \ldots, u_{k-1}, v'_k$ are the children of v, and that in order to generate v'_k, we first need to know the Grundy numbers of $P_1, \ldots, P_{k-1}$, as $n' = n \oplus gn(P_1) \oplus \cdots \oplus gn(P_{k-1})$. Thus, if v'_k has not been generated, we use u_k instead and set $pn(v) = dn(v) = \sum_{i=1}^{k} pn(u_i)$ and w to be the first node from $u_1, \ldots, u_{k-1}$ with unknown Grundy number. Otherwise, we let $pn(v) = pn(v'_k)$, $dn(v) = dn(v'_k)$, and $w = v'_k$.

We improve this version of PNS from Lemoine (2011) by following Schiff, Allis, and Uiterwijk (1994) and merging nodes representing identical states to transform the tree into a directed acyclic graph. This significantly reduces the computation time without increasing memory overhead when combined with storing all touched nodes.

DFPN with Grundy Numbers. We now introduce a new variant of DFPN for extended NAND trees with Grundy numbers (*DFPN with GN*). The algorithm follows the same structure as DFPN for NAND trees, requiring only modifications to the thresholds and the way they are computed.

For each node v from the currently explored path $\mathcal{P}$, we again maintain thresholds $pt(v)$ and $dt(v)$ upper-bounding proof and disproof numbers. However, this time we also need to maintain a new threshold $mt(v) \in \mathbb{N}_0 \cup \{\infty\}$ with parameters $ps(v) \in \mathbb{N}_0$ and $ds(v) \in \mathbb{N}_0$. Intuitively, $mt(v)$ bounds the minimum of $pn(v)$ and $dn(v)$, shifted by $ps(v)$ and $ds(v)$, which is necessary due to decomposable nodes.

To simplify the definition of the thresholds, we replace each Grundy node u in its decomposable parent with the node v_j, which would be selected as the next node if we were in u. Then, we can keep the thresholds only for atomic and decomposable nodes. Now, let v be the currently visited node, and let w be the next node to be selected. For an atomic v, we let w' be its child with the second-lowest disproof number, and we set the thresholds as

$$\begin{aligned} pt(w) &= dt(v) - dn(v) + pn(w), \\ dt(w) &= \min\{pt(v), dn(w') + 1\}, \\ mt(w) &= mt(v), \end{aligned} \tag{2}$$

where $ps(w) = ds(v) + dn(v) - pn(w)$ and $ds(w) = ps(v)$. For a decomposable v, we set $pt(w) = pt(v)$, $dt(w) = dt(v)$, and $mt(w) = mt(v)$ if v'_k has been generated. If v'_k has not been generated yet, we define the thresholds as

$$pt(w) = dt(w) = \infty,$$
$$mt(w) = t(v) - pn(v) + \min\{pn(w), dn(w)\},$$

where $ps(w) = ds(w) = 0$ and $t(v) = \min\{pt(v), dt(v), mt(v) - \min\{ps(v), ds(v)\}\}$. For the root r, we initialize $pt(r) = dt(r) = mt(r) = \infty$ and $ps(r) = ds(r) = 0$. With this choice of thresholds, we obtain the following result.

Theorem 2. *For every node v of $\mathcal{P}$, the MPN is in the subtree of v if and only if $pn(v) < pt(v)$, $dn(v) < dt(v)$, and $\min\{pn(v) + ps(v), dn(v) + ds(v)\} < mt(v)$.*

The proof can be found in Appendix of the full version of this paper (Čížek, Balko, and Schmid 2025). To balance the time complexity and memory consumption of DFPN with GN, we also incorporate a transposition table of proof and disproof numbers, maintained by the replacement strategy proposed by Nagai (2002). In addition, we store the Grundy numbers derived during computation in a separate database. Since this database is much smaller and fits easily into memory, we do not apply any replacement strategy to it.

PNS-PDFPN Algorithm

Our new parallel variant *PNS-PDFPN* of Proof-Number Search operates on two parallelized levels. The first PNS level targets distributed-memory systems and is based on Job-Level Proof-Number Search used by Schaeffer et al. (2007), Saffidine, Jouandeau, and Cazenave (2012), and Wu et al. (2013). At this level, the *master* process maintains the current proof state and repeatedly assigns jobs to *workers* for asynchronous second-level processing. At the second level, each worker performs the parallel DFPN algorithm by Kaneko (2010), which we call *PDFPN*, to utilize shared memory within a single cluster node. We also introduce the sharing of key results between workers to reduce search overhead. We now describe both levels in detail.

First-Level Parallelization. The master and workers typically run on separate nodes of a cluster and communicate over an interconnected network. When a worker becomes idle, the master selects a so-called pseudo-MPN leaf ℓ and assigns it to the worker. The worker then processes ℓ asynchronously until it is solved or the maximum number of expansions is reached. The resulting proof and disproof numbers of ℓ and its children are then sent back to the master, which uses them to expand ℓ in the master tree and updates its proof and disproof numbers. During processing of ℓ, the worker periodically sends updated values $pn(\ell)$ and $dn(\ell)$ to keep the master tree as up-to-date as possible. To prevent reassignment of the same jobs, the assigned leaves are locked, and the definition of proof and disproof numbers is slightly adjusted to avoid misleading attraction to the locked leaves; see Wu et al. (2013) for details.

Similar to Liang, Wei, and Wu (2015) and their JL-UCT Search, our master maintains a database of key computed results shared with all workers to reduce search overhead. Newly derived results are sent to workers along with their assigned jobs, while workers report new results back to the master with each update. Finally, if multiple workers are located on the same node, then they are grouped together to share the same local version of the database.

Second-Level Parallelization. As cluster nodes consist of multiple CPU cores, it becomes advantageous, especially with an increasing number of workers, to start parallelizing the workers themselves, rather than adding more workers that would otherwise process less relevant jobs. This worker reduction also decreases communication overhead and allows sharing of proof and disproof numbers within a node.

Our workers are based on DFPN to make full use of available memory; thus, we parallelize them using PDFPN. In this algorithm, there are as many threads as the number of cores assigned to the worker. Each thread performs an independent DFPN search in the subtree of ℓ, while sharing the lock-protected transposition table of proof and disproof numbers. To decrease redundant thread computation in shared subtrees, the disproof numbers are virtually increased during the selection of the next node w by the number of threads currently computing in the corresponding subtrees. The thresholds in (1) and (2) are then modified by subtracting $th(w)$ from the term $dn(w') + 1$, where $th(w)$ is the number of threads in the subtree of w. Additionally, if a thread solves a position currently computed by a different one, the other thread is notified to backtrack to that position.

Enhancements for Impartial Games. To employ the theory of Grundy numbers, we incorporate PNS with GN and DFPN with GN. We use all computed Grundy numbers as the key shared results, since they are highly reusable and inexpensive to share. To enable parallel computation of multiple children of a decomposable node, the next node w within a decomposable node is selected so that the value $\min\{pn(w), dn(w)\}$ is minimum, rather than always selecting the first unsolved child.

Experiments

We implemented *SPOTS*, a PNS-PDFPN-based solver for combinatorial games, augmented with Grundy numbers to reduce game trees of impartial games effectively. From now on, we use PNS and DFPN to refer to their Grundy-enhanced variants. We evaluate the performance of SPOTS on Sprouts, an impartial game well-suited for Proof-Number Search due to its highly unbalanced game trees, large branching factors, and the fact that it often naturally decomposes into independent subpositions. To incorporate Sprouts into SPOTS, we implemented the string-based position representation from Applegate, Jacobson, and Sleator (1991). Our experiments were carried out on the 29-spot and 47-spot positions, which are the largest that allow each trial to complete within 24 hours. The experimental results are reported as the mean $\pm$ standard error of the mean, based on three independent measurements.

Efficiency of DFPN with GN

In Figure 3, we demonstrate how DFPN with GN allows for tuning the trade-off between computation time and memory

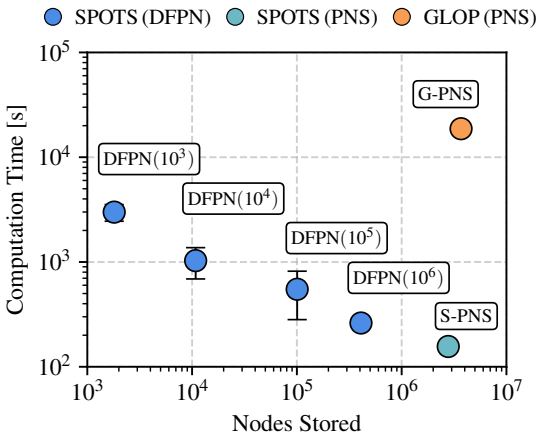

Figure 3: Performance of PNS implementations in GLOP and SPOTS and of DFPN(C) in SPOTS on the 29-spot position, where C is the capacity of the transposition table.

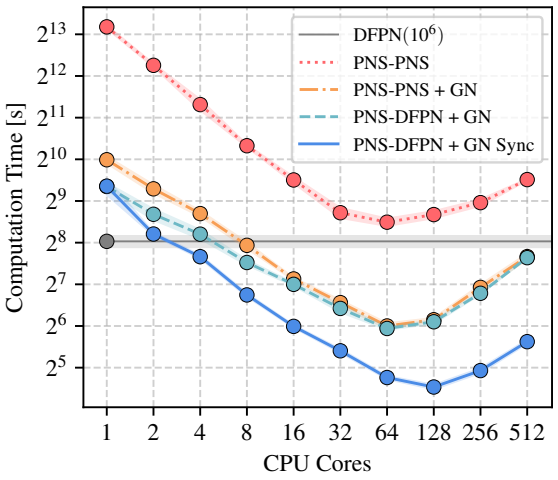

Figure 4: Comparison of PNS-PNS and PNS-DFPN, including the impact of retention of Grundy numbers (GN) in workers and their synchronization (GN Sync). All variants are run on the 29-spot position with 100 `iterations`, 100 `updates`, no grouping, and no second-level parallelization.

consumption. By adjusting the capacity C of the transposition table, memory requirements can be substantially reduced at the cost of a moderate increase in computation time. We observe that the transposition table with $C = 10^6$ is already sufficiently large, as it is not fully saturated. Despite this, our PNS implementation still outperforms this DFPN configuration, as it operates faster on its explicitly expanded tree; however, the PNS tree can grow indefinitely.

We also compare our PNS with the PNS implemented by Lemoine and Viennot (2015) in their solver GLOP. Unlike our method, GLOP operates on a proper tree without transpositions and discards all visited nodes in the solved subtrees to reduce memory usage. Along with a roughly three-times faster state representation, our design choices lead to a roughly 100-fold reduction in computation time compared to GLOP while maintaining a comparable maximum tree size.

Scaling Efficiency of PNS-PDFPN

PNS-PDFPN includes five tunable parameters: `workers`, the total number of workers; `iterations`, the maximum number of expansions per job; `updates`, the number of iterations between each update sent to the master; `grouping`, the number of workers grouped within a single node; and `threads`, the number of threads assigned to PDFPN per worker. The number of CPU cores used by PNS-PDFPN is then equal to `workers` times `threads` plus the number of cores allocated to the master. We now analyze the effect of the worker synchronization and parameter setting on the scaling efficiency of the algorithm, which is reported with respect to the number of CPU cores allocated to workers, excluding the master CPU.

Retaining and Sharing Second-Level Information. The parallel *PPN*^{2} search by Saffidine, Jouandeau, and Cazenave (2012) corresponds to PNS-PNS using PNS at both levels. However, after each job is completed, the second-level PNS search tree is discarded, losing the entire worker state. In contrast, our PNS-DFPN workers retain their transposition tables, completely preserving their state throughout the whole computation. Figure 4 shows how the retention and sharing of the worker state impact scaling efficiency. Although PNS-PNS scales well to up to 64 cores, worker state resets hamper its performance so significantly that it is outperformed by sequential DFPN(10^6). However, when the key results are retained, specifically the computed Grundy numbers, the speedup improves significantly. Further performance gains can be achieved by PNS-DFPN preserving computed proof and disproof numbers. However, this has a smaller effect than retaining the Grundy numbers. Finally, sharing Grundy numbers among workers yields a further significant performance gain.

PNS-PDFPN Ablations. We evaluate the impact of the PNS-PDFPN parameters on the scaling efficiency, measured relative to the runtime of DFPN($5 \cdot 10^7$), serving as a baseline for comparing various parallel Proof-Number Search algorithms. We compare PNS-PDFPN against PDFPN by Kaneko (2010) and the PNS-DFPN scheme employed by Schaeffer et al. (2007) to solve Checkers, which, according to our measurements, represent state-of-the-art approaches for shared- and distributed-memory systems, respectively.

In Figure 5, we incrementally add parameters until we arrive at the full PNS-PDFPN with all the improvements. For each setup, we report the speedup achieved using the best-performing parameter setting, where PNS-DFPN is already optimized for `iterations` and `updates`, as proposed by Saffidine, Jouandeau, and Cazenave (2012). Without GN synchronization, PNS-DFPN scales very poorly. Enabling synchronization improves scalability up to 512 cores, but then the performance degrades due to significant synchronization overhead. This issue is overcome by grouping workers, reducing the amount of shared results, and thus the

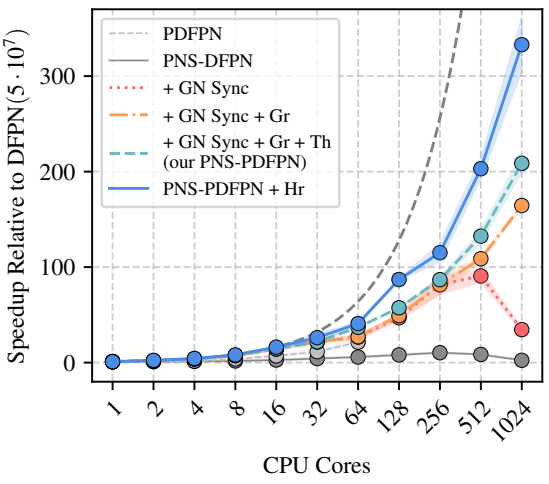

Figure 5: Impact of GN synchronization (GN Sync), grouping (Gr), second-level parallelization (Th), and the child-ordering heuristic (Hr) on the final speedup of PNS-PDFPN relative to DFPN($5 \cdot 10^7$), which finished in 293 ± 41.7 minutes. Measured on the 47-spot position.

	Cores	Parameters		Speedup	
		It	Th	PNS-PDFPN	+ Heuristic
Best Values for No-Heuristic	1	1k	1	**1.08** ± 0.10	0.99 ± 0.00
	2	1k	1	1.87 ± 0.02	**2.36** ± 0.28
	4	1k	1	3.91 ± 0.31	**4.34** ± 0.32
	8	1k	1	**7.41** ± 0.21	7.10 ± 0.53
	16	10k	1	**14.37** ± 0.86	10.15 ± 4.48
	32	10k	1	21.75 ± 0.68	**23.26** ± 0.05
	64	1k	4	36.97 ± 0.29	**37.93** ± 0.21
	128	1k	8	**57.43** ± 0.05	45.76 ± 2.32
	256	1k	8	86.87 ± 1.42	**92.35** ± 1.57
	512	1k	8	**132.44** ± 6.11	123.72 ± 3.97
	1024	1k	16	**208.67** ± 9.17	169.24 ± 2.86
Best Values for Heuristic	1	1k	1	**1.08** ± 0.10	0.99 ± 0.00
	2	1k	1	1.87 ± 0.02	**2.36** ± 0.28
	4	1k	1	3.91 ± 0.31	**4.34** ± 0.32
	8	10k	1	7.36 ± 0.41	**8.06** ± 0.09
	16	100k	1	11.90 ± 0.94	**16.22** ± 0.53
	32	100k	4	18.50 ± 0.76	**26.13** ± 3.87
	64	100k	8	27.95 ± 2.19	**40.86** ± 1.13
	128	100k	4	29.44 ± 0.53	**87.02** ± 0.81
	256	100k	8	54.81 ± 3.71	**115.11** ± 8.88
	512	100k	16	73.21 ± 6.36	**203.05** ± 9.82
	1024	100k	32	97.60 ± 9.24	**332.97** ± 26.8

Table 1: Best-performing values of `iterations` (It) and `threads` (Th) for PNS-PDFPN without (top) and with (bottom) the heuristic. `Grouping` set to its maximum and `updates` to 1,000 are optimal across all experiments.

synchronization overhead. Further scalability is achieved by introducing second-level parallelization, which ultimately leads to a speedup of 208.67 ± 9.17 without relying on any domain knowledge, which exceeds even the best speedup of 21.35 ± 2.40 achieved by PDFPN using shared memory.

We observe a notable improvement in scaling efficiency after incorporating the child-ordering heuristic proposed by Lemoine and Viennot (2015) to break ties during the selection of the next node w. Adding the heuristic does not enhance the overall performance if applied to optimally tuned parameters without the heuristic, as shown in the upper part of Table 1. However, the heuristic becomes beneficial when the parameters are tuned specifically for its use; see the bottom part of Table 1. The heuristic then enables more effective utilization of second-level parallelism in longer-running jobs by providing better guidance to branches that are more likely to yield shorter proofs. With this modest domain knowledge, the final speedup of PNS-PDFPN reaches 332.97 ± 26.8. Further parameter analysis and hardware specification are given in Appendix.

Solving Sprouts Positions

With SPOTS, we determined 32 new outcomes of n-spot positions using the sequential version and 10 more with its parallel counterpart, nearly doubling the number of previously known results; see Appendix. We verified the new proofs using the verification software by Lemoine and Viennot (2015). All computed outcomes agree with the Sprouts Conjecture, which thus remains open. Compared to the previous state-of-the-art solver GLOP, our parallel version of SPOTS achieves a speedup of four orders of magnitude on 1024 cores. We base this estimate on the combined improvements of the sequential SPOTS over GLOP and the parallel SPOTS over the sequential version. To illustrate the speedup, SPOTS solves the 39-spot position, one of the most complex solved by GLOP, in just 7.7 minutes using 1024 cores. The outcomes determined by parallel SPOTS are much more complex than those previously computed. The most difficult position was solved in 24 days on 512 cores, and its proof size (the number of stored Grundy numbers) exceeds prior ones by a factor of 1,000.

Conclusion

While our experiments focus on the impartial game Sprouts, PNS-PDFPN is domain-independent and applicable to any combinatorial game. For non-impartial games, the solver naturally reduces to exploring standard NAND trees, where computed Grundy numbers correspond to loss outcomes. Position decomposition is the only impartial-specific component, used as an optional acceleration in all baselines, and therefore does not affect overall scalability. Scalability arises from multi-level parallelization and intermediate-result sharing, both of which are general. Sprouts offers one advantage for scaling: its relatively slow expansion rate allows frequent synchronization. However, this does not favor our algorithm, which strongly outperforms state-of-the-art methods in the same setting. In faster-expanding domains, we suggest using costlier search-guiding heuristics in exchange for improved scalability. Lastly, since even minimal domain knowledge for job assignment notably boosts scaling, more sophisticated use of domain knowledge, such as RL-based policies, may yield further speedups.

Acknowledgments

All authors were supported by the grant no. 25-18031S of the Czech Science Foundation (GAČR). T. Čížek was supported by the Charles University Grant Agency (GAUK) project no. 326525. M. Schmid was also supported by the Charles University project UNCE 24/SCI/008. The authors thank EquiLibre Technologies, Inc. for providing computational resources. Additional computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic. Special thanks to Neil Burch for valuable comments and insights.

References

Allis, L.; van der Meulen, M.; and van den Herik, H. 1994. Proof-number search. *Artificial Intelligence*, 66(1): 91–124.

Allis, L. V. 1988. *A Knowledge-Based Approach of Connect Four: The Game is Over*. M.Sc. thesis, Vrije Universiteit Amsterdam.

Allis, L. V.; van den Herik, H. J.; and Huntjens, M. P. H. 1996. Go-Moku Solved by New Search Techniques. *Computational Intelligence*, 12(1): 7–23.

Applegate, D.; Jacobson, G.; and Sleator, D. 1991. Computer Analysis of Sprouts. Technical report, Carnegie Mellon University.

Beling, P.; and Rogalski, M. 2020. On pruning search trees of impartial games. *Artificial Intelligence*, 283: 103262, 16.

Berlekamp, E. R.; Conway, J. H.; and Guy, R. K. 2003. *Winning Ways for Your Mathematical Plays. Vol. 3*. A K Peters.

Bouton, C. L. 1901. Nim, A Game with a Complete Mathematical Theory. *Ann. of Math. (2)*, 3(1-4): 35–39.

Čížek, T.; and Balko, M. 2021. Implementation of Sprouts: A Graph Drawing Game. In Purchase, H. C.; and Rutter, I., eds., *Graph Drawing and Network Visualization*, 391–405. Cham: Springer International Publishing.

Čížek, T.; Balko, M.; and Schmid, M. 2025. Massively Parallel Proof-Number Search for Impartial Games and Beyond. arXiv:2511.10339.

Dechter, R.; and Mateescu, R. 2007. AND/OR search spaces for graphical models. *Artificial Intelligence*, 171(2): 73–106.

Franz, C.; Mogk, G.; Mrziglod, T.; and Schewior, K. 2022. Completeness and Diversity in Depth-First Proof-Number Search with Applications to Retrosynthesis. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence (IJCAI-22)*, 4747–4753. International Joint Conferences on Artificial Intelligence Organization.

Gardner, M. 1967. Mathematical Games: of Sprouts and Brussels Sprouts; Games with a Topological Flavour. *Sci. Amer.*, 217: 112–115.

Grundy, P. M. 1939. Mathematics and Games. *Eureka*, 2: 6–8.

Heifets, A.; and Jurisica, I. 2012. Construction of New Medicines via Game Proof Search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI Conference on Artificial Intelligence, 1564–1570.

Kaneko, T. 2010. Parallel Depth First Proof Number Search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI Conference on Artificial Intelligence, 95–100.

Kishimoto, A.; and Kotani, Y. 1999. Parallel AND/OR Tree Search Based on Proof and Disproof Numbers. In *Proceedings of the 5th Game Programming Workshop*, volume 99, 24–30.

Kishimoto, A.; Marinescu, R.; and Botea, A. 2015. Parallel Recursive Best-First AND/OR Search for Exact MAP Inference in Graphical Models. In Cortes, C.; Lawrence, N.; Lee, D.; Sugiyama, M.; and Garnett, R., eds., *Advances in Neural Information Processing Systems (NIPS-15)*, volume 28. Curran Associates, Inc.

Kishimoto, A.; Winands, M. H. M.; Müller, M.; and Saito, J. T. 2012. Game-Tree Search Using Proof Numbers: The First Twenty Years. *ICGA Journal*, 35(3): 131–156.

Lample, G.; Lacroix, T.; Lachaux, M.; Rodriguez, A.; Hayat, A.; Lavril, T.; Ebner, G.; and Martinet, X. 2022. HyperTree Proof Search for Neural Theorem Proving. In Koyejo, S.; Mohamed, S.; Agarwal, A.; Belgrave, D.; Cho, K.; and Oh, A., eds., *Advances in Neural Information Processing Systems*, volume 35, 26337–26349. Curran Associates, Inc.

Lemoine, J. 2011. *Méthodes Algorithmiques pour la Résolution des Jeux Combinatoires*. Ph.D. thesis, Université des Sciences et Technologie de Lille - Lille I. In French.

Lemoine, J.; and Viennot, S. 2012. Nimbers are inevitable. *Theoret. Comput. Sci.*, 462: 70–79.

Lemoine, J.; and Viennot, S. 2015. Computer Analysis of Sprouts with Nimbers. In *Games of no chance 4*, volume 63 of *Math. Sci. Res. Inst. Publ.*, 161–181. Cambridge Univ. Press, New York.

Liang, X.; Wei, T.; and Wu, I.-C. 2015. Job-level UCT search for solving Hex. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*.

Nagai, A. 2002. *Df-pn Algorithm for Searching AND/OR trees and its Applications*. Ph.D. thesis, The University of Tokyo.

Pawlewicz, J.; and Hayward, R. B. 2014. Scalable Parallel DFPN Search. In van den Herik, H. J.; Iida, H.; and Plaat, A., eds., *Computers and Games*, 138–150. Cham: Springer International Publishing.

Roberts, S. 2015. *Genius at Play: The Curious Mind of John Horton Conway*. New York: Bloomsbury Press.

Saffidine, A.; Jouandeau, N.; and Cazenave, T. 2012. Solving Breakthrough with Race Patterns and Job-Level Proof Number Search. In van den Herik, H. J.; and Plaat, A., eds., *Advances in Computer Games*, 196–207. Berlin, Heidelberg: Springer Berlin Heidelberg.

Saito, J.-T.; Winands, M. H. M.; and van den Herik, H. J. 2010. Randomized Parallel Proof-Number Search. In van den Herik, H. J.; and Spronck, P., eds., *Advances in Computer Games*, 75–87. Berlin, Heidelberg: Springer Berlin Heidelberg.

Schaeffer, J.; Bjoernsson, Y.; Burch, N.; Kishimoto, A.; Mueller, M.; Lake, R.; Lu, P.; and Sutphen, S. 2005. Solving

Checkers. In Kaelbling, L. P.; and Saffotti, A., eds., *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, 292–297. International Joint Conferences on Artificial Intelligence Organization.

Schaeffer, J.; Burch, N.; Björnsson, Y.; Kishimoto, A.; Müller, M.; Lake, R.; Lu, P.; and Sutphen, S. 2007. Checkers is Solved. *Science*, 317(5844): 1518–1522.

Schiff, M.; Allis, L. B.; and Uiterwijk, J. W. H. M. 1994. Proof-Number Search and Transpositions. *ICCA Journal*, 17(2): 63–74.

Sprague, R. 1935. Über Mathematische Kampfspiele. *Tohoku Math. J., First Series*, 41: 438–444.

Takizawa, H. 2023. Othello is Solved. arXiv:2310.19387.

Wu, I.-C.; Lin, H.-H.; Lin, P.-H.; Sun, D.-J.; Chan, Y.-C.; and Chen, B.-T. 2011. Job-Level Proof-Number Search for Connect6. In van den Herik, H. J.; Iida, H.; and Plaat, A., eds., *Computers and Games*, 11–22. Berlin, Heidelberg: Springer Berlin Heidelberg.

Wu, I.-C.; Lin, H.-H.; Sun, D.-J.; Kao, K.-Y.; Lin, P.-H.; Chan, Y.-C.; and Chen, P.-T. 2013. Job-Level Proof Number Search. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(1): 44–56.