Distributed Nested Rollout Policy for Same Game

Benjamin Negrevergne[1] and Tristan Cazenave[1]

PSL Université Paris-Dauphine, LAMSADE UMR CNRS 7243, Place du Maréchal de Lattre de Tassigny, 75775 Paris Cedex 16, France

Abstract. *Nested Rollout Policy Adaptation* (NRPA) is a Monte Carlo search heuristic for puzzles and other optimisation problems. It achieves state of the art performance on several games including *SameGame*. In this paper, we design several parallel and distributed NRPA-based search techniques, and we provide number of experimental insights about their execution. Finally, we use our best implementation to discover 15 better playouts for 20 standard SameGame boards.

1 Introduction

SameGame is a popular puzzle game whose goal is to clear a rectangular area filled with coloured blocks (See Figure 1). When the player clears a block, all the consecutive blocks with the same color are also cleared, and the score is increased by the square of the number of blocks cleared minus two, creating an incentive for the player to clear larger coloured areas. After clearing one or more blocks, gaps are filled by the effect of gravity, creating new arrangements of coloured blocks. Finally, a bonus of one thousand is given for clearing the entire area.

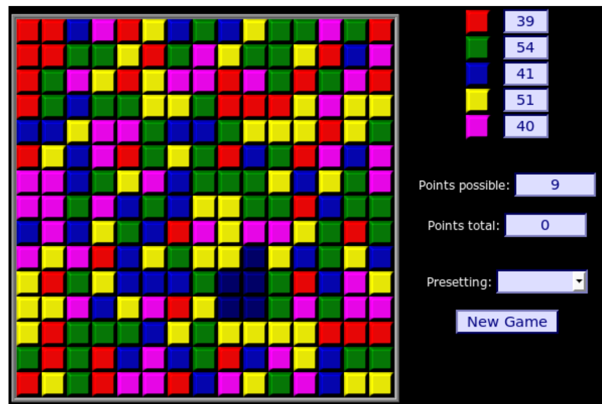

Fig. 1. Example of the initial state of a SameGame problem

The number of possible moves and the computational complexity of SameGame (discussed in [1]) has made it a challenging candidate problem for solving techniques. State of the art solving techniques for SameGame use a Monte Carlo search strategy to learn a successful playout policy. For example, Nested Monte Carlo Search (NMCS) [3] biases its playouts using lower level playouts. At the lowest level, NMCS adopts a uniform random playout policy.

Combining NMCS with online learning of playout strategies has been proposed and has given good results on many optimisation problems [13]. Online learning of a playout policy in the context of nested searches has been further developed for puzzles and optimisation with Nested Rollout Policy Adaptation (NRPA) [14]. NRPA has discovered new world records in Morpion, Solitaire and crosswords puzzles. Furthermore, Stefan Edelkamp and co-workers have applied the NRPA algorithm to multiple problems including the Traveling Salesman with Time Windows (TSPTW) problem [5, 7]. Other applications deal with 3D Packing with Object Orientation [9], the physical travelling salesman problem [10], the Multiple Sequence Alignment problem [11] or Logistics [8].

The principle of NRPA is to adapt the playout policy so as to learn the best sequence of moves found so far at each level. As with most Monte Carlo search algorithms, NRPA is computationally intensive, and the quality of the solution is closely tied with the time available to run the algorithm: the longer it runs, the better the solution. Furthermore, discovering better solutions becomes increasingly challenging as it gets closer to the optimal solution. To cope with this problem, we design several parallelization strategies for NRPA. We then run thorough experiments on SameGame to understand the performances of NRPA in a distributed context. The main insight in this paper is that one should use hybrid parallelization strategies to balance exploration of the search space with exploitation of the intermediary results. Finally, we used our distributed implementation to discover better solutions for several standard SameGame boards. Using only two hours of computation, our algorithm is able to discover better solutions for 15 out of 20 problems (better or equal on 17 out of 20).

2 Nested Rollout Policy Adaptation (NRPA)

NRPA currently holds world records for several puzzle games including Morpion, Solitaire and crossword puzzles. The Playout Policy Adaptation algorithms, which is closely related to NRPA, was also used to improve MCTS-based Go programs [12] and a number of other game playing programs, resulting in a number of great successes [4]. To achieve these results, NRPA efficiently combines multiple levels of nested searches with online policy learning [14].

In NRPA, a policy is a set of weights, one weight for each possible move in the game. The policy is initialised with random weights and is then used by the playout algorithm as a bias to search for a solution: moves with higher weights are more likely to be sampled. The search is repeated N times and after each iteration, the sequence of moves that has led to the best score is then used to update the policy. To update the policy we increase the weights of the moves

occurring in the best sequence and decrease the weights of the others legal moves. To further improve the quality of the policy, this procedure is nested multiple times as described in Algorithm 1.

The playout algorithm is given in Algorithm 2, it performs Gibbs sampling to choose a legal move with a probability proportional to the exponential of its weight. Finally, the policy adaptation algorithm is given in Algorithm 3. For each move in the sequence. The Adapt() function increases the weight of the corresponding move by α, and decreases the weights of the other possible moves by a value proportional to the exponential of their weight. Empirical evaluation has shown that 1.0 is a good value for α.

Algorithm 1 The NRPA algorithm.

```
1: NRPA (level, policy)                    /* All variables are passed by value */
2: if level == 0 then
3:    (score, sequence) ← playout (initial-state, policy)
4:    return (score, sequence)
5: else
6:    best-score ← −∞
7:    best-sequence ← [ ]              /* The best sequence of moves found so far */
8:    for N iterations do
9:       (score, sequence) ← NRPA(level − 1, policy)
10:      if score ≥ best-score then
11:         best-score ← score
12:         best-sequence ← sequence
13:      end if
14:      policy ← Adapt(policy, best-sequence, 1.0)
15:   end for
16:   return (best-score, best-sequence)
17: end if
```

Algorithm 2 The playout algorithm

```
1: playout (state, policy)
2: sequence ←[ ]
3: while state ≠ terminal-state do
4:    z ← 0.0
5:    for m in legal-moves(state) /* All moves that are legal from state */ do
6:       z ← z + exp(policy[ code(m) ])  /* code() converts m to an integer repr. */
7:    end for
8:    choose move with probability exp(policy[code(move)]) / z
9:    state ← play(state, move)
10:   sequence ← sequence + move
11: end while
12: return (score(state), sequence)
```

Algorithm 3 The Adapt algorithm

```
1:Adapt (policy, sequence, α)
2: new-policy ← policy   /* This copy is optional since arguments passed by value*/
3: state ← initial-state
4: for move in sequence do
5:    new-policy[ code(move) ] ← new-policy[ code(move) ] + α
6:    z ← 0.0
7:    for m in legal-moves(state) do
8:       z ← z + exp(policy[ code(m) ])
9:    end for
10:   for m in legal-moves(state) do
11:      new-policy[ code(m) ] ← new-policy[ code(m) ] - α * exp(policy[ code(m) ])/z
12:   end for
13:   state ← play(state, move)
14: end for
15: return new-policy
```

2.1 Solving SameGame with NRPA

To solve SameGame with NRPA we have to find an adequate representation for the board state and the moves in order to specify $score(state)$ and $play(state, move)$ according to rules of the game described in the introduction.

The game state is easily represented with a 2D array of integers with as many lines and columns as the board itself. Integers are then used to code the colour of the cell, or empty cells. Representing the moves is more challenging. There are so many possible moves in SameGame that it is not possible to code them with a simple function without exceeding storage capacities. Naive hashing techniques quickly lead to hash collisions.

We deal with this problem by using Zobrist hashing [15], which is popular in computer games such as Go and Chess [2]. It uses a 64 bits random integer for each possible colour of each cell of the board. The code for a move is the XOR of the random numbers associated to the cells of the move. A transposition table is used to store the codes and their associated weights. The index of a move in the transposition table is its 16 lower bits. For each entry of the transposition table, a list of move codes and weights is stored. Note that this is not the only way to represent moves. Alternative representations that include the surrounding of the blocks removed, or that merge similar moves with slightly different block configuration can also be considered.

Using the technique described above, we were able to implement the $code(move)$ function, as well as $play(state, move)$ and $score(state)$. The rest of the NRPA code is generic and does not have to be specialised for SameGame. The performance of this sequential implementation of NRPA is discussed later in the experimental section (Section 4).

3 Executing NRPA on large scale computing platforms

In this section, we discuss various strategies for solving NRPA on medium scale parallel architectures (e.g. multi-cores computing platforms) and large scale architectures (e.g. cluster of computers).

Parallelizing NRPA is a challenging problem because each node in the NRPA call tree (i.e. the tree formed by the recursive NRPA calls) has sequential dependency with the previous node in the tree. To illustrate, we represent the sequential NRPA call tree in Figure 2. As we can see in this figure, each node needs to wait for the completion of the previous sibling node in order to perform the call to Adapt(). To decompose the search tree into independent tasks, we need to break some of theses dependencies. One important consequence, is that the parallel NRPA algorithm will not be strictly equivalent its sequential counterpart, and will produce different results. In this section, we propose various ways of breaking the sequential dependencies: *root parallelization*, *node parallelization/leaf parallelization* and *hybrid parallelization* and we discuss their benefits.

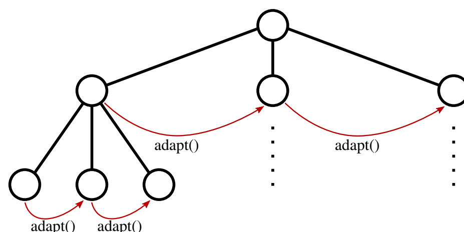

Fig. 2. Sequential NRPA call tree.

3.1 Root parallelization

As any other randomised optimisation algorithm, NRPA can benefit from parallel architectures by running one independent instance of the main procedure on each available core. When all the instances have returned a solution, the best solution among them is returned to the user.

Conceptually, this parallelization strategy is equivalent to removing the sequential dependencies between the nodes that are directly below the root (i.e. removing the long, red arrows in Figure 2). As a consequence no communication is needed between the sibling nodes and they can be executed in parallel. Because we remove the dependencies between the top-most nodes of the call tree, we call this strategy *root parallelization*.

Root parallelization has number of remarkable properties. First, it can be implemented using an unmodified version of the sequential algorithm and a simple wrapper procedure, to collect the solutions and return the best one. Second,

almost no communication is needed between the simultaneous NRPA jobs (only at the end, to collect the solutions). This guarantees that the resulting algorithm will scale well, even on clusters or grids with limited network bandwidth. Finally, not sharing intermediary results will maximise the exploration of the search space, which is useful to avoid *overfitting* the policy to a local minimum. However, this comes at the cost of *exploitation*, since the best sequences found by one NRPA job will not be used to improve the policies of the other NRPA jobs.

It is important to remark that with this strategy, the exploration vs. exploitation ratio keeps increasing as we increase the number of simultaneous NRPA jobs. As we will show later, when the number of cores is very large, increasing exploitation can be desirable.

3.2 Node and leaf parallelization

To keep the exploration vs. exploitation ratio balanced on large computing platforms, we need a parallel implementation of NRPA that can exchange intermediary results (i.e. intermediary best sequences).

We implement an alternative NRPA procedure that spawns M threads to execute the N children NRPA calls (with $M \leq N$). The parallel NRPA is only called at a certain depth controlled by a user-defined parameter L. If the level is not L, the original sequential NRPA procedure is called. To achieve this, we replace Line 9 in Algorithm 1 by the following code.

if $level == L$ **then**
  $(score, sequence) \leftarrow$NRPA-Par$(level - 1, policy)$
else
  $(score, sequence) \leftarrow$NRPA$(level - 1, policy)$
end if

Since inner calls to NRPA are called more frequently than outer calls, lowering L will increase the frequency at which the best sequences are exchanged between parallel NRPA calls, thus increasing the exploitation at the cost of more communication. If L is equal to the depth of the call tree, we refer to this strategy as *leaf parallelization*, or *node parallelization* otherwise.

Exchanging the intermediary best sequences and updating the current policy can also be done in two different ways: either the policy is shared and updated by all threads, this is the *shared policy strategy* or the policy is local to each thread, this is the *thread-local strategy.*

Shared policy strategy: In this strategy, we run M simultaneous lower level calls sharing the same parent policy. When all the calls have completed, we update the parent policy with the best sequence that was found so far. To compensate for the lower number of calls to Adapt() (N/M instead of N), we set $\alpha = M$ (instead

$\alpha = 1.0$ in the sequential version). The shared policy strategy is described in Algorithm 4.

Algorithm 4 Parallel NRPA call with shared policy.

```
 1: NRPA-Par1 (level, policy)
 2: best-score ← −∞
 3: num-iter ← N/M
 4: for i ∈ 1 ... num-iter do
 5:    for j ∈ 1 ... M do
 6:       (score_j, sequence_j) ← spawn  NRPA(level − 1, policy)
 7:    end for
 8:    wait                                            /* Thread barrier */
 9:    for j ∈ 1 ... M do
10:       if (score_j ≥ best-score) then
11:          best-score ← score_j
12:          best-sequence ← sequence_j
13:       end if
14:       policy ← Adapt(policy, best-sequence, M)
15:    end for
16: end for
17: return  (best-score, best-sequence)
```

Thread-local policy strategy: In this strategy, we create M thread-local copies of the policy and run updates in parallel on each local copy. At the end of each call, the parent policy is replaced by the best local policy if it has achieved a better score. The thread local strategy is described in Algorithm 5.

3.3 Hybrid search strategy

As discussed earlier, the root parallelization can scale on large clusters, but reduces the exploitation of intermediary results. Node and leaf parallelization can exchange intermediary results at the cost of more intensive synchronisation and communication. Therefore, they are more fitted to run on a single cluster node with shared memory.

For clusters with multi-core nodes (most frequent configuration nowadays) it is natural to combine the two approaches. On a K-node cluster with M cores each, we will run K parallel NRPA jobs (root parallelization) with M threads each (node parallelization). The best score in the K jobs will be reported.

4 Experiments

In this section, we first study the performance of the different parallelization strategies discussed in the previous section. We compare the parallelization

Algorithm 5 Parallel NRPA call with thread local policy.

```
1: NRPA-Par2 (level, policy-ref)                    /* Policy is passed by reference */
2:  best-score ← ∞
3:  for i ∈ 1...M do
4:     (score_i, sequence_i, policy_i) ← spawn NRPA-Sub(level, policy, best-score)
5:  end for
6:  wait                                                          /* Thread barrier */
7:  max ← argmax_i score_i
8:  policy-ref ← policy_max                                 /* Update parent policy */
9:  return (score_max, sequence_max)

1: NRPA-Sub (level, policy, best-score)
2:  local-policy ← policy
3:  local-best-score ← best-score
4:  num-iter ← N/M
5:  for i ∈ 1...num-iter do
6:     (score, sequence) ← NRPA(level − 1, local-policy)
7:     if score ≥ local-best-score then
8:        local-best-score ← score
9:        local-best-sequence ← sequence
10:    end if
11:    local-policy ← Adapt(local-policy, best-sequence, α)
12: end for
13: return (local-policy, local-best-score, local-best-sequence)
```

strategies by looking at their score after a fixed number of iterations, or after a fixed duration, for problem one of the SameGame test suite. In Section 4.1 we look at the performance of parallel NRPA on a single cluster node (parallel setting), and in Section 4.2 we look at the performance of NRPA on a cluster with 10 nodes (distributed setting). Finally, we use the best performing NRPA implementation to beat the state of the art at 20 NRPA instances described in [6].

Program source code: The Nrpa source code is implemented in C++. The source code used for this experiments is available online at `https://github.com/bnegreve/nrpa`.

Hardware description: All cluster nodes used in these experiments are based on 2×8-cores Intel Xeon CPU[1] (16 cores per node). In addition, the Xeon CPUs have 2-way *hyperthreading*[2] providing hardware support for 32 threads per node. For distributed executions in Section 4.2 and 4.3, we use 10 of such nodes, which makes a total of 160 cores.

Parameters description: For each implementation, vary 2 parameters: `level`: the level L in the call tree at which the parallel call are performed and `threads`: the number of threads used inside a single NRPA job. In the distributed setting, we also vary the number of simultaneous NRPA jobs.

Statistical significance of the results: All data points are averaged over at least 20 runs or more if necessary. Since the standard deviation among each NRPA run is generally high and since few points at SameGame can make a difference, we compute 95% confidence intervals and make sure that all our interpretations are based on significant results. To improve chart readability, we only show the confidence intervals when they are the most relevant (as in Section 4.2).

4.1 Parallel NRPA

In this section, we discuss the performance of the two parallel implementations of NRPA described in Section 3.2. Experiments in this section are run on a single cluster node with 16 cores (parallel setting). In this first experiment, the depth of the NRPA call tree is set to 4.

In a first set of experiments, we are interested in measuring **the cost of decomposing the NRPA search procedure into independent tasks**. As discussed earlier, the search procedures implemented in the parallel NRPA algorithms are not strictly equivalent to their sequential counterpart. In the sequential NRPA implementation, each iteration depends on the previous one, to execute NRPA in parallel, we had to break some of these dependencies. To

[1] CPU: Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
[2] `https://en.wikipedia.org/wiki/Hyper-threading`

measure the cost of doing so, we first observe the score obtained by different implementations (sequential, parallel) at the same iteration (i.e. regardless of the execution times). The result of these experiments are presented in Figure 3 (left) and Figure 3 (right) for parallelization strategy 1 (shared policy) and 2 (thread-local policy) respectively. Note that for a single run, the score can only increase with time. However, the average score can occasionally drop if the best performing run finishes early (as in Figure 3 right).

Looking at these plots can provide a number of insights. First, we can see that the cost of decomposing the search procedure is indeed significant. The sequential NRPA always reaches the best score at the end of a complete execution.

Moreover, we can also see that the higher `level` is, the lower the score. This suggests that breaking the sequential dependencies in the outermost calls (calls that are closer to the root) has a stronger negative impact on the score. Thus, leaf or deep-node parallelization should be preferred.

In a second set of experiments, we look at **the score achieved by each implementation after a 500 seconds**. The result of these experiments are presented in Figure 4 (left) and Figure 4 (right) for parallelization strategy 1 and 2 respectively.

As expected the sequential NRPA is outperformed by a number of parallel execution strategies. The best results are obtained with the first parallelization strategy, with `level`=1.

This is also the only implementation that benefits from using 32 threads. Since the execution times are similar, this suggests that increasing the exploration vs. exploitation ratio in the deep NRPA calls can be beneficial.

The second strategy achieves similar results after 500 seconds using either `level = 1` or `level = 2`. However, this strategy is penalised when using 32 cores. Further experiments show that this is mostly due to increased synchronisation overhead.

The best settings for the two parallelization strategies generate similar results. However, strategy 1 is better for parallelizing innermost NRPA (leaf parallelization) whereas the second strategy can also be used to parallelize the NPRA calls at a higher level (node parallelization).

4.2 Distributed NRPA

In this section we look at the performance of 3 distributed implementations of NRPA. We run NRPA on a 10-node cluster with various number of jobs. The results of this experiments are presented in Figure 5. The first strategy uses 160 independent NRPA jobs running on all 160 cores of the cluster (`Root-parallelization`). The second and the third strategies use the hybrid distribution strategy described in Section 3.3. `Hybrid-parallelization-1` uses a combination of root-parallelization and and leaf-parallelization implemented with shared policy strategy, whereas `Hybrid parallelization-2` uses a combination of root-parallelization and leaf-parallelization implemented with the thread-local policy strategy.

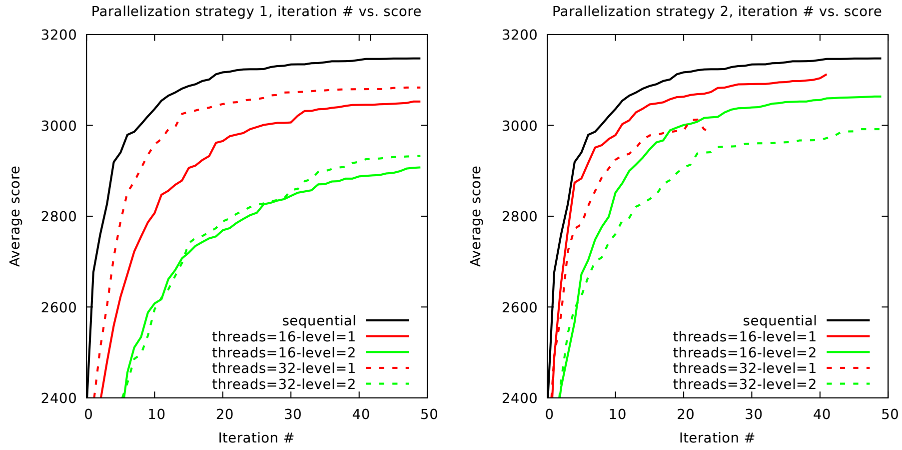

Fig. 3. Iteration based comparison for parallelization strategy 1 (shared policy) on the left and strategy 2 thread-local policy on the right.

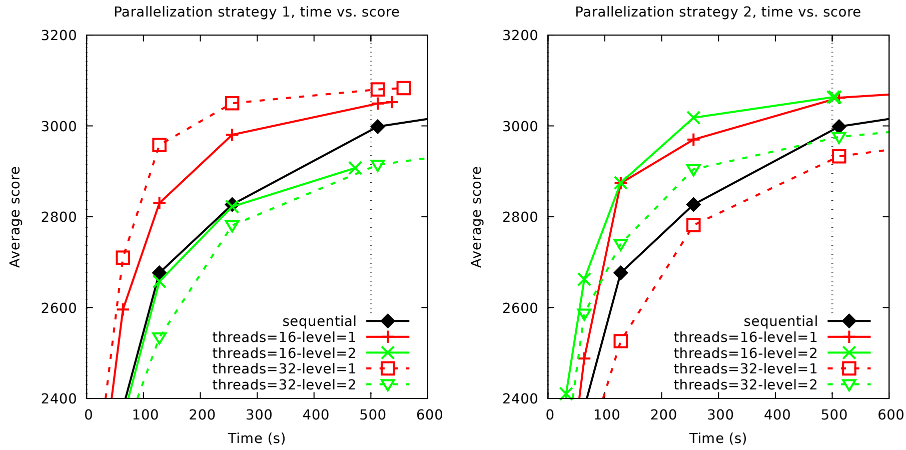

Fig. 4. Time based comparison for parallelization strategy 1 (shared policy) on the left and strategy 2 thread-local policy on the right.

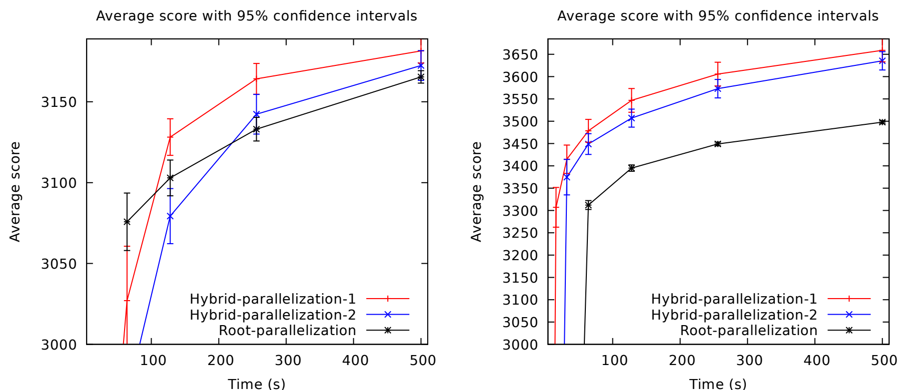

Fig. 5. Average score obtained with 3 different parallelization strategies on Board 1 (left) and Board 3 (right).

Root-parallelization achieves good performance in the early stage of the runs. Thanks to the large number of simultaneous NRPA instances, it is able to quickly find reasonably good solutions for SameGame. (High exploration, low exploitation ratio). As we spend more time improving existing solutions, Hybrid parallelization outperforms the root parallelization. Using this result, we can further speculate that pure root parallelization will not perform well with an even larger number of nodes since the exploration vs. exploitation ratio will continue to grow. With hybrid parallelization strategies, we were able to keep this ratio balanced, ultimately resulting in better scores.

Hybrid-parallelization-1 is based on the shared policy strategy running at the lowest level of the NRPA call tree (`level = 1`). It is the most communication intensive strategy but its behaviour is close to the sequential NRPA implementation. In this experiment, we can see that Hybrid-parallelization-1 performs significantly better than more scalable strategies such as Hybrid-parallelization-2 which is based on local policy NRPA implementation. We can conclude that the penalty for breaking the dependencies (mentioned in Section 3) has an important impact on the score and should be avoided when possible.

4.3 Beating Same game

In this last section, we use Hybrid-parallelization-1 to solve 20 standard SameGame boards following the experimental protocol described in [6]. We run each our algorithm (**Dist-NRPA**) with a timeout of 2 hours and report the score at termination. Each solving was only performed once. The results are reported in Table 1. As can be seen in Table 1, the distributed NRPA implementation is able to discover new best score on 15 boards, and tied on two of the 20 available boards.

Table 1. Scores at SameGame. The number in parenthesis represents the depth of the call tree.

Problem	NMCS(4)	NRPA(4)	Diversity-NRPA(4)	**Dist-NRPA (5)**
1	3121	3179	3145	**3185**
2	3813	**3985**	**3985**	**3985**
3	3085	3635	**3937**	3747
4	3697	3913	3879	**3925**
5	4055	4309	4319	**4335**
6	4459	**4809**	4697	**4809**
7	**2949**	2651	2795	2923
8	3999	3879	3967	**4061**
9	4695	4807	4813	**4829**
10	**3223**	2831	3219	3193
11	3147	3317	3395	**3455**
12	3201	3315	3559	**3567**
13	3197	3399	3159	**3591**
14	2799	3097	3107	**3135**
15	3677	3559	3761	**3885**
16	4979	5025	5307	**5375**
17	4919	5043	4983	**5067**
18	5201	5407	5429	**5481**
19	4883	5065	5163	**5299**
20	4835	4805	5087	**5203**
Total	77934	80030	81706	**83050**

5 Conclusions

We have proposed several parallel and distributed implementations of the NRPA algorithm, and evaluated their performance at solving the SameGame algorithm. We also have demonstrated how using hybrid parallelization strategies to keep the exploration vs. exploitation ratio balanced can lead to better performances than standard root parallelization. Finally, we have used our best implementation to discover 15 new best score for well known SameGame boards in a single run of less than two hours for each problem. For comparison, the competitor reports running for more than half a day. Further work will include running larger scale experiments and evaluating the performance of the distributed NRPA implementation on other known problems.

6 Acknowledgements

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organisations. (see `https://www.grid5000.fr`).

References

1. Biedl, T.C., Demaine, E.D., Demaine, M.L., Fleischer, R., Jacobsen, L., Munro, J.I.: The complexity of clickomania. More games of no chance 42, 389 (2002)
2. Breuker, D.M.: Memory versus Search in Games. Ph.D. thesis, Universiteit Maastricht (1998)
3. Cazenave, T.: Nested Monte-Carlo Search. In: Boutilier, C. (ed.) IJCAI. pp. 456–461 (2009)
4. Cazenave, T.: Playout policy adaptation with move features. Theor. Comput. Sci. 644, 43–52 (2016), http://dx.doi.org/10.1016/j.tcs.2016.06.024
5. Cazenave, T., Teytaud, F.: Application of the nested rollout policy adaptation algorithm to the traveling salesman problem with time windows. In: Learning and Intelligent Optimization - 6th International Conference, LION 6, Paris, France, January 16-20, 2012, Revised Selected Papers. pp. 42–54 (2012)
6. Edelkamp, S., Cazenave, T.: Improved diversity in nested rollout policy adaptation. In: Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz). pp. 43–55. Springer (2016)
7. Edelkamp, S., Gath, M., Cazenave, T., Teytaud, F.: Algorithm and knowledge engineering for the tsptw problem. In: Computational Intelligence in Scheduling (SCIS), 2013 IEEE Symposium on. pp. 44–51. IEEE (2013)
8. Edelkamp, S., Gath, M., Greulich, C., Humann, M., Herzog, O., Lawo, M.: Monte-carlo tree search for logistics. In: Commercial Transport, pp. 427–440. Springer International Publishing (2016)
9. Edelkamp, S., Gath, M., Rohde, M.: Monte-carlo tree search for 3d packing with object orientation. In: KI 2014: Advances in Artificial Intelligence, pp. 285–296. Springer International Publishing (2014)
10. Edelkamp, S., Greulich, C.: Solving physical traveling salesman problems with policy adaptation. In: Computational Intelligence and Games (CIG), 2014 IEEE Conference on. pp. 1–8. IEEE (2014)
11. Edelkamp, S., Tang, Z.: Monte-carlo tree search for the multiple sequence alignment problem. In: Eighth Annual Symposium on Combinatorial Search (2015)
12. Graf, T., Platzner, M.: Adaptive playouts in monte-carlo tree search with policy-gradient reinforcement learning. In: Advances in Computer Games - 14th International Conference, ACG 2015, Leiden, The Netherlands, July 1-3, 2015, Revised Selected Papers. pp. 1–11 (2015)
13. Rimmel, A., Teytaud, F., Cazenave, T.: Optimization of the Nested Monte-Carlo algorithm on the traveling salesman problem with time windows. In: Applications of Evolutionary Computation - EvoApplications 2011: EvoCOMNET, EvoFIN, EvoHOT, EvoMUSART, EvoSTIM, and EvoTRANSLOG, Torino, Italy, April 27-29, 2011, Proceedings, Part II. Lecture Notes in Computer Science, vol. 6625, pp. 501–510. Springer (2011), http://dx.doi.org/10.1007/978-3-642-20520-0_51
14. Rosin, C.D.: Nested rollout policy adaptation for Monte Carlo Tree Search. In: IJCAI. pp. 649–654 (2011)
15. Zobrist, A.L.: A new hashing method with application for game playing. ICCA journal 13(2), 69–73 (1970)