Scalable Parallel DFPN Search

Jakub Pawlewicz[1] and Ryan B. Hayward[2]

[1] Institute of Informatics, University of Warsaw, pan@mimuw.edu.pl
[2] Computing Science, University of Alberta, hayward@ualberta.ca

Abstract. We present Scalable Parallel Depth-First Proof Number Search, a new shared-memory parallel version of depth-first proof number search. Based on the serial DFPN $1+\varepsilon$ method of Pawlewicz and Lew, SPDFPN searches effectively even as the transposition table becomes almost full, and so can solve large problems. To assign jobs to threads, SPDFPN uses proof and disproof numbers and two parameters. SPDFPN uses no domain-specific knowledge or heuristics, so it can be used in any domain. Our experiments show that SPDFPN scales well and performs well on hard problems.
We tested SPDFPN on problems from the game of Hex. On a 24-core machine and a 4.2-hour single-thread task, parallel efficiency ranges from 0.8 on 4 threads to 0.74 on 16 threads. SPDFPN solved all previously intractable 9×9 Hex opening moves; the hardest opening took 111 days. Also, in 63 days, it solved one 10×10 Hex opening move. This is the first time a computer or human has solved a 10×10 Hex opening move.

1 Introduction

Depth-First Proof-Number search is effective for solving problems — e.g. two-player games — that can be modelled with an and-or tree. It is especially effective on trees with non-uniform branching, and has been successful in checkers [23], shogi [9], tsume-shogi [12], Go [13, 14, 25], and Hex [8, 1].

DFPN search often jumps around the tree. An effective parallel DFPN variant thus needs a shared *transposition table* (tt). The starting point for Scalable Parallel DFPN search, our shared-memory DFPN variant, is the serial $1+\varepsilon$-method [19], which is effective when search space exceeds memory.

Other parallel variants based on proof numbers have been proposed. Nagai introduced proof-disproof search [16] and Kishimoto gave a parallel version [11]. Saffidine et al. proposed a job-level PN^2 [21] which is effective when search space exceeds memory, although nodes can be recomputed many times. Kaneko achieves a parallel efficiency of 0.5 with 8 threads in tsume-shogi [9], whereas SPDFPN achieves 0.7 in Hex. Hex solvers tend to obtain better speedups than tsume-shogi solvers, perhaps because Hex has a larger branching factor and so smaller node expansion rate.

Saito et al. introduced randomized parallel PNS [22]. Wu et al. [24] use PNS with virtual proof and disproof numbers and a time-intensive leaf initiation in a tree that is small enough to stay in memory. SPDFPN is inspired by these virtual numbers, but is based on DFPN rather than PNS. It distributes work among threads using only proof and disproof numbers and thus needs no game-specific heuristic.

We tested SPDFPN on two sets of Hex problems, including all thirteen previously intractable 9×9 opening moves. See Figure 1. Experiments show that our algorithm scales well up to at least 16 threads.

2 DFPN

2.1 PN Search

Proof number search maintains a tree in which each node — corresponding to a game position — has a *proof number* (pn) and *disproof number* (dn). A node's (dis)proof number is the smallest number of leaves that, if all true (false), would make the node true (false). Thus for an or-node $p = \min_j p_j$ and $d = \sum_j d_j$, where p (d) is the node's (dis)proof number, and p_j (d_j) the j'th child's (dis)proof number. PN search relies on the existence of a most-proving node (mpn): this node's (dis)proof reduces either the proof or disproof number. PN search iteratively selects a most-proving leaf and expands it. PN search needs the whole search tree to be in memory. This restriction can be mitigated by taking a depth-first search approach and using a tt.

2.2 Depth-first PN Search

In computing (dis)proof numbers, only descendant (dis)proof values are needed. DFPN search exploits this property by postponing ancestor updates until the most-proving node is no longer in the current node's subtree. For each node, thresholds P and D are defined so that $p < P$ and $d < D$ if and only if there exists a most-proving node in the node's subtree. These thresholds are computed recursively using the following formula for an or-node. Children are indexed in non-decreasing proof number, e.g. $p_1(d_1)$ are the (dis)proof numbers of a child with smallest proof number. See [18] or [19] or the recent survey on game tree search using pns [10].

$$P_1 = \min(P, 1 + p_2), \tag{1}$$

$$D_1 = D - (d - d_1). \tag{2}$$

3 Transposition table and DAGs

To this point we have described the process for computing (dis)proof numbers in trees. Many games allow transpositions, and so are modelled by *directed acyclic graphs* (dags) rather than trees. While more complicated variants of DFPN are available for dags [12], we find that in Hex, good results are obtained by treating the dag as a tree, as long as a tt is used.

3.1 Problems, memory and the $1 + \varepsilon$ method

Even with a tt, DFPN search behaves differently from the usual depth-first search, as the continued selection of the next mpn causes frequent switching among child branches.

To force DFPN to switch branches less often, one can use the $1+\varepsilon$ method [19], which simply enlarges the pn threshold in (1):

$$P_1 = \min(P, \lceil (1+\varepsilon)p_2 \rceil).$$

Another problem arises in (d)pn calculations when dags are treated as trees: due to transpositions, (d)pns can be (exponentially) overcounted. Miscounting can lead to the mistaken selection of a node which is not most-proving. Techniques address this issue in games, such as tsume-shogi, with many transpositions [12]. We chose rather to address these issues by starting with the $1+\varepsilon$ method, which reduces branch switching even if search space is large. For our Hex experiments, possibly miscounting (dis)proof numbers did not cause difficulties. After trial and error, we set ε to 0.25.

4 Parallelization

The version of PN search we have described to this point behaves like a usual depth-first algorithm, spending significant periods of time in deep recursive calls. This allows for parallelization if these criteria, which motivate the design of SPDFPN, can be met:

(i) Different threads should call DFPN searches for different states.
(ii) A thread should not duplicate the work of another. (The threads share a tt, and different threads might explore different strategies for the same state.)
(iii) Assignment of states to threads should follow the natural order of PN search. (Assume that state A is assigned to thread α. If state B is likely to be the next state considered, we can assign B to another thread before α finishes.)
(iv) The assignment of states to threads should take little time.
(v) A thread should exit its search once other thread results render it unnecessary.

4.1 Measuring work

We now show how we realize our criteria. Because of (v) we want to allow individual threads to halt and perhaps later resume. A straightforward attempt to parallelize DFPN that assigns a node to a processor, and has the processor return only when the subtree is solved, would not permit this.

To realize (v), we set a work threshold for a single DFPN call. The threshold (*MaxWorkPerJob*) must be small enough to allow even distribution among threads, but not so small that we lose (iv). If it is large enough we satisfy (iv) and can use more sophisticated methods for state assignment among threads. We need such a threshold, since (d)pns in a DFPN search can remain low and not reach any (d)pn thresholds for long periods of time, especially in sudden-death games.

Implementation So how do we alter DFPN to incorporate the work threshold? In addition to the DFPN threshold parameters P and D, we introduce a new threshold parameter W, the maximum work that a thread should perform before halting. We define work as the number of calls to the DFPN function.

The $1+\varepsilon$ method works best with advanced tt collision resolution. We use a method from computer chess (e.g. [15, 20]) with $k = 4$: upon collision, search the next at most k cells for an empty location; if none is found, overwrite the location whose job has performed the smallest amount of work. See [3, 17, 18] for other replacement or collection techniques.

SPDFPN pseudocode in Algorithms 1 and 2 uses these variables: c — array of child nodes; P, D — proof and disproof number thresholds; W — work threshold; n — node. A node has fields $.p$, $.d$ (proof and disproof numbers), $.w$ (work), $.j$ (index of last selected child). TTWRITE(n) writes results for node n in the tt. TTREAD(n, j) tries to read the jth child of node n; if this fails, it creates a child node with (d)pns each set to 1 and work set to 0.

Algorithm 1. DFPN Search

```
1:  function DFPN(n, P, D, W)
2:      w_local ← 1, n.w ← n.w + 1
3:      for each child j of n do
4:          c[j] ← TTREAD(n, j)
5:      loop
6:          PNUPDATE(n, c)
7:          TTWRITE(n)
8:          if n.p ≥ P ∨ n.d ≥ D then
9:              return w_local
10:         if w_local ≥ W then
11:             return w_local
12:         j, P_j, D_j ← SELECT(n, c, P, D)
13:         w_child ←
14:             DFPN(c[j], P_j, D_j, W − w_local)
15:         w_local ← w_local + w_child
16:         n.w ← n.w + w_child
```

DFPN returns the amount of work done locally, i.e. in this call and all recursive calls. SELECT returns a child together with thresholds for recursive call. In the single-threaded version of the $1 + \varepsilon$ method, processing remains at a node n_1 until its pn p_1 exceeds the second-smallest pn p_2 (among siblings) by a ratio of $1+\varepsilon$. Since we allow processes to be interrupted, it can be that upon resumption p_1 is larger than p_2 but smaller than $p_2(1 + \varepsilon)$. In this case, processing should resume at n_1 rather than its sibling. This requires adding to function SELECT the if-statement in lines 6–11.

5 Work assignment

What are a thread's candidates for state assignment? PN-Search descends through a path from root to a most-proving node. In DFPN this path is created by successive recursive calls and so is stored on a stack. However, DFPN search can remain deep in the search tree for long periods. Thus for a node whose path is close to the most-proving node, DFPN can stay in a subtree of that node for some time. Such a node is a good candidate for a thread state assignment. But how deep on the subtree path should the assigned state be? It should be deep enough that DFPN will stay in subtree of the node, not too shallow because of (v) and not too deep because of (iv). A good candidate is a node that is closest to the root and with past work performed below *MaxWorkPerJob*, because we expect that total DFPN work for this node will be proportional to the total DFPN past work.

Once we assign a state A to a thread α, how should we assign a state B to another thread? If we follow the same procedure we would arrive at the same state. Instead, following Rémi Coulom (see [4, p. 64]), we temporarily assign a virtual win or loss to A until α finishes its search. This idea is also used by Job Level PN search, which achieves superlinear scalability, and fulfils (i), (ii) and (iii). See [24].

Algorithm 2. DFPN Search — utility functions for OR node

```
1: procedure PNUPDATE(n, c)
2:     n.p ← min_{child j of n} c[j].p
3:     n.d ← Σ_{child j of n} c[j].d
4: function SELECT(n, c, P, D)
5:     j_1 ← child with the smallest pn
6:     if n.j is set and n.j ≠ j_1 then
7:         ▷ Try continue with the same child
8:         P_{n.j}, D_{n.j} ←
9:             THRESHOLDS(n, c, P, D, n.j, j_1)
10:        if c[n.j].p < P_{n.j} then
11:            return n.j, P_{n.j}, D_{n.j}
12:    n.j ← j_1
13:    j_2 ← child with the second smallest pn
14:    P_{j_1}, D_{j_1} ←
15:        THRESHOLDS(n, c, P, D, j_1, j_2)
16:    return j_1, P_{j_1}, D_{j_1}
17: function THRESHOLDS(n, c, P, D, j_1, j_2)
18:    P_{j_1} ← min(P, ⌈(1 + ε) · c[j_2].p⌉),
19:    D_{j_1} ← D − (n.d − c[j_1].d)
20:    return P_{j_1}, D_{j_1}
```

5.1 Virtual proof and disproof numbers

As we set the value of node A to virtual win or loss depending on its (d)pn, we must update other (d)pns along the path to the root. If we modify existing (d)pns, then other threads can reach a state with incorrect (d)pns via transposition. So the use of virtual win/loss requires the use of virtual (d)pns.

An inaccurate assignment of a virtual win or loss will cause SPDFPN search to diverge from DFPN search, violating condition (iii). Our initial assignments of these virtual win/loss values are often accurate, as they are often made at nodes that have already been partly searched. The amount of previous search is determined by the threshold parameter *MaxWorkPerJob*.

Virtual (d)pns can be stored efficiently (see below). In descending from the root towards a most-proving node in order to find a state assignment for a thread, we use virtual (d)pns if available, otherwise true (d)pns. After assigning a state A to a thread α, we update virtual (d)pns along the path to the root. Solving work then starts by calling DFPN on state A. This call uses true (d)pns. When the call returns, we reset the virtual win/loss back to a true (d)pn and update virtual (d)pns along the path to root. This completes an iteration of a thread loop, i.e. the thread now seeks its state assignment.

The entire phase of candidate-finding is guarded by a lock, so only a single thread can operate on virtual (d)pns at a time. A lock is released only when DFPN is called and solving work resumes.

Virtual (d)pns are kept in a virtual tt, which stores a node's virtual (d)pns and the number of threads assigned on the path to the root that contain that node. A node is added to the virtual tt at most as many times as the number of threads. This tt is easy to

implement for dags. Each level (move) of the game corresponds to an array whose size is at most the number of threads. Virtual tt operations are as follows:

- VTTADD(n): if the entry already exists, increment the counter.
- VTTREMOVE(n): decrement the counter; if 0, remove the entry, otherwise restore n's previous virtual (d)pdn.
- VTTREAD(n, j, n_j): return entry for n's jth child; if no such entry then initialize v(d)pns by returning the default n_j, which contains true (d)pns.

5.2 Finding a state candidate for a thread

Finding a candidate for state assignment can fail, as follows. Suppose we are at node n. We have read its (d)pns from the tt. We must select a child to descend to, so we also read all of its children's (d)pns from the tt. Normally, the recursive formulas should give the (d)pns of n from those of its children. A child's (d)pns can be lost due to tt overwrites, but this is not a problem, as the (d)pns are recalculated when we descend to such child. The problem is that c was reached via transposition and work was done at c after the last update of n. Thus (d)pns can be stale, and an update can reveal that (d)pn thresholds were reached or even that n has been solved.

So descending to a mpn via the usual rules is not sufficient. Instead, we recursively search as in DFPN, using virtual (d)pns whenever they exist, and stopping the search as soon as we find a state with whose previous work performed is below a fixed threshold (*MaxWorkPerJob*). This search is performed by TRYRUNJOB, explained in §5.4.

5.3 Sharing transposition table

Threads share the tts, so we use multiple-reader/single-writer locks. Following [9], if a worker thread discovers immediately before writing that the node has (during the worker's processing) been solved by another thread, we do not overwrite the tt.

5.4 Implementation

Algorithm 3 shows the main scheme of SPDFPN, our parallel DFPN search. In the loop, each thread calls TRYRUNJOB, tries to find a candidate for state assignment, and if successful then runs a job by calling DFPN on the assigned state. But first we need to update all virtual (d)pns, accessing them in nodes on the node-to-root path. We also need virtual (d)pns of children of each such node. So, we introduce a list v directed towards the root. Each list entry contains this data for the associated node: $v.n$ — a node with virtual (d)pns, $v.c$ — array of the node's children with (d)pns, $v.parent$ — refer to corresponding parent's v.

TRYRUNJOB is shown in Algorithm 4. It works as DFPN, but additionally calculates virtual (d)pns and stores them in list v. Once it finds a candidate — the condition in line 5 is true — virtual (d)pns are propagated upwards to the root and an actual job is run. Once this job is done the entire recursion ends, i.e. no more search is performed and virtual (d)pns are updated by VTTREMOVE calls.

Algorithm 3. Parallel DFPN

```
1: procedure PARALLELDFPN(root)
2:     for i = 1, ..., # of threads do
3:         spawn thread with call RUN(root)
4: procedure RUN(n)
5:     while n is not solved do
6:         LOCK(job_lock)
7:         v.n ← n, v.parent ← null
8:         job_done ← false
9:         TRYRUNJOB(n, v, ∞, ∞)
10:        UNLOCK(job_lock)
11:        if job_done then
12:            notify waiting threads
13:        else
14:            wait
```

5.5 Comparison to Kaneko's algorithm

Kaneko parallelizes DFPN like this [9]: in an OR node's tt access, a child's pn is increased by the number of threads searching that child. Kaneko calls this augmented value a virtual pn[3], discouraging — but not preventing — the search from repeatedly selecting the same child. In our experiments we observe that, near tree-top, sibling pns vary, whereas near tree-bottom, they are all small and so similar. In the former case, Kaneko's algorithm is likely to always select the same child for search.

By contrast, in our approach, by setting the value of *MaxWorkPerJob*, we implicitly control how deep in the tree the diversion of thread selections should occur. Moreover, our threads always work on different subtrees.

6 Experiments

We implemented SPDFPN for Hex on the open-source Hex repository Benzene [2], which in turn is built on the open-source game-independent framework Fuego [5]. Benzene uses Focussed DFPN search [1, 7], which employs an evaluation function to sort a node's children, and then focusses the search on a fraction of the most-promising children. The size of the search window is given by $\lceil b + f \times \#\text{active children} \rceil$, so new children can enter the window as siblings are proved to be a loss. We use $b = 0$ and $f = 0.25$. FDFPN search maintains the usual correctness properties of PN search. We used FDFPN because it is embedded in Benzene's DFPN; our use of this DFPN variant does not diminish the generality of SPDFPN.

Before starting our experiments, we improved Benzene's virtual connection engine and solver. The resulting implementation performs typically 2 to 10 times faster than the previous version on similar hardware [8].

We tested SPDFPN on two sets of Hex problems: Suite 1, the thirteen previously intractable 9×9 opening moves plus the (previously intractable) centremost 10×10 opening move; and Suite 2, the eight hardest 8×8 opening moves plus eight positions from the 2011 Olympiad Hex competition [6].

6.1 Previously intractable 9×9 and 10×10 Hex openings

Suite 1 tests the limits of SPDFPN on 24 threads of a hyperthreaded 12-core Intel Xeon 2.93 GHz with 48 Gbyte RAM and 8 threads of an 8-core Intel Xeon 2.8 GHz with

[3] Do not confuse Kaneko's augmented proof/disproof values with our definition of virtual pn.

Algorithm 4. Try find a candidate and run a job

```
1: function TRYRUNJOB(n, v, P, D)
2:   if v.n.p ≥ P ∨ v.n.d ≥ D then
3:     return 0
4:   w_local ← 0
5:   if n.w < MaxWorkPerJob then
6:     if n.p ≤ n.d then
                                   ▷ Virtual win
7:       v.n.p ← 0, v.n.d ← ∞
8:     else
                                   ▷ Virtual loss
9:       v.n.p ← ∞, v.n.d ← 0
10:    UPDATEVIRTUALS(v)
11:    UNLOCK(job_lock)
                                   ▷ Candidate is found
                                   ▷ Actual job is run here
12:    w_local ←
13:      DFPN(n, P, D, MaxWorkPerJob)
14:    LOCK(job_lock)
15:    job_done ← true
16:    v.n.p ← n.p, v.n.d ← n.d
17:    VTTREMOVE(v.n)
18:    return w_local
19:  for each child j of n do
20:    c[j] ← TTREAD(n, j)
21:    v.c[j] ← VTTREAD(n, j, c[j])
22:  loop
23:    PNUPDATE(n, c)
24:    PNUPDATE(v.n, v.c)
25:    TTWRITE(n)
26:    if job_done then
27:      VTTREMOVE(v.n)
28:      return w_local
29:    if v.n.p ≥ P ∨ v.n.d ≥ D then
30:      return w_local
31:    j, P_j, D_j ←
32:      SELECT(v.n, v.c, P, D)
33:    v_child.n ← v.c[j], v_child.parent ← v
34:    w_child ←
35:      TRYRUNJOB(c[j], v_child, P_j, D_j)
36:    w_local ← w_local + w_child
37:    n.w ← n.w + w_child
38: procedure UPDATEVIRTUALS(v)
39:   VTTADD(v.n)
40:   while v.parent is not null do
41:     v ← v.parent
42:     PNUPDATE(v.n, v.c)
43:     VTTADD(v.n)
```

32 Gbyte RAM. We used a tt with sizes varying from 2^{27} to 2^{28} entries depending on machine and stage of a search. Here, in TRYRUNJOB, in addition to a tt we used a database storage capable of handling simple board isomorphism (180 degree rotation). For more difficult openings we gradually raised the value of ε from 0.25 up to 0.5 in order to reduce the number of tt lookup failures.

Table 1 shows machine used and approximate running times. Due to occasional machine shutdown, e.g. power failure, some runs were restarted several times from database and tt backups; for these runs the running times are cumulative estimates based on logs. As an indication of achieved speedup on this problem suite, the previous algorithm with 8 threads failed to solve any of these openings after 480 hours[4], whereas SPDFPN with 24 threads solves `c2` in under 33 hours. On the 9×9 board `a6` was the hardest opening. The behaviour of SPDFPN on this problem was different than on all others: the main lines of play were extremely balanced, and the winner unclear, until deep into the search. See Figure 2. Although the search space was around 100,000 times larger than the size of tt, SPDFPN showed continuous progress. In the previously strongest Hex solver the search often gets stuck whenever search space is this much larger than the tt [2, 1].

[4] Private communication with Broderick Arneson.

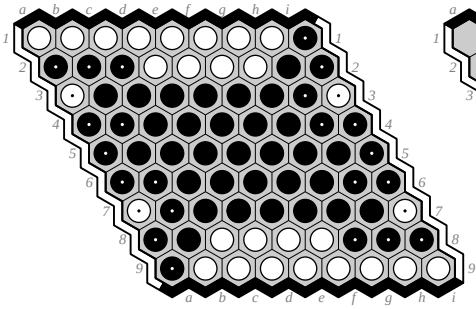

Fig. 1. Newly solved 9×9 opening values (dots), winner if black opens there.

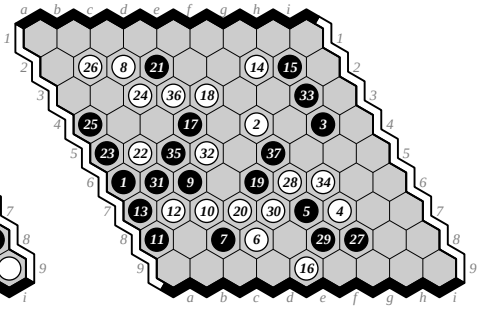

Fig. 2. PV of a6, the hardest 9×9 opening.

6.2 8×8 and Olympiad Hex problems

Suite 2 measures the parallel efficiency of our algorithm. On a 24-core Intel Xeon 2.4 GHz with 64 Gbyte RAM, we used 16 threads (the others were in use). We used a tt with 2^{24} entries, which was more than sufficient. We picked moderate problems: challenging but still tractable for a single thread. This suite consists of eight (hardest) 8×8 openings and eight 11×11 positions from the 2011 ICGA Olympiad found by starting with the final position and proceeding backwards to a moderate position. See Figures 4 and 5.

6.3 Scalability

In this experiment we measured scalability, or parallel efficiency, — the average[5] speedup ratio over serial version — for positions from suite 2. We ran our algorithm over all instances two times. For run 1, on board size 8×8 (11×11), the default value of *MaxWorkPerJob* was 100 (20). For run 2, *MaxWorkPerJob* was 500 (100). In many domains, small values such as these can yield lower scalability due to thread management

[5] As usual when measuring a ratio (here, speedup), we use geometric mean for averaging.

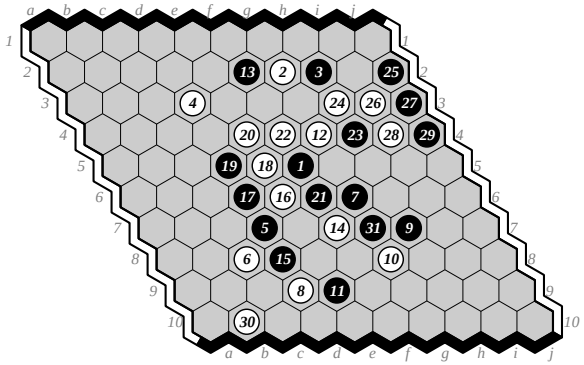

Fig. 3. Principle variation of f5, the first-ever solved 10×10 opening. Black wins.

opening	#threads	time	winner
a2	8/24	68d09:40:18	black
a3	8	80d08:37:34	white
a4	8	33d14:06:03	black
a5	8	65d04:14:52	black
a6	24	110d14:35:06	black
a7	24	4d08:56:03	white
a8	24	6d14:21:30	black

opening	#threads	time	winner
b2	8	53d15:18:21	black
b4	8	29d23:53:14	black
b6	8	1d21:52:28	black
b7	8	4d17:19:13	black
c2	24	1d08:42:57	black
i1	24	6d00:51:25	black
10x10:f5	24	63d20:44:30	black

Table 1. Times (days:hrs:mins:secs) and threads for newly solved 9×9 and 10×10 openings.

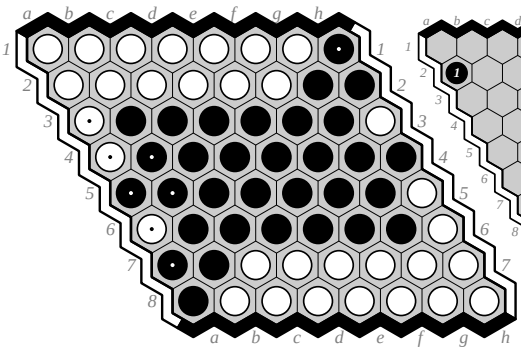

Fig. 4. The hardest 8×8 openings (dots), winner if black opens there.

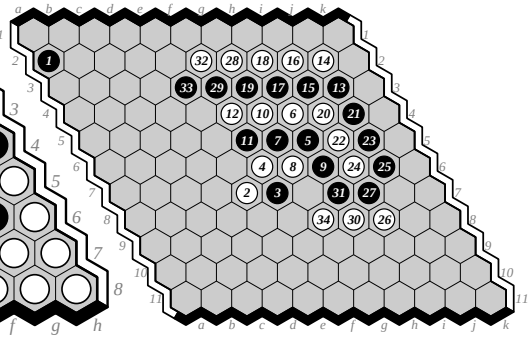

Fig. 5. The hardest suite 2 position, from game 7 of the 2011 Olympiad.

overhead. However, our Hex solver spends a large fraction of the time on VC engine computations, so for this solver these small values of *MaxWorkPerJob* are suitable.

Our algorithm scales well on up to 16 threads. Figure 6 shows the scalability from run 1 (9.4 with 16 threads, or .59), and from the six hardest problems — three each from 8×8 and 11×11 — from run 2 (e.g. 11.8 with 16 threads, or .74). Table 2 shows how time is lost due to parallelization. For each run $i = 1, 2$, f_t^i denotes the average fraction of time lost due to multi-threading overhead (shared tt access, hardware overhead) and f_s^i denotes the average fraction of time lost due to extra leaf expands (extra states searched).

These values are computed as follows. Let s_1 and s_n be the number of leaf expands (number of states for which the VC engine was used) for the serial and n-thread runs respectively. Then $f_s^i = s_n/s_1$. Let t_1 and t_n be the actual running times for the serial and n-thread runs. If there is no multi-threading overhead then the expected time of the n threads run is $E_t = t_1 s_n/s_1/n$, so $f_t^i = t_n/E_t$. Thus $t_n = f_t f_s t_1/n$, so the scalability e_i is $n/(f_t^i f_s^i)$.

7 Conclusions

We have introduced SPDFPN, a parallel version of depth-first proof number search that scales well. We tested our algorithm on two suites of Hex problems, in the process

	run 1			run 2		
n	f_t^1	f_s^1	e_1	f_t^2	f_s^2	e_2
1	1.000	1.000	1.000	1.000	1.000	1.000
2	0.981	1.051	1.940	1.033	1.095	1.768
4	1.071	1.094	3.414	1.041	1.158	3.318
8	1.111	1.124	6.405	1.072	1.268	5.885
12	1.098	1.398	7.816	1.008	1.319	9.028
16	1.219	1.401	9.368	1.091	1.245	11.780

Table 2. Scalability.

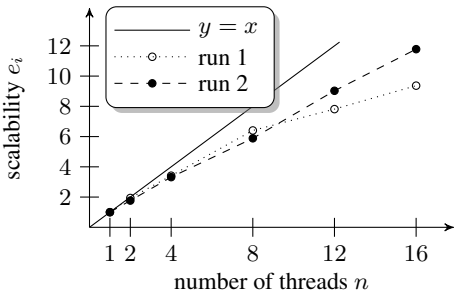

Fig. 6. Scalability.

solving all thirteen previously intractable 9×9 openings and the first-ever solution to a 10×10 opening. Our experiments showed a speedup of .74, namely 11.8 on 16 threads. Our algorithm is general and game-independent, and so should be equally effective on any problem that can be modelled by and-or trees. It would be of interest to see whether the SPDFPN speedups we achieved in Hex can be achieved in other domains, and to compare and contrast SPDFPN to Kaneko's parallel DFPN.

8 Acknowledgements

We thank Broderick Arneson, Yngvi Björnsson, Phil Henderson, Aja Huang, Timo Ewalds, Martin Müller, and the referees for their feedback. We thank Martin for generously loaning the use of his computing cluster for our experiments.

References

1. Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Solving Hex: Beyond humans. In H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat, editors, *Computers and Games 2010*, volume 6515 of *LNCS*, pages 1–10. Springer, 2011.
2. Broderick Arneson, Philip Henderson, and Ryan B. Hayward. Benzene, 2009-2012. `http://benzene.sourceforge.net/`.
3. D.M. Breuker, J.W.H.M. Uiterwijk, and H.J.van den Herik. Replacement schemes and two-level tables. *ICGA*, 19(3):175–180, 1996.
4. Guillaume Chaslot, Mark H.M. Winands, and H. Jaap van den Herik. Parallel monte-carlo tree search. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H.M. Winands, editors, *Computers and Games 2008*, volume 5131 of *LNCS*, pages 60–71. Springer, 2008.
5. Markus Enzenberger, Martin Müller, Broderick Arneson, Rich Segal, Fan Xie, and Aja Huang. Fuego, 2007-2012. `http://fuego.sourceforge.net/`.
6. Ryan B. Hayward. 2011 ICGA Computer Games Olympiad Hex Competition Report, 2011. `http://webdocs.cs.ualberta.ca/~hayward/papers/rptTilburg.pdf`.
7. Philip Henderson. *Playing and solving Hex*. PhD thesis, University of Alberta, 2010. `http://webdocs.cs.ualberta.ca/~hayward/theses/ph.pdf`.
8. Philip Henderson, Broderick Arneson, and Ryan Hayward. Solving 8×8 Hex. In *Proc. IJCAI-09*, pages 505–510, 2009.

9. Tomayuki Kaneko. Parallel depth first proof number search. In *Proc. AAAI-10*, pages 95–100, 2010.
10. A. Kishimoto, M.H.M. Winands, M. Müller, and J-T. Saito. Game-tree search using proof numbers: The first twenty years. *ICGA*, 35(3):131–156, 2012.
11. Akihiro Kishimoto. Parallel AND/OR tree search based on proof and disproof numbers. In *5th Games Programming Workshop*, volume 99 of *IPSJ Symposium Series*, pages 24–30, 1999.
12. Akihiro Kishimoto. Dealing with infinite loops, underestimation, and overestimation of depth-first proof-number search. In *Proc. AAAI-10*, pages 108–113, 2010.
13. Akihiro Kishimoto and Martin Müller. A solution to the ghi problem for depth-first proof-number search. *Information Sciences*, 175(4):296–314, 2005.
14. Akihiro Kishimoto and Martin Müller. About the completeness of depth-first proof-number search. In H. Jaap van den Herik, X. Xu, Z. Ma, and Mark H.M. Winands, editors, *Computers and Games*, volume 5131 of *LNCS*, pages 146–156. Springer, 2008.
15. Fabien Letouzey. Fruit, 2004-2013. `http://www.fruitchess.com/`.
16. Ayumu Nagai. A new AND/OR tree search algorithm using proof number and disproof number. In *Proceeding of Complex Games Lab Workshop*, pages 40–45, Tsukuba, November 1998. ETL.
17. Ayumu Nagai. A new depth-first-search algorithm for and/or tree. Master's thesis, University of Tokyo, Japan, 1999.
18. Ayumu Nagai. *Df–pn Algorithm for Searching AND/OR Trees and its Applications*. PhD thesis, University of Tokyo, Japan, 2002.
19. Jakub Pawlewicz and Lukasz Lew. Improving depth-first pn-search: $1+\varepsilon$ trick. In H.J̃aap van den Herik, P. Ciancarini, and H.H.L.M(J.) Donkers, editors, *Computers and Games 2006*, volume 4630 of *LNCS*, pages 160–170. Springer, 2007.
20. Tord Romstad. Stockfish, 2008-2013. `http://stockfishchess.org/`.
21. Abdallah Saffidine, Nicolas Jouandeau, and Tristan Cazenave. Solving breakthrough with race patterns and job-level proof number search. In H. Jaap van den Herik and Aske Plaat, editors, *Advances in Computers and Games 2011*, volume 7168 of *LNCS*, pages 196–207. Springer, 2012.
22. Jahn-Takeshi Saito, Mark H.M. Winands, and H. Jaap van den Herik. Randomized parallel proof-number search. In H. Jaap van den Herik and Pieter Spronck, editors, *Advances in Computer Games*, volume 6048 of *Lecture Notes in Computer Science*, pages 75–87. Springer Berlin Heidelberg, 2010.
23. Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317:1518–1522, 2007.
24. I-Chen Wu, Hung-Hsuan Lin, Ping-Hung Lin, Der-Johng Sun, Yi-Chih Chan, and Bo-Ting Chen. Job-level proof-number search for connect6. In H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat, editors, *Computers and Games 2010*, volume 6515 of *LNCS*, pages 11–22. Springer, 2011.
25. Kazuki Yoshizoe, Akihiro Kishimoto, and Martin Müller. Lambda depth-first proof number search and its appplication to go. In *Proc. IJCAI-07*, pages 2404–2409, 2007.