Practical Large-Scale Distributed Parallel Monte-Carlo Tree Search Applied to Molecular Design

Xiufeng Yang  \quad Tanuj Kr Aasawat  \quad Kazuki Yoshizoe
RIKEN Center for Advanced Intelligence Project
{xiufeng.yang, tanujkr.aasawat, kazuki.yoshizoe}@riken.jp

Abstract

It is common practice to use large computational resources to train neural networks, as is known from many examples, such as reinforcement learning applications. However, while massively parallel computing is often used for training models, it is rarely used for searching solutions for combinatorial optimization problems. In this paper, we propose to apply a hash function based distributed parallel Monte-Carlo Tree Search (MCTS) to a real-world problem of molecular design. By running our massively parallel MCTS combined with a simple RNN on 1024 CPU cores for 10 minutes, we achieved a score on a molecular design problem that significantly outperforms existing work. Whereas existing studies on massively scalable parallel MCTS only compare the number of rollouts, we prove the practicality of the algorithm by comparing the quality of the solutions obtained in practice. This method is generic and is expected to speed up other applications of MCTS.

1 Introduction

A survey paper on MCTS, published in 2012, has cited 240 papers, including many game and non-game applications [2]. Since the invention of Upper Confidence bound applied to Trees (UCT) [17] (the most representative MCTS algorithm) in 2006, MCTS has shown remarkable performance in various problems. Recently, the successful combination with Deep Neural Networks (DNN) in computer Go by AlphaGo [31] has brought MCTS into the spotlight. Combining MCTS and DNN is becoming one of the standard tools for solving decision making or combinatorial optimization problems. Therefore, there is a significant demand for parallel MCTS. However, in contrast to the enormous amount of parallel computing resources invested in training DNN models in many recent studies, MCTS is rarely parallelized.

Parallelizing MCTS/UCT is notoriously challenging. For example, in UCT, the algorithm follows four steps, selection-expansion-rollout-backpropagation. Non-parallel vanilla UCT updates (back-propagates) the values in the tree nodes after each rollout. The behavior of the subsequent selection steps depends on the results of the previous rollouts-backpropagation. Therefore, there is no apparent parallelism in the algorithm.

Using virtual-loss technique (explained in section 2.3), MCTS has been efficiently parallelized on shared memory environment, where the number of CPU cores are limited [4, 6, 28]. However, there is limited research on large-scale parallel MCTS using distributed memory environments. There is only one work which efficiently scales on distributed memory environment, but it is only validated for an artificial game [37].

Recently, the combination of (non-parallel) MCTS and DNN has been applied to molecular design problems, which aims to find new chemical compounds with desired properties [36, 33], utilizing...
the ability of MCTS to solve single-agent problems. In general, designing novel molecules can be formulated as a combinatorial optimization or planning problem to find the optimal solutions in vast chemical space and can be tackled with the combinations of deep generative models and search [18, 11, 14, 24, 25, 36]. However, there are no previous studies about massively parallel MCTS for molecular design.

In this paper, we apply distributed parallel MCTS to the molecule design problem. Our experimental results show that a simple RNN model combined with massively parallel MCTS outperforms existing work using more complex models combined with Bayesian Optimization or Reinforcement Learning (other than UCT).

2 Background

2.1 (non-parallel) MCTS

In 2006, Kocsis and Szepesvári proposed UCT based on a Multi-Armed Bandit algorithm UCB1 [1], which is the first MCTS algorithm having a proof of convergence to the optimal solution. It has shown good performance for many problems, including the game of Go [10].

One round of UCT consists of four steps, as shown in Fig. 1. It repeats the rounds for a given number of times or until a specified time has elapsed.

**Selection:** The algorithm starts from the root node and selects the child with the highest UCB1 value (Fig. 2[left]) until it reaches a leaf node. For each child $i$, $v_i$ is the number of visits, $w_i$ is the cumulative reward, and $V$ is the number of visits at the parent node. Exploration constant $C$ controls the behavior of UCT: the smaller, the more selective; the greater, the more explorative.

**Expansion:** If the number of visits exceeds a given threshold, expand the leaf node (add the children of the leaf to the tree) and select one of the new children for simulation. Do nothing otherwise.

**Simulation:** UCT then evaluates the node by a simulation. This step is often called playout or rollout. A simple example of rollout is to go down the tree by selecting a random child at each node until it reaches a terminal node and returns the value at the terminal node as the reward $r$ (win or loss for games). Replacing rollout with a DNN based evaluation is becoming more popular following the success of AlphaGo.

**Backpropagation:** UCT finally traverses the path all the way back to the root, and update the values of the nodes in the path ($w_i = w_i + r$, $v_i = v_i + 1$).

2.2 Solving Molecular Design using MCTS

The Simplified Molecular-Input Line-Entry System (SMILES) [35] is a standard notation in chemistry for describing molecules using ASCII strings, defined by a grammar. SMILES uses ASCII symbols to denote atoms, bonds, or structural information such as rings. For example, SMILES for Carbon dioxide is "O=C=O" where "=" means a double bond, and for Benzene, it is "C1=CC=CC=C1" or "c1cccc1" which forms a ring by connecting the two "C1"s. The search space for molecular design can be defined as a tree based on the SMILES, where the root node is the starting symbol (denoted as

![Figure 1: Four steps of (non-parallel) MCTS, with simulation for molecular design.](image-url)
UCB1 = \frac{w_i'}{v_i} + C \sqrt{\frac{\log v_i'}{v_i}}

UCBvl = \frac{w_i'}{v_i + t_i} + C \sqrt{\frac{\log (V_i + T)}{v_i + t_i}}

Green worker goes first, increments \( t_i \) on the path and reaches a leaf.

Red goes second, increments \( t_i \) and avoids the leaf because of the penalty.

Blue further avoids the same path because of the greater penalty.

"&", usually not shown) and a level \( d \) node denotes the \( d \)-th symbol in a SMILES string (see Fig. 1. The subscripts shows the depth.).

Yang et al. [36] were the first to apply non-parallel MCTS to this problem using the AlphaGo-like approach, which combines MCTS with DNN and a computational chemistry simulator. They trained an RNN model with a chemical compound database, and the RNN predicts the next symbol of a partial SMILES string which is given as the input. The model is used for expansion and rollout as described below.

**Expansion for Molecular Design:** The node (=2) in Fig. 1 denotes SMILES strings starting with "O=",. The RNN receives "O=" as input and outputs the probability of the next SMILES symbols. Instead of simply adding all symbols as child nodes, the low probability branches are pruned based on the output of the model.

**Simulation for Molecular Design:** Fig. 1 illustrates the simulation step for molecular design. Node (C3) denotes SMILES strings starting with "O=C". Firstly, a rollout generates a complete SMILES string, a new candidate molecule, by repeatedly sampling the next symbol using the model, until the model outputs the terminal symbol. Then, a computational chemistry simulator receives the SMILES string and calculates the target chemical property for the molecule, which is used to determine the reward.

### 2.3 Parallel MCTS

Non-parallel MCTS finds the most promising leaf one at a time in the selection step. In parallel MCTS, multiple workers must find the promising leaves (to launch simulations from) in parallel, hence, should find leaves speculatively without knowing the latest results.

Fig. 2 shows the example of a three workers case (a worker means a thread or a process, which runs on either a CPU or GPU). If the workers (shown in green, red, blue) follow the UCB1 formula, all end up at the same leaf node (Fig. 2(a)), and the parallelization fails.

The Virtual loss [4] let the workers find the promising but different leaves. The original UCB1 is modified to UCBvl shown in Fig. 2(b), where \( t_i \) is the number of workers currently searching in the subtree of the child node \( i \), and \( T \) is the sum of \( t_i \) of the children. It is called virtual loss because it assumes that the current ongoing simulations will obtain 0 reward. With this modification, UCBvl value is penalized based on the number of workers, which makes the subsequent workers avoid the subtrees already being explored (see Fig. 2(b)).

Parallel UCT with virtual loss was proved to improve the strength of shared-memory (using up to 16 cores) Go programs [4, 6]. Segal’s experiment on an emulated multithreaded machine that assumes no communication overhead shows speedup on up to 512 threads. However, his experiments on real distributed machines did not provide significant speedup beyond 32 cores [28].

### 2.4 Hash-driven Parallel Search

When many workers perform a search in parallel, it is essential to distribute the search space as evenly as possible. If there is an imbalance in the workload, overall performance will be less efficient because only some workers will continue the computation while the rest remain idle. If all the search
space is explicitly given in advance, this is a trivial matter. However, in case of problems such as games and combinatorial optimization, where the search space is generated while the search is performed, it is difficult to distribute the search space evenly.

Hash-driven (HD) parallel search is one of the methods for resolving this difficulty; Evett et al. [8] (PRA*) and Romein et al. [26] (TDS) developed this method independently and applied to parallel Iterative Deepening A* search. Kishimoto et al. later applied to parallel A* search [16]. HD parallel search requires a hash function that defines the value of each node. Each time the algorithm creates a node during the search, it assigns the node to a specific worker, called the home processor, based on the hash function, and achieves a near-equal load balancing if the hash function is sufficiently random (such as Zobrist Hashing [40]).

Figure 3 illustrates this method. The hash function randomly divides the tree into four partitions (shown in four colors), and each worker holds the nodes in the assigned partition in its hash table. A distinct drawback of this method is that it requires communication between workers almost every time the algorithm traverses a branch (unless the two nodes are on the same worker). The key to efficient parallel speedup is the trade-off between uniform load balancing and frequent communication. The experimental results for IDA* and A* in [8, 26, 16] prove its efficiency for these search algorithms. However, a straightforward application of HD parallel search to UCT is not efficient enough as described below.

3 Distributed Parallel MCTS and Application to Molecular Design

3.1 Hash-driven Parallel MCTS (H-MCTS)

Hash-driven parallel search efficiently shares the tree among workers. As a drawback, workers need to communicate during the execution steps frequently. Figure 4 illustrates the behavior of the four steps in Hash-driven Parallel MCTS (H-MCTS).

Selection: The worker which holds the root node selects the best child \((O_1)\) based on UCB1 formula (Figure 2(b)). It then sends a selection message, which holds information of \(O_1\), to the home processor of \(O_1\) (the green worker in Figure 4). If a worker receives a selection message, it selects the best child of the node and pass the message to another worker until the message reaches a leaf node. The worker-count \(t_i\) of each node is incremented during the step.

Expansion: If a worker receives a selection message and the node is a leaf, the expansion step is done in the same way as in non-parallel MCTS.

Simulation: Simulation is done by the home processor of the leaf (the green worker in Figure 4) in the...
same way as in non-parallel MCTS. 

**Backpropagation:** After a simulation (at \(C_3\)), a backpropagation message, which holds the reward \(r\), is sent to the parent \(= 2\). The workers pass the backpropagation message to the parent until it reaches the root node. At each node, the values are updated by the corresponding worker \((w_i = w_i + r, v_i = v_i + 1, t_i = t_i - 1)\).

To reduce the number of idle workers, the sum of the number of selection messages, backpropagation messages and ongoing simulations must be more than the number of workers. Yoshizoe et al. [37] and Graf et al. [12] independently proposed to control the number to \(N \times \#\text{workers}\) where \(N\) is the overload factor. We used \(N = 3\) following the experimental results in [37, 12].

### 3.2 Depth-First (D-MCTS) Reformulation of Hash-driven MCTS

The scalability of the above explained H-MCTS is limited because of the communication contention around the root node. In H-MCTS, all backpropagation messages are sent up until to the root node. As the number of messages increase, the workers that hold shallow nodes (especially the root node) spend more time for communication. Because of this problem, the scalability of H-MCTS quickly diminishes beyond 100 workers [37].

A solution for this communication contention problem was proposed based on the observation that the promising part of the tree does not change so often after each simulation [37]. The next selection step will likely reach a node that is close to the previously selected leaf node, as shown in Fig. 4(a). The leaf node is \(C_3\) for the 1st selection step and \(N_3\) for the 2nd selection step. In our D-MCTS, along with its own statistical information \((w, v, t)\), each node maintains a table that contains the history of the values of the siblings in the path from the root. With the help of this table, we can omit unnecessary backpropagations. After each simulation, we update the values in the node and the history table, and re-calculate the UCB\(v_i\) of the siblings, to check whether we are still in the most promising part of the tree (Note that the UCB\(v_i\) value of all the siblings can change because \(V\) is incremented). Fig. 4(b) shows a trajectory of a message in D-MCTS. After the first selection-expansion-simulation, we do not send the backpropagation message to the parent of \(= 2\). Instead we directly send a selection message to \(N_3\), since \(= 2\) is still the most promising node. Our algorithm is the first to record the history table in the nodes for this purpose. This technique dramatically reduces the number of messages sent to the root node by staying at the more promising part, thus solving the communication contention problem.

### 4 Experiment Methodology

To evaluate the quality of the solutions and to compare against state-of-the-art methods, we use the octanol-water partition coefficient (logP) penalized by the synthetic accessibility (SA) and large Ring Penalty score, a popular benchmarking physicochemical property [18, 14, 38, 25, 21] used in molecular design.

**Model and Dataset:** We use the GRU-based model and pre-trained weights publicly available on the repository of ChemTS [36], which mainly consists of two stacked GRU units. Input data represents SMILES symbols using 64-dim one-hot vectors. The first GRU layer has 64-dim input/256-dim output. The second GRU layer has 256-dim input/256-dim output, connected to the last dense layer, which outputs 64 values with softmax. The model was pre-trained using a molecule dataset that contains 250K drug molecules extracted from the ZINC database, following [18, 36, 14].

**MCTS implementation:** The GRU model explained above takes a partial SMILES string as the input and outputs the probability of the next symbol. We use this model for both Expansion and Simulation. In the expansion step, we add branches (e.g., SMILES symbols) with high probability until the cumulative probability reaches 0.95. For a rollout in the simulation step, we repeatedly sample over the model to generate a complete SMILES string. Then the string is passed to a computational chemistry tool, RDKit [19] for calculating the penalized logP score, which is commonly used in existing work. We use the reward definition described in Yang et al. [36] which is normalized to \([-1, 1]\) and consider the same value for exploration constant, \(C = 1\).
Other experimental settings: Algorithms are implemented using Keras with TensorFlow and MPI library for Python (mpi4py). All experiments, unless otherwise specified, were run for 10 minutes on up to 1024 cores of a CPU cluster (each node equipped with two Intel Xeon Gold 6148 CPU (2.4GHz, 20 cores) and 384GB of memory), and one MPI process (called worker in this paper) is assigned to one core.

5 Experiment Results

To evaluate the distributed MCTS approaches, we study the quality of the solutions obtained, scalability over up to 1024 processors, and analyze the performance bottlenecks.

Maximizing penalized logP score. Table 1 presents the penalized logP score of distributed MCTS approaches for varying number of CPU cores. With increasing number of cores available, more number of simulations can be performed in parallel, which improves the quality of the score. We performed 10 runs for each settings, and show the average and standard deviation of the best scores. For H-MCTS, which suffers from communication contention, with increasing number of cores it performs less simulations (discussed later), hence yields lower score. D-MCTS, which mitigates the issues of H-MCTS, shows strict increase in score with increase in number of cores leveraged.

Table 1: Penalized logP score (higher the better) of H-MCTS and D-MCTS, with 10 minutes time limit, averaged over 10 runs.

| Methods      | cores | 4     | 16    | 64    | 256   | 1024  |
|--------------|-------|-------|-------|-------|-------|-------|
| H-MCTS       |       | 5.83 ± 0.31 | 6.24 ± 0.59 | 7.47 ± 0.72 | 7.39 ± 0.92 | 6.22 ± 0.27 |
| D-MCTS       |       | 6.82 ± 0.76 | 8.01 ± 0.61 | 9.03 ± 0.85 | 11.46 ± 1.52 | 11.94 ± 2.03 |
| *non-parallel-MCTS |       | 6.97 ± 0.49 | 8.54 ± 0.34 | 9.23 ± 0.53 | 11.17 ± 0.88 | – |

Quality of parallel solution over non-parallel solution. As mentioned earlier, any parallel MCTS must speculatively start to search before knowing the latest search results, and it may return different outcomes from those of the non-parallel version. In addition, the exploration and exploitation trade-off of distributed MCTS is controlled via the virtual-loss based UCBs/v instead of theoretically guaranteed UCB1 [2]. Hence, it is significant to compare the quality of distributed MCTS solution with non-parallel MCTS.

The bottom row of Table 1 presents the penalized logP score for non-parallel MCTS. Note that non-parallel MCTS was run for equivalent core-hours (for example, 256 cores for non-parallel MCTS indicates it was run for 256×10 minutes on a single core; while distributed MCTS is run on 256 cores for 10 minutes). While taking much less time, the distributed-MCTS is on-par and yields higher score than non-parallel when large computing resources (i.e. with 256 and 1024 cores) are leveraged. The experiment demonstrates that MCTS can be accelerated by distributed MCTS (such as D-MCTS) without trading-off quality of the solution.

Comparison against related work. Table 2 presents the top 3 log-P scores obtained by the existing state-of-the-art work (description in section 6). The scores of D-MCTS are the best among 10 runs (10 minutes each), which outperforms the existing work significantly in maximizing the penalized logP score. It is also notable that D-MCTS significantly improved the score of the GRU-based model [36]. The bottom two lines compare the results obtained from 10 minutes random sampling from the GRU-based model with score obtained by D-MCTS which uses the same model (as mentioned in Section 4). This result suggests the possibility of improving existing work by combining their models with parallel MCTS.

Message traffic reduction at root. Table 3 presents that D-MCTS reduces the contention on the root node significantly by propagating 9× less messages to the root node, while being able to do 13× more number of simulations in the same 10 minutes time limit. Note that, even though D-MCTS sends 1.6× more back-propagation messages as it does significantly more simulations, it does not suffer from the contention because back-propagation goes up only 4.7 levels on average. On the other hand, H-MCTS is overburdened with messages as every leaf-node, after simulation, back-propagates...
Table 2: Comparison of the best three penalized logP scores

| Methods              | 1st   | 2nd   | 3rd   |
|----------------------|-------|-------|-------|
| JT-V AE (2018) [14]  | 5.30  | 4.93  | 4.49  |
| GCPN (2018) [38]     | 7.98  | 7.85  | 7.80  |
| Mol-CycleGAN (2020)  | 9.76  | 7.29  | 7.27  |
| MolecularRNN (2019)  | 10.34 | 10.19 | 10.14 |
| GRU-based [36]       | 6.47  | 5.65  | 5.01  |
| D-MCTS               | 15.13 | 14.77 | 14.48 |

Table 3: Number of back-propagation (BP) messages for H-MCTS and D-MCTS using 256 cores, with time limit of 10 minutes.

| Methods | Total simulations | Total BP | BP per simulation | BP at root |
|---------|------------------|----------|-------------------|------------|
| H-MCTS  | 62730            | 2427634  | 38.7              | 121682     |
| D-MCTS  | 815270           | 3831770  | 4.7               | 13268      |

message to its predecessors until the root node. Back-propagation per simulation is 38.7, which is equal to the average depth of the leaves where the simulations started.

Fig. 5(a) presents back-propagation messages received by each processor. D-MCTS leads to better load balance across all processors, while for H-MCTS, root’s and few other hot nodes’ home processor get overwhelmed by the messages, and suffers from extremely high load-imbalance. Please see supplementary materials for further comparison between H-MCTS and D-MCTS regarding traffic contention.

**Scalability.** Fig. 5(b) presents the scalability of H-MCTS and D-MCTS using up to 1024 cores. Scalability, as standard, measures the number of simulations performed by distributed MCTS compared to single core (i.e., non-parallel MCTS) for the same time limit.

As observed in Fig. 5(b), both H-MCTS and D-MCTS scale well when using small number of processors. But, with large number of processors, H-MCTS suffers from heavy communication contention, as described earlier, and start showing negative scaling. D-MCTS, where all the processors have uniform load balance, achieves the best scalability of $675 \times$ on using 1024 cores. However, it should be noted that the rollout becomes faster if the tree grows deeper because the rest of the SMILES strings become shorter, which makes it possible for D-MCTS to achieve more number of rollouts than the ideal case for 16 and 64 cores.

![Figure 5: (a) Number of back-propagation (BP) messages received by each of 256 processors for H-MCTS and D-MCTS. (b) Scalability](image-url)
6 Related work

6.1 Molecular Design

In molecular design, it is a common approach to use deep generative models (to generate candidate molecules), followed by optimization algorithms to focus on the promising candidates having desired molecular property (mainly Bayesian Optimisation (BO) or Reinforcement learning (RL)). Gomez-Bombarelli et al. [11] were the first to employ variational autoencoders (VAE). Kusner et al. [18] enhanced it to grammar variational autoencoder (GVAE) by combining context free grammars. Both of the above used BO for optimization. Segler et al. [29] focused on generating molecules using LSTM [13]. Olivecrona et al. [23] used simple RNN and Popova et al. [24] used GRU for generation, both combined with RL. These work use SMILES representations. Graph-based molecule generation, such as JT-VAE [14] and GCPN [38] generate molecules by directly operating on molecular graphs, optimized with BO and RL respectively. Popova et al. [25] proposed MolecularRNN, an improved RNN model by extending the GraphRNN [39] with RL.

The above mentioned work do not use MCTS. Jin et al. [15] applied MCTS for molecule design with multi-objective constraints, where MCTS is first used for extracting important substructures, then a graph-based molecule generation is trained on a data set that has these substructures and outperforms RL based methods. Yang et al. [36] combined non-parallel MCTS [36] with a simple GRU and outperformed BO based methods in penalized logP. Sumita et al. [33] later applied the same approach to a wavelength problem using a quantum mechanical simulation method (DFT).

6.2 Parallel MCTS

The first reported parallel UCT on shared memory environment [10], did not rely on virtual loss and it was less efficient. Many other work on shared memory environment use virtual loss [4, 6, 28].

Hash driven parallel search was first applied to distributed parallel MCTS by TDS-df-UCT [37] (for an artificial game) and UCT-tree-split [12] (for the game of Go) independently. TDS-df-UCT use DF-UCT for solving communication contention problem. UCT-tree-split solves the problem by preserving a replicated subtree for each worker which contains the nodes with large number of visits. UCT-tree-split periodically synchronizes the subtrees, hence, suffers from communication overhead compared to TDS-df-UCT. Our D-MCTS is similar to TDS-df-UCT except for the way of recording the history in the path. D-MCTS records the history information in the nodes (use more memory but provides more up-to-date information) while TDS-df-UCT only records the information in the messages (use less memory but delays information exchange - changing the behavior more differently from non-parallel UCT).

There are parallel MCTS methods which do not share the tree among different compute nodes. Root parallelization relies on the fact that random rollout based UCT returns different results with different seeds [4]. It does multiple runs of UCT and periodically gathers the results at the root node or for the shallow part of the tree. It is shown to be less effective compared to the other approaches mentioned above through extensive analysis [32]. Another approach, Leaf parallelization is described in [4], but its performance was lower than that of root parallelization [3].

7 Conclusion

Applying MCTS to molecular design is relatively less explored. Ours is the first work to explore distributed parallel MCTS for molecular design. The extensive experiments have shown that an efficient distributed MCTS significantly outperforms other approaches that use more complex DNN models combined with optimizations such as Bayesian Optimization or Reinforcement Learning (other than UCT).

It is not straightforward to combine our SMILES based search approach with other graph-based DNN models, but it would be an interesting future work to enhance further the performance of such complex and improved models with parallel MCTS. Also, D-MCTS could be applied to other MCTS applications, including retrosynthetic analysis to which an AlphaGo-like approach is applied [30]. Furthermore, although we analyzed the performance for molecular design, the parallelization technique is independent of the chemistry specific components. Our experiments strongly suggest that MCTS can be a better alternative for real-world optimization problems.
References

[1] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multi-armed bandit problem. *Machine Learning*, 47:235–256, 2002.

[2] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlffshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[3] T. Cazenave and N. Jouandeau. On the parallelization of UCT. In *Proceedings of Computer Games Workshop 2007*, pages 93–101. 2007.

[4] G. M. J.-B. Chaslot, M. H. M. Winands, and H. J. van den Herik. Parallel Monte-Carlo tree search. In *Proceedings of 6th International Conference on Computers and Games (CG)*, volume 5131 of *Lecture Notes in Computer Science*, pages 60–71. 2008.

[5] R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *Proc. 5th Int. Conf. on Computers and Games (CG’2006)*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. 2006.

[6] M. Enzenberger and M. Müller. A lock-free multithreaded Monte-Carlo tree search algorithm. In *Proceedings of 12th Advances in Computer Games (ACG)*, volume 5131 of *Lecture Notes in Computer Science*, pages 14–20. 2010.

[7] M. Enzenberger, M. Müller, B. Arneson, and R. Segal. FUEGO - an open-source framework for board games and Go engine based on Monte-Carlo tree search. *IEEE Trans. on Computational Intell. and AI in Games*, 2(4):259–270, 2010.

[8] M. Evett, J. Hendler, A. Mahanti, and D. Nau. PRA*: Massively Parallel Heuristic Search. *Journal of Parallel and Distributed Computing*, 25(2):133–143, Mar. 1995.

[9] M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, J. R. Cheeseman, G. Scalmani, V. Barone, G. A. Petersson, H. Nakatsuji, X. Li, M. Caricato, A. V. Marenich, J. Bloino, B. G. Janesko, R. Gomperts, B. Mennucci, H. P. Hratchian, J. V. Ortiz, J. A. Montgomery, Jr., D. W. Fox, A. P. C. Peralta, F. Ogliaro, M. J. Bearpark, J. L. Kiester, P. E. Staroverov, T. A. Keith, R. Kobayashi, J. Normand, K. Raghavachari, A. P. Bell, M. Ariani, K. Morokuma, O. Farkas, J. B. Foresman, and D. J. Fox. Gaussian 16, Revision A.12, 2016. Gaussian Inc. Wallingford CT.

[10] S. Gelly, Y. Wang, R. Munos, and O. Teytoua. Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA, 2006.

[11] R. Gómez-Bombarelli, J. N. Wei, D. Duvenaud, J. M. Hernández-Lobato, B. Sánchez-Lengeling, D. Sheberla, J. Aguilera-Iparraguirre, T. D. Hirzel, R. P. Adams, and A. Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science*, 4(2):268–276, 2018.

[12] T. Graf, U. Lorenz, M. Platzner, and L. Schaefer. Parallel monte-carlo tree search for hpc systems. In *European Conference on Parallel Processing*, pages 365–376. Springer, 2011.

[13] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[14] W. Jin, R. Barzilay, and T. Jaakkola. Junction tree variational autoencoder for molecular graph generation. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, volume 80, pages 3233–3232, 2018.

[15] W. Jin, R. Barzilay, and T. Jaakkola. Composing molecules with multiple property constraints. *arXiv preprint arXiv:2002.03244*, 2020.

[16] A. Kishimoto, A. Fukunaga, and A. Botea. Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial Intelligence*, 195:222–248, 2013.

[17] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.
[18] M. J. Kusner, B. Paige, and J. M. Hernández-Lobato. Grammar variational autoencoder. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1945–1954. JMLR. org, 2017.

[19] G. Landrum et al. Rdkit: Open-source cheminformatics. 2006.

[20] A. Liu, J. Chen, M. Yu, Y. Zhai, X. Zhou, and J. Liu. Watch the unobserved: A simple approach to parallelizing monte carlo tree search. *International Conference of Learning Representation (ICLR)*, 2020.

[21] L. Maziarka, A. Pocha, J. Kaczmarczyk, K. Rataj, T. Danel, and M. Warchol. Mol-cyclegan: a generative model for molecular optimization. *Journal of Cheminformatics*, 2020.

[22] M. Nakata and T. Shimazaki. Pubchemqc project: a large-scale first-principles electronic structure database for data-driven chemistry. *Journal of chemical information and modeling*, 57(6):1300–1308, 2017.

[23] M. Olivecrona, T. Blaschke, O. Engkvist, and H. Chen. Molecular de-novo design through deep reinforcement learning. *Journal of cheminformatics*, 9(1):48, 2017.

[24] M. Popova, M. Shvets, J. Oliva, and O. Isayev. Molecularrnn: Generating realistic molecular graphs with optimized properties. *arXiv preprint arXiv:1905.13372*, 2019.

[25] J. W. Romein, A. Plaat, H. E. Bal, and J. Schaeffer. Transposition table driven work scheduling in distributed search. In *Proceedings of 16th National Conference on Artificial Intelligence (AAAI)*, pages 725–731. 1999.

[26] C. D. Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61:203–230, 2010.

[27] R. Segal. On the scalability of parallel UCT. In *Proceedings of 7th International Conference on Computers and Games (CG)*. 2010.

[28] M. H. Segler, M. Preuss, and M. P. Waller. Planning chemical syntheses with deep neural networks and symbolic ai. *Nature*, 555(7698):484–489, Jan 2016.

[29] Y. Soejima, A. Kishimoto, and O. Watanabe. Evaluating root parallelization in Go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):278–287, 2010.

[30] M. Sumita, X. Yang, S. Ishihara, R. Tamura, and K. Tsuda. Hunting for organic molecules with artificial intelligence: molecules optimized for desired excitation energies. *ACS central science*, 4(1):120–131, 2018.

[31] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016.

[32] Y. Tian, J. Ma, Q. Gong, S. Sengupta, Z. Chen, J. Pinkerton, and L. Zitnick. ELF OpenGo: an analysis and open reimplementation of AlphaZero. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6244–6253, Long Beach, California, USA, 09–15 Jun 2019. PMLR.

[33] D. Weininger. Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of chemical information and computer sciences*, 28(1):31–36, 1988.

[34] X. Yang, J. Zhang, K. Yoshizoe, K. Terayama, and K. Tsuda. Chents: a efficient python library for de novo molecular generation. *Science and technology of advanced materials*, 18(1):972–976, 2017.

[35] K. Yoshizoe, A. Kishimoto, T. Kaneko, H. Yoshimoto, and Y. Ishikawa. Scalable distributed monte-carlo tree search. In *Fourth Annual Symposium on Combinatorial Search*, 2011.
[38] J. You, B. Liu, Z. Ying, V. Pande, and J. Leskovec. Graph convolutional policy network for goal-directed molecular graph generation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS)*, pages 6412–6422, 2018.

[39] J. You, R. Ying, X. Ren, W. L. Hamilton, and J. Leskovec. Graphrnn: Generating realistic graphs with deep auto-regressive models. In *35th International Conference on Machine Learning (ICML)*, volume 80, pages 5708–5717, 2018.

[40] A. Zobrist. A new hashing method with applications for game playing. Technical report, Department of Computer Science, University of Wisconsin, 1970.
A Virtual Loss Details

A.1 The Origin of Virtual Loss

One of the first detailed paper about parallel MCTS (Chaslot et al. 2008 [4]) already describes the use of virtual loss (page 5, section 3.3 bottom paragraph titled “Virtual loss”). The explained behavior about avoiding multiple threads visiting the same leaf is the same as we described in Section 2.3 of the main text. Yoshizoe et al. 2011 [37] explains virtual loss in more detail.

Chaslot et al. mentions that Coulom suggested, in personal communication, to assign one virtual loss per one thread during phase 1 (e.g. the selection step). However, when we personally contacted Coulom to confirm the inventor of virtual loss, it turned out that Coulom was not the inventor. We could not track further and the original inventor of virtual loss is still unknown.

Recently, in early 2020, Liu et al. proposed a method Watched the Unobserved in UCT (WU-UCT) [20], which uses a formula similar to virtual loss (eq. (2)), without comparing it against vanilla virtual loss (eq. (1)). We presume Liu et al. misunderstood or overlooked the explanation of virtual loss formula in existing work for some reasons (although they refer to Chaslot et al. [4], mentioned above). The explanation about virtual loss in the related work section in (Liu et al. [20]) is wrong. In Response to reviewer #4 (part 1 of 2) they say,

“To our best knowledge, NONE of the existing TreeP algorithms (or any existing parallel MCTS algorithm) updates the visit counts BEFORE the simulation step finishes. TreeP only updates the values ahead of time using virtual loss. This is also the case for the work [1] and [2].”

However, this is not true. To our best knowledge, most of the existing work on multithreaded parallel MCTS published after 2008 updates the values BEFORE the simulation step completes. (Actually, we can not find the benefit of adding virtual loss AFTER the simulation step.)

Liu et al. say AlphaGo (referred as [1] and [2] in their comment) do not update virtual loss BEFORE the simulation step completes. However this is presumed to be a misunderstanding caused by the explanation in the first AlphaGo paper [31]. The paper explains that virtual loss value is added during the Backup phase, so it sounds like the value is added after the simulation (explained in Methods, Search Algorithms Section, in the paper [31]). However, if you read carefully, the paper says that the virtual loss is added before the end of the simulation, and removed (subtracted) after the end of the simulation. Therefore, AlphaGo does update the visit count before the simulation step finishes. The main difference between WU-UCT and existing Virtual-loss-based parallel MCTS is the difference of the two formulas (eq. (1) and (2)).

\[
\begin{align*}
\text{UCB}_{vl} &= \frac{w_i + 0}{v_i + t_i} + C \sqrt{\frac{\log(V + T)}{v_i + t_i}} \quad (1) \\
\text{UCB}_{wu} &= \frac{w_i}{v_i} + C \sqrt{\frac{\log(V + T)}{v_i + t_i}} \quad (2) \\
\text{UCB}_{vl_{LCB}} &= \frac{w_i + \text{LCB}}{v_i + t_i} + C \sqrt{\frac{\log(V + T)}{v_i + t_i}}, \quad \text{LCB} = \min \left\{ 0, t_i \left( \frac{w_i}{v_i} - C \sqrt{\frac{\log(V + T)}{v_i + t_i}} \right) \right\} \quad (3)
\end{align*}
\]

A.2 Comparison of Virtual Loss Formulas

We compared the results of our D-MCTS using three different virtual loss formulas, the vanilla virtual loss, WU [20], and a new formula shown in eq. (3). UCB_{vl_{LCB}} (LCB stands for Lower Confidence

---

1Remi Coulom, invented the first MCTS algorithm [5] and applied to his Go program CrazyStone (before UCT).

https://openreview.net/forum?id=BJlQtJSKDB
Table 4: Penalized logP score obtained by D-MCTS using different virtual loss formulas for 256 and 1024 workers.

| Methods       | 256         | 1024        |
|---------------|-------------|-------------|
| UCB\(_{wu}\)  | 10.19 ± 0.82| 10.37 ± 1.01|
| UCB\(_{VL}\)  | 10.75 ± 1.32| 10.78 ± 0.27|
| UCB\(_{vl}\) (vanilla) | 11.46 ± 1.52| 11.94 ± 2.03|

Bound). Vanilla virtual loss assumes zero reward from the ongoing simulations and WU assumes the reward remains unchanged. UCB\(_{VL}\) assumes something between these two, assumes a decreased reward estimated by Lower Confidence Bound.

Table 4 shows the logP score results of our D-MCTS using three virtual loss formulas, averaged for 10 runs. The results suggests that, for molecular design, WU and UCB\(_{VL}\) do not improve the results over vanilla virtual loss.

### A.3 Examples of Virtual Loss in Codes

It is common to use virtual loss for parallel game programs. We show real examples of virtual loss implementations in existing open source Go or other game programs, both before and after AlphaGo. We can see the examples of the usage of virtual loss, and in all of these, the values are updated BEFORE the simulation step completes. It is also interesting to note that the majority of these work modify virtual loss equation for their implementations because they rely on different variations of UCT, such as P-UCT [27].

**Fuego:** Fuego [7], one of the best open source Go programs until 2015, is a generic game library. It started to use virtual loss in 2008 (from r677, committed on Nov. 27, 2008). In the PlayGame function starting from line 673 (in the following URL), it calls AddVirtualLoss at line 685, clearly before the playout (simulation), playouts start right below at line 693 after StartPlayouts.

https://sourceforge.net/p/fuego/code/HEAD/tree/tags/VERSION_0_3/smartgame/SgUctSearch.cpp

**ELF OpenGo:** It is a Go program and generic game library developed by FaceBook researchers [34]. Source code of the search part is in the following URL. In the single_rollout function starting from line 258, it calls addVirtualLoss at line 282. (Please note that here “rollout” means the one whole cycle of the UCT, start selection from the root, reach a leaf, do a (random) rollout, and backpropagate.)

https://github.com/pytorch/ELF/blob/113aba73ec0b9c9db00d3b03b3c439bc60fecabc89/src_cpp/elf/ai/tree_search/tree_search.h

**LeelaZero:** An open source Go program. In play_simulation function starting from line 59, virtual_loss updates the values Please note that this is a part of the selection step. play_simulation is recursively called at line 88 or 94.

https://github.com/leela-zero/leela-zero/blob/0d1791e3f4de1f52389f4e41d341484f4f66ea1e9/src/UCTSearch.cpp

**AQ:** An open source Go program. Virtual loss is added before simulations at line 142, and subtracted at line 199 or around line 247 after simulations. Also it is interesting to note that AQ uses virtual loss in two different ways, one for random rollouts and one for Neural Network based evaluation.

https://github.com/ymgaq/AQ/blob/36f6728f2f817c2fb0c69d73b00ce155582edba10/src/search.cc

### B Top Molecules

#### B.1 logP optimization

Figure 1 (a) shows the top 3 molecules by D-MCTS for penalized logP optimization. D-MCTS can design molecules with extremely high penalized logP scores, which demonstrates that our D-MCTS algorithm has the ability of identifying the promising branches. However, these molecules are not
Table 5: Wavelength (nm) score (higher the better) with 6 hours time limit, averaged over 3 runs. – indicates that experiments under the settings were not performed. + 4 and 16 cores for non-parallel-MCTS indicates the algorithm was performed for 4×6 and 16×6 hours. For 64×6, 256×6 and 1024×10 hours, non-parallel-MCTS was not performed due to the huge execution time.

| Methods         | cores | 4      | 16     | 64     | 256      | 1024    |
|-----------------|-------|--------|--------|--------|----------|---------|
| non-parallel-MCTS | 1213.5 ± 169.4 | 1850.1 ± 281.9 | –      | –      | –        | –       |
| D-MCTS          | 1038.7 ± 98.1  | 1896.9 ± 275.8 | 1960.9 ± 232.7 | 2308.5 ± 104.4 | 2412.9 ± 31.6 |

desirable because they are likely to have inaccurate predicted properties, which shows a limitation of maximizing penalized logP using an empirical tool (e.g. RDKit).

Therefore it would be interesting to apply MCTS based approach for a different optimization problem with more accurate simulations. Figure 1 (b) shows the best three molecules designed by D-MCTS for another problem, wavelength property optimization (explained below).

**B.2 Wavelength optimization**

It is possible to predict the wavelength of a given molecule using quantum mechanical simulation methods based on Density-Functional Theory (DFT). Following Sumita et al. [33], we apply our D-MCTS for finding molecules with greater absorption wavelength but limited to 2500 nm.

**Model and dataset.** Our model is on GRU, which mainly consists of one GRU layer. Input data represents 27 SMILES symbols in one-hot vectors, which represents the symbols appeared in the training dataset. The GRU layer has 27-dim input/256-dim output and the last dense layer has outputs 27 values with softmax. The model was pre-trained using a molecule dataset that contains 13K molecules (only H, O, N, and C elements included) extracted from the PubChemQC database [22].

**Calculating wavelength property and reward function.** Following [33], the wavelength property was evaluated by DFT calculations with B3LYP/3-21G* level setting using Gaussian 16 [9]. We used one core (for simplicity) for the DFT calculation. The reward function is defined as equation 4. If the generated molecules are valid, then we assign a positive reward within \((-1, 1]\) as shown in equation 4. Negative reward -1 is assigned in case the generated molecule is invalid, DFT fails, or the wavelength is greater than the limit.

\[
r = 0.01 \times \frac{\text{wavelength}}{1 + 0.01 \times |\text{wavelength}|} \tag{4}
\]

**Optimization of wavelength property.** Table 2 summarizes the optimized wavelength (nm) of D-MCTS approach for varying number of CPU cores. 3 independent runs and 6 hours time limit were applied to each setting, and the average and standard deviation of the best scores were shown.

**C Pseudo code**

The Pseudocode for H-MCTS and D-MCTS are shown in Algorithm 1 and 2. All of the workers call the function H_MCTS() or D_MCTS() respectively, and continue until timeup.

**D Comparison of the Idle Time of the Workers**

Figure 7 illustrates the average idle time of the workers in H-MCTS and D-MCTS for penalized logP optimization.
Figure 6: Top molecules designed by D-MCTS (a) Top 3 molecules with highest penalized logP score. (b) Top 3 molecules with highest wavelength/nm property.

Figure 7: Average idle time of the workers for 10 minutes run using different number of CPU cores. H-MCTS suffers from the communication contention problem near the root. When messages sent by other workers congested at the root, the workers will be in idle status and wait for the messages from root worker.
Algorithm 1 Pseudocode for H-MCTS

1: function H_MCTS()
2:   Initialize Hash_Table, Initialize job_queue, N_jobs=3×#workers,
3:   initial_job = [SELECT, root_node]
4:   if this worker is the home_proc(root_node) then
5:     for N_jobs do
6:       job_queue.Push(initial_job)
7:     end for
8:   end if
9:   while TimeIsRemaining do
10:      Receive all incoming messages and push to job_queue
11:      if job_queue.NotEmpty() then
12:         (type, node) = job_queue.Pop()
13:         if type == SELECT then
14:           node = LookUpHashTable(node)
15:           if node was not in HashTable then
16:             best_child=Selection(node)
17:             AddVirtualLoss(node.children[best_child])
18:             WriteToHashTable(node)
19:             SendMessage([SELECT, best_child], dest=home_proc(best_child))
20:           else
21:             new_node=Expansion(node)
22:             Reward=Simulation(new_node)
23:             node.Update(Reward),
24:             WriteToHashTable(node)
25:             SendMessage([BP, node], dest=home_proc(node.parent))
26:           end if
27:         else if type == BP then
28:           parent = LookUpHashTable(node.parent)
29:           RemoveVirtualLoss(parent.children[node])
30:           parent.Update(node.reward)
31:           WriteToHashTable(parent)
32:           if parent != root_node then
33:             SendMessage([BP, parent], dest=home_proc(parent.parent))
34:           else
35:             best_child=Selection(parent)
36:             AddVirtualLoss(parent.children[best_child])
37:             SendMessage([SELECT, best_child], dest=home_proc(best_child))
38:           end if
39:         end if
40:      end while
41:   end while
42: end function
Algorithm 2 Pseudocode for D-MCTS

1: function D_MCTS()
2:    Initialize Hash_Table, Initialize job_queue, N_jobs=3 × #workers,
3:    initial_job = [SELECT, root_node, None]
4:    if this worker is the home_proc(root_node) then
5:        for N_jobs do
6:            job_queue.Push(initial_job)
7:        end for
8:    end if
9:    while TimeIsRemaining do
10:       Receive all incoming messages and push to job_queue
11:       if job_queue.NotEmpty() then
12:          (type, node, ucb_history) = job_queue.Pop()
13:          if type == SELECT then
14:             node = LookUpHashTable(node)
15:             if node was not in HashTable then
16:                 node.UpdateUCBHistory(ucb_history)
17:                 best_child=Selection(node)
18:                 AddVirtualLoss(node.children[best_child])
19:                 WriteToHashTable(node)
20:                 ucb_history.Append(node.children)
21:                 SendMessage([SELECT, best_child, ucb_history], dest=home_proc(best_child))
22:             else
23:                 new_node=Expansion(node)
24:                 Reward=Simulation(new_node)
25:                 node.Update(Reward),
26:                 WriteToHashTable(node)
27:                 SendMessage([BP, node, node.ucb_history], dest=home_proc(node.parent))
28:             end if
29:          else if type == BP then
30:             parent = LookUpHashTable(node.parent)
31:             RemoveVirtualLoss(parent.children[node])
32:             parent.Update(node.reward)
33:             parent.UpdateUCBHistory(ucb_history)
34:             WriteToHashTable(parent)
35:             current_best_node = parent.ucb_history.GetCurrentBest()
36:             if parent == current_best_node OR parent == root_node then
37:                 best_child=Selection(parent)
38:                 AddVirtualLoss(parent.children[best_child])
39:                 WriteToHashTable(parent)
40:                 ucb_history.Append(parent.children)
41:                 SendMessage([SELECT, best_child, ucb_history], dest=home_proc(best_child))
42:             else
43:                 SendMessage([BP, parent, parent.ucb_history], dest=home_proc(parent.parent))
44:             end if
45:          end if
46:       end if
47:    end while
48: end function