Abstract—Nested Rollout Policy Adaptation (NRPA) is a Monte Carlo tree search algorithm that excels in the domain of single agent optimization problems. It was applied to a wide class of problems, including vehicle routing, DNA alignment, 3D packing, travelling salesman problem, combinatorial puzzles and more.

We develop a parallel version of NRPA that replicates results of the sequential version. The parallelization allows us to run deeper calculations. The experimental data shows that depth of the calculation is a deciding factor in the result quality. Earlier parallelization attempts used parallel architecture to run wider, but not deeper, calculations.

We applied the parallel version to the Morpion Solitaire benchmark. To aid parallelization, we used a different best result replacement rule. Using the new rule, on a distributed architecture with 768 cores we obtained an average speedup over a single core computation by a factor 547.48 (measured by a wall-clock time). The different replacement rule was quite effective. In each run, the record 178-move sequence was found.

II. OPTIMIZATION PROBLEM ABSTRACTION

We assume that the optimization problem supports the following three operations.

SIMULATE(P) - given a policy P, generate a random playout;

ADAPT(P, B) - given a policy P and a playout B, adapt P to increase the probability of generating B;

MAX(B₁, B₂) - select a playout with higher reward.

We assume that the following conditions are satisfied.

1) ADAPT converges quickly: an ADAPT with the same playout B repeated few dozen times results in a policy that generates B with probability close to 1.

2) ADAPT is fast: we can compute hundreds of adapts per second.

3) SIMULATE may be much slower; in fact, if it is fast, we replace it with a short sequential NRPA run (that we call an atomic computation) to slow it down.

4) Reward is discrete and does not change often.

Note that condition 4) is a characteristic of more challenging optimization problems, such as Montezuma Revenge. For parallel NRPA, sparsity of rewards aids parallelization, as each improvement of the best sequence usually discards part of the calculations.

In the present paper we used Morpion Solitaire as a benchmark problem, as it was used by Rosin in the original NRPA paper [18]. The benchmark is described in Section V.

III. ROLLOUT PARALLELIZATION

A rollout is a reinforcement learning loop displayed as Algorithm 1. A rollout is a basic building block of the NRPA algorithm.
Algorithm 1 A rollout - basic building block of NRPA

1: function ROLLOUT
2: \( p_1 \leftarrow I \) \> Initial policy (uniform random)
3: \( b_0 \leftarrow \emptyset \) \> Best result (so far)
4: for \( i \leftarrow 1 \) to \( N \) do
5: \( r_i \leftarrow \text{SIMULATE}(p_i) \)
6: \( b_i \leftarrow \max(b_{i-1}, r_i) \)
7: \( p_{i+1} \leftarrow \text{ADAPT}(p_i, b_i) \)
8: return \( b_N \)

To show how to parallelize ROLLOUT, consider a sample run shown below.

The diagram shows three iterations of the rollout loop. In the first iteration, SIMULATE on policy \( p_1 \) returns a result with value 167, which is stored in \( r_1 \) and \( b_1 \) (in the diagram we identify a sequence with its reward, for simplicity, so number 167 stands for sequence \( r_1 \)). Then \( b_1 \) is used to ADAPT \( p_2 \). In the second iteration, SIMULATE returns a result with value 155, which is stored in \( r_2 \), but not in \( b_2 \), which stays equal to \( b_1 \). Then \( b_2 = b_1 \) is used to ADAPT \( p_3 \). In the third iteration, SIMULATE yields a result with value 173, which is stored as \( b_3 \) and \( b_3 \) is returned as the final value of the rollout.

We can observe that the calculation can be made faster if we compute simulations from \( p_2 \) and \( p_3 \) in parallel (after adapting \( p_3 \) from \( p_1 \) using sequence \( r_1 \) twice). This is possible because \( r_2 \) does not improve \( b_1 \). The following diagram shows a computation of the same rollout in two iterations, with two computations running in parallel.

An algorithm that implements this idea is displayed as Algorithm 2. Note that we do not know beforehand which calculations can be made in parallel, so some calculations are started and then have to be discarded, when we find out that an earlier simulation improved the best sequence.

IV. Nested rollout parallelization

The NRPA algorithm uses nested rollouts. At level \( L > 0 \), NRPA replaces a call to SIMULATE with a recursive call to NRPA at level \( L - 1 \). The lower level NRPA calculation uses its parent’s policy, instead of a uniform random one. The NRPA algorithm is displayed as Algorithm 3. Note that \( L = 1 \) NRPA is equivalent to the ROLLOUT function described in the last section.

Figure 1a shows a sample run of level 2 NRPA (sequential version that is displayed as Algorithm 3). Note that as before, this process can be parallelized at the lowest level \( L = 1 \) of simple rollouts, using strategy described in Section III. But we can also parallelize the computation at the higher level \( L = 2 \). The idea is shown in Figure 1b. Note that we benefit greatly from \( L = 2 \) parallelization: a sequential version of the algorithm runs in 9 iterations; if we only parallelized at \( L = 1 \), it would run in 8 time steps; the fully parallel version finishes in 6 iterations. The full algorithm is displayed as in the Appendix Algorithm 4.
(a) A sample run of \( L = 2 \), \( N = 3 \) sequential NRPA (Algorithm 3). Bottom row shows \( L = 2 \) rollout. At iteration \( i = 1 \) we start with uniform random policy \( p_1 \), which is passed to a recursive NRPA call at level \( L = 1 \). At \( L = 1 \) a rollout is performed (top row, left side) and a result with value 81 is passed back to level 2 rollout. Policy \( p_4 \) is adapted from \( p_1 \) using result with value 81. Policy \( p_4 \) is passed to another recursive NRPA call at \( L = 1 \); it yields a result with value 85. It is used to adapt policy \( p_7 \) from policy \( p_4 \). The final recursive call yields result with value 98.

(b) A parallelization of a sample run of \( L = 2 \) of NRPA discussed above (Algorithm 4). Policy \( p_4 \) can be adapted straight from \( p_2 \), before first \( L = 1 \) run is completed, as sequence simulated by \( p_3 \) does not improve the best result. Likewise, simulations from policies \( p_5 \), \( p_6 \) and \( p_7 \) can be made in parallel.

Fig. 1. The idea of parallelization of the NRPA algorithm.

Algorithm 3 NRPA algorithm with \( N \) iterations and \( L \) levels, sequential version from Rosin’s paper [18]. The parallel version is displayed as Algorithm 4.

1: procedure NRPA\((L, p)\)  
2: if \( L = 0 \) then  
3: return SIMULATE\((p)\)  
4: else  
5: \( p_1 \leftarrow p \) \hspace{1em} \( \triangleright \) Initial policy (passed by parent)  
6: \( b_0 \leftarrow \emptyset \) \hspace{1em} \( \triangleright \) Best result (so far)  
7: for \( i \leftarrow 1 \) to \( N \) do  
8: \( r_i \leftarrow \text{NRPA}(L - 1, p_i) \)  
9: \( b_i \leftarrow \max(b_{i-1}, r_i) \)  
10: \( p_{i+1} \leftarrow \text{ADAPT}(p_i, b_i) \)  
11: return \( b_N \)

V. THE EXPERIMENTS - MORPION SOLITAIRE

A. Morpion Solitaire

The Morpion Solitaire is a paper-and-pencil single-player game played on a square grid with the initial configuration of 36 dots depicted in Figure 2. In each move the player puts a dot on an unused grid position and draws a line that consists of four consecutive segments passing through the dot. The line must be horizontal, vertical or diagonal and no segment may be drawn twice, i.e. the moves have to be segment-disjoint. The goal is to find the longest possible sequence of moves. It was proved in [6] that the problem is finite. The best known upper bound on the length of the sequence is 485 [15] and is far away from the best known sequence of length 178.

The problem of finding the longest sequence of moves in the Morpion Solitaire is notoriously hard for computers.
For 34 years the longest known sequence was one of 170 moves discovered by hand by Bruneau in 1976. Despite considerable computational effort, until 2010 the computer generated sequences were much shorter. In 2010 the human record was broken, when Rosin [18] obtained the current world record of 178 moves using the NRPA algorithm. The webpage [4] maintained by Boyer contains an extensive and up-to-date information about records in all Morpion Solitaire variants.

B. Policy, adapt and max functions

We used a weight-table based policy and gradient-ascent adapt, faithfully following Rosin’s paper [18].

The standard NRPA algorithm always replaces the best sequence if the new one that was found has greater or equal length. For greater parallel efficiency, we want to make as little replacements as possible. We found that sequences of equal length often differ by just few moves and replacement by a similar sequence does not aid exploration, while it ruins parallelization. Therefore we ran our experiments with different replacement strategy: we replaced a sequence with a sequence of equal length only if the new sequence had more than 30% different moves than the old one. Note that during atomic computations (a short run of sequential NRPA done at the worker nodes) we used the usual replacement rule. Hence our rule was used on top two or three levels of the computation.

C. Order of computation

We used the following heuristic as the SELECT function in Algorithm 4. We guesstimated a probability that a simulation will beat a sequence of the given length (a probability table was used; probability that a sequence of length 130 will be beaten was set to 10%; to 20% for length 140 and so on). Then for each leaf node SELECT function calculated probability that the search will not change its policy and a node with greatest probability was selected for computation.

D. Atomic calculations

The algorithm was ran on a distributed architecture using Intel’s MPI library (2018 version) for inter-process communication. The server code was implemented in Python with C++ extensions and worker code mostly in C++. Because of communication overhead, workers instead of calculating a single SIMULATE call calculated a $L = 2$ or $L = 3$ NRPA run (an atomic calculation). For $L = 6$, $N = 80$ runs a single worker task was $L = 3$, $N = 80$ NRPA call. In these runs the server had to distribute 80$^3 = 512000$ atomic calculations among it workers (this number does not include restarted calculations). A single worker calculated at an approximate speed of 20000 simulations per second. An atomic calculation with $L = 3$, $N = 80$ took approximately 25 seconds.

E. Implementation

The full implementation is published in a repository

https://github.com/amn/parallel-nrpa/

F. Experimental results

We ran 10 calculations with $N = 80, L = 6$. Every calculation found the record sequence of length 178 (it was always equal up to symmetry to the record sequence found by Rosin, see Figure 4). The calculations were ran on 768 cores (with three levels of atomic computations) and the average parallel speedup was 547.48 (71% efficiency).

We ran 20 calculations with $N = 70, L = 6$. The record sequence of 178 moves was found 10 times, a sequence with 177 moves was found 9 times and a sequence of length 172
Parallel speedup over iteration. Efficiency increases when the best sequence stabilizes.

Best sequence over iteration. Sequence of length 178 was found near the end of 20-hour run.

Fig. 3. Statistics from a $L = 6, N = 80$ NRPA run.

Fig. 4. A 178 moves sequence found by $L = 6, N = 80$ run, identical (up to symmetry) to Rosin’s record sequence.

VI. FURTHER WORK

The parallelization of NRPA is achieved at the top-level of the algorithm. We do not modify the policy and treat it as a black-box component (as described Section II). The algorithm developed in this paper and the speedup obtained by the parallelization allows for a replacement of a computationally inexpensive weight-table based policy (used by the original NRPA) by a much more computationally expensive neural networks. Such an approach was previously successfully applied to MCTS algorithm in the Alpha Go Zero project [2] to create world’s best Go player. In a similar fashion, parallelized NRPA with neural networks can be applied to a variety of single-agent problems, such as challenging instances of Atari games, such as Sokoban [17].

Sources

The full source code for experiments reported in the paper is published in a repository

https://github.com/amn/parallel-nrpa/

ACKNOWLEDGMENT

This research would not be possible without a support from the PL-Grid Infrastructure. In particular, we used extensively the Prometheus supercomputer, located in the Academic Computer Center Cyfronet in the AGH University of Science and Technology in Kraków, Poland (4320 Intel Xeon E5-2680 v3 processors, each with 12 cores).

REFERENCES


Algorithm 4 Parallelization of NRPA with $N$ iterations, $L$ levels and $K$ workers

1: procedure SERVER
2: $p_1 \leftarrow 1$  
   \hspace{1em} \triangleright \text{Root node policy}
3: $b_1 \leftarrow \emptyset$
4: pending $\leftarrow \text{LEAVES(\text{TREE($L$ levels, degree $N$)})}$
5: workers $\leftarrow \{1, 2, \ldots, K\}$, finished $\leftarrow \emptyset$
6: \hspace{1em} while $\# \text{finished} < N^L$ do
7: \hspace{2em} while $\# \text{workers} > 0$ and $\# \text{pending} > 0$ do
8: \hspace{3em} $i \leftarrow \text{SELECT(pending)}$, pending $\leftarrow \text{pending} \setminus \{i\}$
9: \hspace{3em} $w \leftarrow \text{min(workers)}$, workers $\leftarrow \text{workers} \setminus \{w\}$
10: \hspace{2em} SELECTPOLICY($i$)
11: \hspace{2em} job$_i$ = UNIQUEID()
12: \hspace{1em} SEND(target=$w$, (i=$i$, id=job$_i$, policy=$p_i$, seed=seed$_i$))
13: \hspace{2em} (i, w, id, result) $\leftarrow \text{RECEIVE()}$  
   \hspace{1em} \triangleright \text{Blocking call}
14: \hspace{2em} workers $\leftarrow \text{workers} \cup \{w\}$
15: \hspace{2em} if job$_i$ = id then  
   \hspace{3em} $r_i \leftarrow \text{result}$  
   \hspace{3em} \triangleright \text{Job was not discarded}
16: \hspace{2em} finished $\leftarrow \text{finished} \cup \{i\}$
17: \hspace{2em} while RESULT($i$) $>$ b$_i$ do
18: \hspace{3em} b$_i$ $\leftarrow \text{RESULT($i$)}$
19: \hspace{3em} for each sibling $j > k$ do DISCARD($j$)
20: \hspace{2em} if $i = 1$ then break  
   \hspace{3em} \triangleright \text{Root node}
21: \hspace{2em} else $i = \text{PARENT($i$)}$
22: \hspace{1em} return b$_1$
23: \hspace{1em} function SELECTPOLICY($i$)
24: \hspace{2em} if $p_i$ is not set then
25: \hspace{3em} if $i$ is first in node then
26: \hspace{4em} $j \leftarrow \text{PARENT($i$)}$
27: \hspace{4em} SETPOLICY($j$)
28: \hspace{4em} b$_i$ $\leftarrow \emptyset$
29: \hspace{4em} $p_i \leftarrow p_j$  
   \hspace{3em} \triangleright \text{Copy parent’s policy}
30: \hspace{3em} else
31: \hspace{4em} SETPOLICY($i - 1$)
32: \hspace{4em} b$_i$ $\leftarrow b_{i-1}$
33: \hspace{4em} $p_i \leftarrow \text{ADAPT($p_j, b_{i-1}$)}$  
   \hspace{3em} \triangleright \text{Adapt sibling’s policy}
34: \hspace{2em} else
35: \hspace{3em} function DISCARD($j$)
36: \hspace{4em} delete $p_j, b_j$
37: \hspace{4em} if $j$ is leaf then
38: \hspace{5em} delete $r_j, \text{job}_j$
39: \hspace{5em} finished $\leftarrow \text{finished} \setminus \{j\}$
40: \hspace{5em} pending $\leftarrow \text{pending} \cup \{j\}$
41: \hspace{4em} else
42: \hspace{5em} for each child $k$ do DISCARD($k$)
43: \hspace{2em} function RESULT($i$)
44: \hspace{3em} return $\max\{r_j : j \text{ is consistent leaf below } i\}$