Journal of Parallel and Distributed Computing 193 (2024) 104944

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

PPB-MCTS: A novel distributed-memory parallel partial-backpropagation Monte Carlo tree search algorithm

Yashar Naderzadeh [a], Daniel Grosu [a,*], Ratna Babu Chinnam [b]

[a] Department of Computer Science, Wayne State University, 5057 Woodward Ave., Detroit, 48202, MI, USA
[b] Department of Industrial and Systems Engineering, Wayne State University, 4815 Fourth Street, Detroit, 48202, MI, USA

ARTICLE INFO

Keywords:
Parallel algorithms
Monte Carlo tree search
Job shop scheduling

ABSTRACT

Monte-Carlo Tree Search (MCTS) is an adaptive and heuristic tree-search algorithm designed to uncover sub-optimal actions at each decision-making point. This method progressively constructs a search tree by gathering samples throughout its execution. Predominantly applied within the realm of gaming, MCTS has exhibited exceptional achievements. Additionally, it has displayed promising outcomes when employed to solve NP-hard combinatorial optimization problems. MCTS has been adapted for distributed-memory parallel platforms. The primary challenges associated with distributed-memory parallel MCTS are the substantial communication overhead and the necessity to balance the computational load among various processes. In this work, we introduce a novel distributed-memory parallel MCTS algorithm with partial backpropagations, referred to as *Parallel Partial-Backpropagation MCTS* (PPB-MCTS). Our design approach aims to significantly reduce the communication overhead while maintaining, or even slightly improving, the performance in the context of combinatorial optimization problems. To address the communication overhead challenge, we propose a strategy involving transmitting an additional backpropagation message. This strategy avoids attaching an information table to the communication messages exchanged by the processes, thus reducing the communication overhead. Furthermore, this approach contributes to enhancing the decision-making accuracy during the selection phase. The load balancing issue is also effectively addressed by implementing a shared transposition table among the parallel processes. Furthermore, we introduce two primary methods for managing duplicate states within distributed-memory parallel MCTS, drawing upon techniques utilized in addressing duplicate states within sequential MCTS. Duplicate states can transform the conventional search tree into a Directed Acyclic Graph (DAG). To evaluate the performance of our proposed parallel algorithm, we conduct an extensive series of experiments on solving instances of the Job-Shop Scheduling Problem (JSSP) and the Weighted Set-Cover Problem (WSCP). These problems are recognized for their complexity and classified as NP-hard combinatorial optimization problems with considerable relevance within industrial applications. The experiments are performed on a cluster of computers with many cores. The empirical results highlight the enhanced scalability of our algorithm compared to that of the existing distributed-memory parallel MCTS algorithms. As the number of processes increases, our algorithm demonstrates increased rollout efficiency while maintaining an improved load balance across processes.

1. Introduction

In recent years, Monte Carlo Tree Search (MCTS) algorithms (e.g., Upper Confidence bounds applied to Trees (UCT) [1]), have attained fame due to the remarkable outcomes obtained when combined with machine learning (ML) algorithms and applied to complex artificial intelligence (AI) problems such as the game Go [2] and video games [3]. Constructing the search tree in MCTS requires performing a significant number of Monte Carlo rollouts (i.e., starting a path from the root of the search tree and taking actions along the path to reach a terminal node and receiving a reward) [4]. This fact motivates parallelizing MCTS at a large scale. Most MCTS algorithms consist of four phases: *selection*, *expansion*, *rollout*, and *backpropagation*. In the *selection* phase, starting from the root of the search tree, nodes with the highest heuristic values are selected recursively until a node that is not fully expanded is reached. Then, in the *expansion* phase, the tree grows from that node by adding

* Corresponding author.
E-mail address: dgrosu@wayne.edu (D. Grosu).

https://doi.org/10.1016/j.jpdc.2024.104944
Received 22 September 2023; Received in revised form 11 May 2024; Accepted 18 June 2024
Available online 26 June 2024

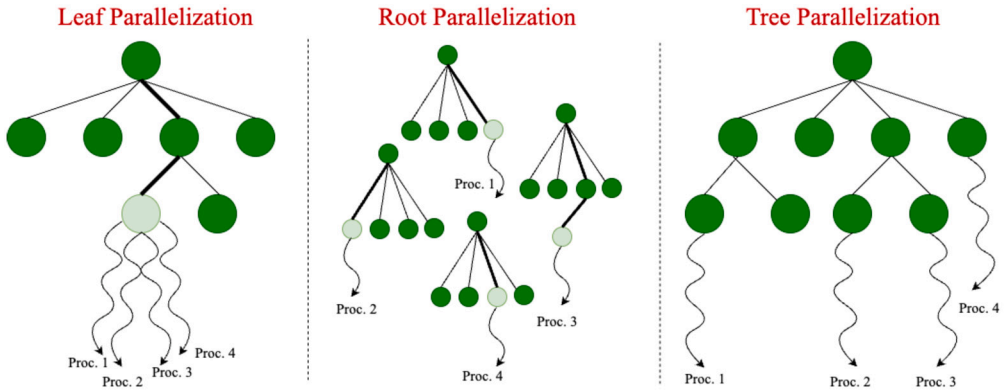


Fig. 1. Three types of MCTS parallelization *leaf parallelization*, *root parallelization*, and *tree parallelization* respectively, from left to right. Here, four processes (Proc.) are assumed for each parallelization type. Parallelization begins at *rollout* phase in leaf parallelization. Root parallelization induces several search trees. In tree parallelization, all processes construct one search tree in coordination with each other.

a new child node as a leaf to the tree. In the *rollout* phase, the new leaf is evaluated by random action selection to meet a terminal state[1] and receive a reward. In the *backpropagation* phase, the reward obtained in the rollout is backpropagated in the tree from the leaf to the root to update the heuristics of the tree nodes. All information from the previous rollouts conducted at a parent node's subtree is required to make proper decisions in the selection phase. Selections should introduce an exploitation-exploration trade-off in the search tree. There are three primary parallelization methods for MCTS [5]: leaf parallelization, root parallelization, and tree parallelization. *Leaf parallelization* involves initiating parallelization from the newly expanded leaf within the tree during the *expansion* phase [6]. *Root parallelization*, on the other hand, entails the concurrent execution of multiple trees, each representing an individual MCTS instance, with the final decision being aggregated from the outcomes of all trees.

In *tree parallelization*, a joint search tree is constructed by a cluster of processes. Although this method can produce a substantial search tree, effectively parallelizing MCTS in this manner without compromising its efficacy poses a challenge [5,7,8]. This difficulty arises primarily from the sequential nature of most MCTS algorithms. In parallel MCTS, the simultaneous execution of rollouts from various child nodes may result in suboptimal decisions during the *selection* phase.

A class of distributed MCTS algorithms is Multi-agent MCTS [9] and decentralized MCTS [10–12], which find utility in applications within robotic industries. In this group of MCTS algorithms, a set of agents grows a single or a set of search trees, each tree belongs to an agent, in coordination with each others. Depending on whether a single search tree is constructed or a set of search trees, algorithms in this class can be categorized into a type of MCTS parallelization.

Fig. 1 illustrates the three distinct types of parallelizing MCTS. In this figure, it is assumed that four processes are available. In leaf parallelization, each process is tasked with executing a *rollout* from the newly appended tree node within the search tree. However, it is noteworthy that all the other phases of MCTS are executed sequentially. In root parallelization, each process constructs an independent tree by performing the sequential MCTS. In tree parallelization, processes execute simultaneously and expand a single tree from different subtrees.

An effective strategy to the tree parallelization of MCTS is to introduce *virtual loss* into the MCTS [5,7]. In *virtual loss*, estimations over the ongoing rollouts are adversely added to their corresponding actions to decrease their chance of being selected while increasing the chance of choosing other actions. This method has been shown to work well for shared-memory parallel MCTS where the computational resources are restricted [13]. However, to improve the scalability of parallel MCTS, distributed-memory parallel MCTS algorithms have been proposed in [14–17].

These algorithms are inspired by Transposition-Table Driven Scheduling (TDS) [18]. In distributed-memory parallel MCTS algorithms, a search tree is progressively constructed by worker processes, where each worker hosts a set of tree nodes according to a shared transposition table. There are two types of messages in distributed MCTS, *forward* and *backward messages*. Forward messages are used for traversing downward the search tree, while the backward messages update the tree nodes with the results of the rollouts.

Based on the observation that the search tree often grows from the promising tree nodes, previous research [14,15] suggested that to have more exploration, it is unnecessary to perform a complete *backpropagation*. To detect where to stop the *backpropgation*, they propose augmenting the messages between workers with an information table containing the heuristics of all tree nodes on the path from the root to the leaf where the rollout takes place. The *backpropagation* stops at the tree node that still has the best heuristic compared to its siblings after updating the information table by the most recent rollout. This is possible because the information table contains the heuristics of all selected nodes and their siblings in the forward path.

There are two considerations with the information table. First, the heuristics of the tree nodes in the information table are not the most updated heuristics since they are attached to the table on the downward path. When we reach a parent node, we want to check whether the child selected previously still has the highest heuristic to keep continuing the exploration from that child. However, other children might have been updated in the meantime, and these updates are not reflected in the information table. Second, the table size is linearly proportional to the size of available actions at each node along the downward path, and they are passed among the workers. This introduces a large communication overhead when solving problems with a large number of actions at each tree node. The other issue with distributed-memory parallel MCTS that has not been addressed in the literature is how to deal with duplicate tree nodes. These nodes transform a search tree into a Directed Acyclic Graph (DAG). This problem has been well-studied in the context of sequential MCTS and robotic task and motion planning [19–21,12]. Existing methods often rely on employing a transposition table to verify whether the search tree has ever explored a state observed by MCTS. However, in distributed-memory tree parallelization of MCTS, employing a central transposition table can lead to considerable communication cost.

We propose a new distributed-memory tree-based parallel MCTS algorithm called *Parallel Partial-Backpropagation MCTS* (PPB-MCTS), that performs exploration at the promising tree nodes with partial backpropagations. At the same time, it does not need to carry or store any information table. We also propose two methods to address the prob-

[1] Note that the terms "node" and "state" are used interchangeably unless otherwise stated.

Table 1
Related work: parallel MCTS algorithms.

Paper \ Parallelization method	Leaf	Root	Tree
Chaslot et al. [5]*	✓	✓	✓
Steinmetz et al. [6]**		✓	✓
Mirsoleimani et al. [8]			✓
Yoshizoe et al. [14]			✓
Yang et al. [15]			✓
Kurzer et al. [28]	✓		
Cazenave et al. [29]	✓		
Enzenberger et al. [30]			✓
This paper			✓

*In this work all three types of parallelization are implemented.
**In this work, root and tree parallelizations are compared with each other.

lem of duplicate tree nodes. In Table 1 we provide a summary of the existing parallel MCTS algorithms and classify them according to the tree types of parallelization approaches.

Parallel MCTS algorithms have often been applied to and studied for board games [22–24]. There is also research on using parallel MCTS in other domains. However, they mostly focused on shared-memory parallel MCTS algorithms. A successful example of the application of distributed parallel MCTS to other domains is the molecular design problem [15]. In this paper, we apply our proposed distributed-memory MCTS algorithm to the Job-Shop Scheduling Problem (JSSP). JSSP is a well-known NP-hard problem with wide applications in manufacturing and transportation [25,26]. In this problem, a set of jobs and machines are given where each task in a job must be executed on a specific machine following its priority in its job. The objective is to find a schedule that minimizes the makespan [27].

Using MCTS for solving JSSP has been studied before [31,32]. However, in those works, the priority dispatching rules such as First-In First-out (FIFO) or Most WorK Remaining (MWKR) are assumed to be the actions at the tree nodes. Our approach is completely different, we model the JSSP as a deterministic Markov Decision Problem (MDP) and enumerate all valid actions according to the decision time, the priority of tasks in a job, the current status of the machines, and the remaining tasks in a job. This generates a large number of valid actions, particularly at the root and the tree nodes close to it that grow exponentially with the problem size. This modeling helps to assess our proposed algorithm against other distributed MCTS algorithms that mostly assume a small to moderate number of actions at each tree node. To investigate the scalability of our algorithm, we compare it against other distributed-memory parallel MCTS algorithms for various numbers of worker processes over multiple compute nodes for several JSSP instances and against its sequential counterpart.

MCTS has been employed to solve the Maximal-Coverage Problem (MCP) within the domain of robotic path planning [11,33]. This problem is a variant of WSCP. Furthermore, a decentralized MCTS algorithm for solving MCP by maintaining a probability distribution of the joint-action space for each robot has been proposed in [10]. The main difference between WSCP and MCP when MCTS is employed to solve them is that a search in MCP has a fixed length while a search in WSCP has length that fluctuates based on the universe coverage. However, when the *backpropagation* phase is partial, this distinction may not hold true. In this work, we use our proposed algorithm (PPB-MCTS) to solve a group of randomly generated WSCP problem instances. Similar to JSSP, we conduct a comparative analysis of PPB-MCTS's performance against that obtained by the existing sequential and distributed-memory MCTS algorithms.

Our contributions. Our *main contributions* are as follows:

- We propose a new distributed-memory parallel MCTS algorithm with *partial-backpropagation* (called PPB-MCTS) that reduces the communication overhead required by previous MCTS algorithms.
- We use the proposed algorithm to solve several instances of JSSP and WSCP as two examples of NP-hard combinatorial optimization problems where the number of actions grows exponentially with the problem size.
- We conduct an extensive experimental analysis of our algorithm. The experimental results show that our algorithm achieves better scalability than the existing distributed-memory parallel MCTS algorithms by performing more rollouts as the number of processes increases while providing better load balance among processes.
- PPB-MCTS often outperforms previous distributed-memory MCTS algorithms in identifying (sub)-optimal solutions through the execution of an increased number of rollouts, particularly evident in problem instances characterized by a high volume of actions.

2. Background

2.1. Sequential upper confidence bounds applied to trees MCTS (UCT-MCTS)

In MCTS, a decision tree is iteratively processed in four phases: *selection*, *expansion*, *rollout*, and *backpropagation*. These four phases repeat for an assigned computational budget. Fig. 2 shows these phases in order of their execution. In the *selection* phase, starting from the root in the existing built tree, as the first parent node, a parent selects its most promising child node based on the number of rollouts and the previously collected rewards, which make the heuristic of a node. The selected child then becomes the new parent and must choose a child with the highest heuristic. This recursive procedure continues up to a tree node that is not fully expanded. In Fig. 2, C_3 is the most promising child of the root, which then becomes the new parent node. Among the children of C_3, C_{3-2} has the highest heuristic and is selected. The node selection policy up to this node is called the *tree policy*. In the *expansion phase*, the tree is expanded from the node by attaching a new node to the tree. In the scenario depicted in Fig. 2, the new node added to the tree is C_{3-2-1}. The trajectory from the root to this node is denoted by $\boldsymbol{P}^{l}_{root\rightarrow s^l}$, where l is the iteration number in which the trajectory is sampled, and s^l is the new leaf node in the tree. A *rollout* from s^l is conducted following a random policy to visit a terminal state. In Fig. 2, this is shown as the red-curved line stemming from node C_{3-2-1} and ending at the terminal node T. The environment returns a reward r^l at the terminal node. After receiving the reward r^l, the leaf s^l and the selected nodes in iteration l are updated by *backpropagating* r^l on the trajectory $\boldsymbol{P}^{l}_{root\rightarrow s^l}$.

Here, we consider UCT-MCTS [1] as the most frequently employed MCTS algorithm. In UCT-MCTS, the heuristic (Upper Confidence Bounds (UCB1)) that is used for choosing nodes in the *selection* phase is

$$\mathrm{UCB1}^{l}(s) = \frac{w^{l}(s)}{N^{l}(s)} + C\sqrt{\frac{\log N^{l}(s.parent)}{N^{l}(s)}}, \tag{1}$$

where $N^l(s)$ is the number of times state s is visited in iteration l, $s.parent$ denotes the parent of state s, $w^l(s)$ is the accumulated reward in state s up to lth iteration, C is a hyperparameter to control the exploitation-exploration trade-off. $\mathrm{UCB1}^l(s)$ estimates the state value $V(s)$ when the agent follows the tree policy. $V(s)$ indicates the expected cumulative reward that can be achieved by starting from state s and following a policy. The policy in UCT-MCTS is to select the child with the highest UCB1. Rollouts sampled on the tree steadily improve the tree policy. The tree policy exploits the first term in (1) to select the tree nodes with better average rewards. In contrast, the second term encourages the tree policy for more exploration of other nodes. The

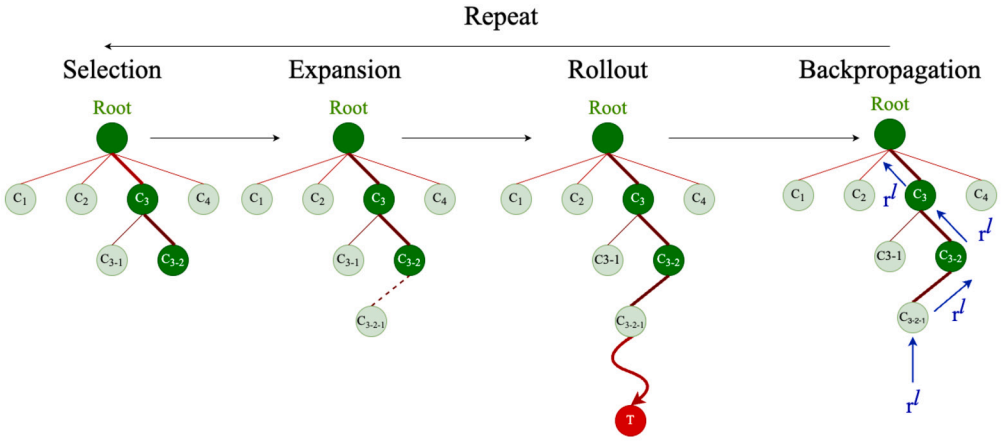


Fig. 2. Sequential UCT-MCTS consists of four phase: *selection*, *expansion*, *rollout*, and *backpropagation*. When the rollout is completed, meaning that a terminal state T is reached, the environment returns reward r^l. After that, nodes on the path $P^l_{root \to C_{3-2-1}}$ are updated according to the update rules. In this figure s^l is C_{3-2-1}.

update rules in the *backpropagation* step are $N^{l+1}(s) = N^l(s) + 1$, and $w^{l+1}(s) = w^l(s) + r^l$, for $s \in P^l_{root \to s^l}$.

2.2. Distributed-memory parallel MCTS

There are three approaches to parallelizing MCTS, *leaf parallelization*, *root parallelization*, and *tree parallelization* [5]. In *leaf parallelization* [29,28], the selection and expansion phases run on a single process. Then, several rollouts are initiated from the expanded leaf node, each executed by a separate process. In *root parallelization* [29], several MCTS jobs are assigned to a group of processes, each job to one process. The best solution from the tree searches developed by processes is then reported. In *tree parallelization* [30], one search tree is created by all processes, where each process hosts a part of tree nodes. *Tree parallelization* is the most prevailing parallel MCTS implementation for distributed-memory platforms. Tree parallelization can construct a larger search tree compared to other implementations [14]. This is because different processes can explore a unique search tree at the same time. However, providing load balancing on processes, preventing a node from being explored by multiple processes, reducing the communication overhead, and having the most updated information for selecting the most promising child node are challenges that must be addressed in this type of parallelization. Next, we discuss these challenges in more detail.

2.2.1. Load balancing

Transposition Table Driven Scheduling (TDS) [18] is an approach for achieving load balance in distributed best-first search algorithms (e.g., iterative deepening A* (IDA*) [34]). In TDS, each process stores a group of tree nodes as entries of a transposition table. In order for a tree node to send a message to another tree node, it must find the transposition-table key of the other node. This is enabled by sharing a hash function among the processes. The hash function takes a tree node's state and returns the transposition-table key and the home-process id of the node's host. Fig. 3 shows the structure of the transposition-table for an implementation with four processes. Each process keeps relevant parts of the transposition-table and uses the hash function shared among the processes to find where to send its outgoing messages. As shown in Fig. 3, entries of the transportation table are divided almost uniformly to provide load-balance among the processes. Inspired by TDS, several researchers [14,17,35] proposed distributed-memory MCTS algorithms in which the tree nodes are distributed evenly among the processes using the transposition table.

2.2.2. Preventing a node from being explored by multiple processes

The other challenge with tree parallelization is preventing a node from being simultaneously explored by multiple forward paths for efficient exploration over the tree. The *virtual loss* technique [5] was

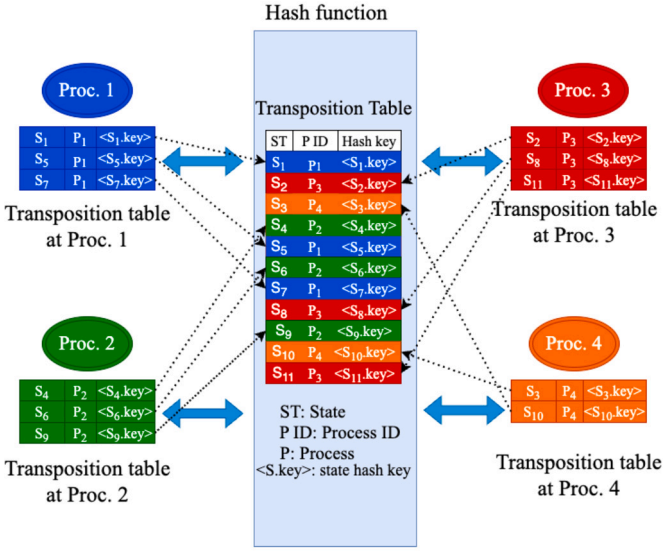


Fig. 3. An example of a transposition table where each entry in the transposition table is stored in one of four processes. In this example, the hash function maps 2 or 3 tree nodes (states) to each process id.

proposed to address this challenge. The idea is to temporarily decrease the value of a node selected on a path of an uncompleted rollout. This technique decreases the chance of the node being selected by another forward path. The induced *virtual loss* is removed from the node when the corresponding rollout is completed, and the node is updated in the backpropagation phase. The heuristic that is used for choosing nodes in the selection phase is modified to reflect the *virtual loss* as follows:

$$\text{UCB}^l vl(s) = \frac{w^l(s) + v_l\, t(s)}{N^l(s) + t(s)} + C\sqrt{\frac{\log(N^l(s.parent) + t(s.parent))}{N^l(s) + t(s)}}, \quad (2)$$

where $s.parent$ is the parent of s, $t(s)$ is the number of ongoing rollouts in the subtree of s, and v_l is a pessimistic estimation of the reward of an ongoing rollout. v_l is mostly assumed to be zero in (2). The reason is that UCBvl is often applied to two-player games where the outcome is either win (1), or lose (0). In other domains, the reward can be mapped into the range [0,1]. As a result, each ongoing rollout lessens UCBvl, assuming that rollout leads to the worst outcome. This is reflected in (2), where the denominators increase while the numerators remain the same or grow strictly slower with respect to the denominators. Fig. 4 demonstrates how using UCBvl can encourage efficient exploration.

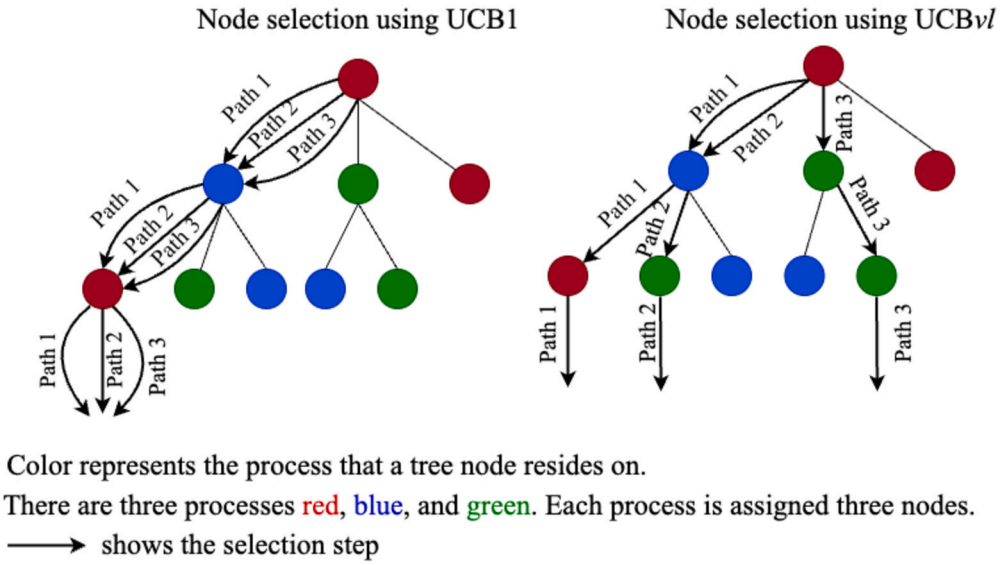


Fig. 4. When the UCB1 is used in the node selection, all search paths select the same tree node, and it prevents exploring other tree nodes. UCBvl penalizes the node selected by an ongoing path to lower its chance of being selected by other search paths. It compels exploration of other nodes.

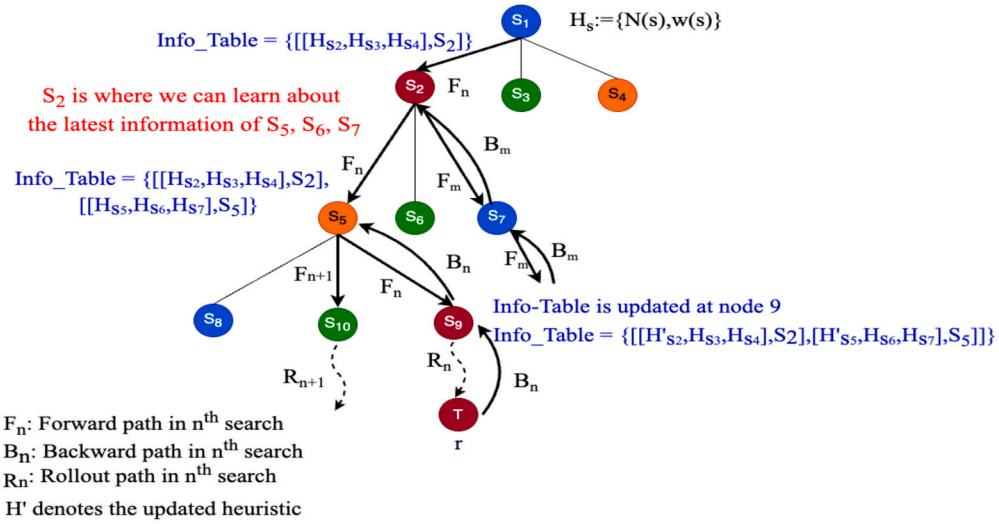


Fig. 5. The information table stores heuristics of selected nodes and their siblings while traversing the forward path. In the backpropagation phase, the node where the rollout is initiated from, updates the table and backpropagates to its parent. The parent checks the table and decides if it is still the best node. An information table is required for each search over the tree.

2.2.3. Reducing the communication overhead

In the TDS-MCTS algorithms, all search paths start from the root. As a result, the process in charge of the tree root incurs significant communication overhead that adversely affects this plain MCTS with TDS. To alleviate this communication overhead on the root home-process, TDS-Depth-First(DF)-UCT and Massively-Parallel (MP)-MCTS were proposed in [14] and [15], respectively. These two algorithms work based on the notion that the search tree often grows from the promising nodes. As long as the $UCB1$ of one of the previously selected nodes on a path from the root to a leaf is still the highest, considering the new reward, backpropagation can stop there.

This partial backpropagation is based on the notion that the search frequently stays around the promising nodes, and the node selection operation degrades slightly. MP-MCTS was applied to solving the molecular design problem [15], and the results show that MP-MCTS outperformed the sequential MCTS and TDS-UCT-MCTS. One possible reason for this performance is that more rollouts can be accomplished since the communication overhead drops by using these partial backpropagations. To detect where to stop in the backward path, both TDS-DF-UCT and MP-MCTS must carry an information table filled with heuristics, $H_s = \{N(s), w(s)\}$, of all selected nodes and their siblings, to determine where to stop on the backpropagation path. An example showing the idea behind these algorithms is given in Fig. 5. The color of each node in the tree represents the process in which it resides. Starting at the root S_1, the information table contains information about root's children S_2, S_3, and S_4. Presuming that node S_2 has the highest heuristic, it is selected, and the information table is sent to its home-process, where the information on its children is appended to the information table. If we assume that node S_5 is the selected node from S_2, we expand it by attaching node S_9 to the tree and appending the information of node S_5's children to the information table. Then, a rollout is initiated from node S_9, and the information table is modified according to the received reward r and sent back to node S_5 home-process. The S_5 home-process checks the information table to see whether S_5 still has the highest heuristic compared to nodes S_6 and S_7. If this is the case, node S_5 expands by adding a new node or selects among its children depending on whether it has been fully expanded. Otherwise, the information table is

Table 2
Notation.

Notation	Description
$\text{UCB1}^l(s)$	UCB value of state s up to iteration l
$w^l(s)$	Accumulated reward at state S up to iteration l
$N^l(s)$	Number of times state s is selected up to iteration l
$V(s)$	Value of state s
$P^l_{root \to s^l}$	A path from root to state s at iteration l
$t(s)$	Number of ongoing rollouts from state s
v_l	A pessimistic estimation of an ongoing rollout
`MBackProp`	Backpropagation message type
`MSearch`	Search message type
$\mathcal{M}$	Set of machines in JSSP
$\mathcal{J}$	Set of jobs in JSSP
O^j_i	Task i in job j
m^j_i	Designated machine for task O^j_i
d^j_i	Length of task O^j_i
S^j_i	Starting time for task O^j_i
$S^j_i(t_k)$	Potential starting time at time t_k for task
$I^{t_k}_{i,j}$	Assignment indicator for O^j_i to machine m^j_i at t_k
$V^\pi(s)$	Value of state s under policy π
$R_{a_t}(s_t, s_{t+1})$	Reward for action a_t leading from state s_t to s_{t+1}
r^i	Rewards gained from rollout i
x_i	Indicator variable for subset i in WSCP
w_i	Weight of subset i in WSCP
$\mathcal{B}(\mathcal{S}, \mathcal{U}, p^{\mathcal{G}})$	Erdős-Rényi bipartite graph $\mathcal{G}$
$\mathcal{U}$	Universe in WSCP
$\mathcal{S}$	Set of subsets in WSCP
$p^{\mathcal{G}}$	Probability an element belongs to a subset in WSCP
$nprocs$	Number of Processes

backpropagated to node S_2. It must be noted that the rollouts occur locally on the home-process of the nodes where they are initiated.

The size of the information table depends on the number of child nodes at each selected node and the search tree height, which increases as the tree grows. For some problems, these parameters are large, and the information table can lead to a large communication overhead. One example is JSSP, where the number of valid actions at the root and tree nodes near the root grows exponentially with the problem size.

2.2.4. Having the most up-to-date information for selecting the most promising child node

MP-MCTS and TDS-DF-UCT must carry a table containing information on the visited nodes and their siblings along the downward path to perform partial backpropagation. However, the information stored in the table is not the most updated heuristics of the nodes. The reason is that the information is collected on the search path while there may be other ongoing searches over the siblings and if one of them completes before the time to decide where to stop in the backpropagation, the accuracy of the information in the information table deteriorates. This scenario is shown in Fig. 5, where node S_5 home-process uses the information table to check if S_5 still has the highest UCBvl value by examining the information table. However, this information is old since when the information was collected, a rollout from node S_7 had already been launched and completed before the rollout initiated from the S_5 subtree terminates.

The other concern with the information table is that its size depends on the number of nodes it stores. This number is a function of where in the tree a rollout is launched, the tree's depth, tree's width, and the number of children at each selected node over the search path. Many JSSP instances often show these characteristics, and the table maintaining the necessary information is expected to lead to a high communication overhead for wide and tall trees. Here, we propose a new approach that can still perform partial backpropagation without carrying an information table. This is achieved by performing one additional backpropagation step.

Table 2 and Table 3 present the notations and the algorithm names abbreviations that are used throughout the paper.

Table 3
Acronyms used in the paper.

Acronym	Description
UCB	Upper Confidence Bound
UCT	Upper Confidence Bound applied to Tree
TDS-MCTS	Transposition-Table Driven Scheduling MCTS
MP-MCTS	Massively Parallel MCTS
PPB-MCTS	Parallel Partial Backpropagation MCTS
JSSP	Job Shop Scheduling Problem
WSCP	Weighted Set Cover Problem

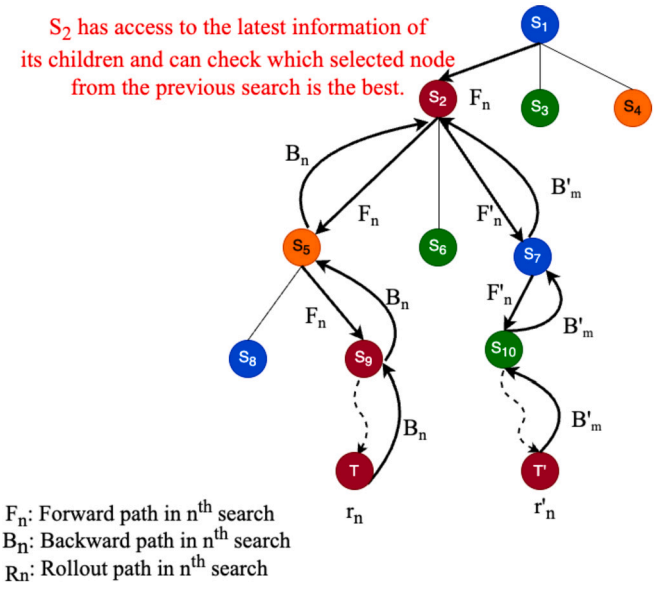


Fig. 6. Both nodes S_5 and S_7 that are children of S_2 are updated. They send the rewards they achieved from the rollouts on their subtrees to node S_2. Node S_2 knows the latest information about its children. To avoid a complete backpropagation, it can initiate a search if a child that backpropagates a reward to it is still the child with the highest UCBvl.

3. PPB-MCTS: parallel partial-backpropagation MCTS

We propose a new parallel algorithm for distributed-memory MCTS (called PPB-MCTS) performing partial backpropagation and addressing the concerns with the previous MCTS algorithms discussed above. A search tree constructed by MCTS often grows under the subtrees of the promising nodes, with the goal of finding a sub-optimal solution. This fact can be exploited to perform partial backpropagation when MCTS runs in a distributed-memory parallel environment and reduce the communication overhead introduced by this phase. In [14,15], a table is appended to the messages exchanged by the processes. The table collects information on the forward path from node s to s', where the rollout commences. After the rollout, the table is updated at node s' home-process. This update only changes $N(s)$ and $w(s)$ for $\forall s \in path_{s \to s'}$, and information on their siblings remains the same. The updated table is sent to the $s'.parent$ home-process, where UCBvl of s' is inspected to see if it is still the greatest. In that case, the propagation stops, and a forward search begins. This recursive procedure can continue up to the root in the worst-case scenario.

Considering the above process, the advantage of using the table is that the detection in the backpropagation can even be accomplished in the child nodes instead of parent nodes since the table provides old information about the siblings' heuristics. However, it is possible to check if a node is yet the most promising node among its siblings without using an information table. This can happen at the node's parent, where the most recent information about the node and its siblings is available. This scenario is demonstrated in Fig. 6. There are two ongoing rollouts over the node S_2 subtree, one from node S_5 subtree and the other from

node S_7 subtree. The rewards achieved from both rollouts backpropagated to node S_2 home-process. Now, at node S_2, we can decide which of the nodes S_5 or S_7 has the highest UCBvl and launch a new forward search from it. In this scenario, we do not need an information table to perform the partial backpropagations. In addition, node S_2 has access to the most updated information of its children S_5, S_6, and S_7 to select a node. Consequently, the accuracy of the selection phase can be enhanced. However, this improvement comes at the expense of an additional backpropagation message. In applications with a significant number of children, particularly among nodes closer to the root, carrying an information table induces a higher communication overhead than sending one extra backpropagation message.

Our algorithm, PPB-MCTS, trades off one more backpropagation step to *remove the information table* and leverages the most updated UCBvl of the siblings. Removing the information table in large trees can reduce the communication overhead compared to MP-MCTS and TDS-MCTS. Hence, we anticipate conducting more rollouts for the same given computational budget. In general, more samples can improve the results in MCTS. However, as discussed in [36], the tree may grow from the nodes rewarded suboptimally as long as the optimal value remains unobserved. This issue depends on the problem structure, reward function, and non-stationarity of tree searches by MCTS. MP-MCTS can escalate this problem using the old sibling information.

Algorithm 1: Parallel Partial-Backpropagation-MCTS (PPB-MCTS).

```
   Input: nprocs,Time_Budget,root_rank,HASH
   Output: Best_Reward
 1 Function PPB-MCTS()
 2     if rank Is root_rank then
 3         Inserts 3 × nprocs initial MSearch messages in the queue;
 4     end
 5     while time ≤ Time_Budget do
 6         while message In receive buffer do
 7             Insert message in queue;
 8         end
 9         if queue Is Not empty then
10             Extract message;
11         end
12         if message.type Is MSearch then
13             Find node.key and node.proc using hash function;
14             if node.key In local trans-table then
15                 if node Is expanded then
16                     Select the child with the highest UCBvl;
17                     Update local trans-table, node parameter and send
                       MSearch message to child;
18                 else
19                     Expand node with new child;
20                     Update local trans-table, node parameters and send
                       MSearch message to child;
21                 end
22             else
23                 Add node to local trans − table;
24                 if rank Is root_rank then
25                     Expand node and send MSearch message to expanded
                       child;
26                 else
27                     Perform rollout and send MBackProp message reward to
                       node.parent;
28                 end
29             end
30         end
31         if message.type Is MBackProp then
32             Update node with message.reward;
33             if node Is expanded then
34                 Find the best child;
35                 if best child Is sender then
36                     Send a MSearch message;
37                 end
38             else
39                 Send MBackProp message reward to node.parent;
40             end
41         end
42     end
43 end
```

Our algorithm, PPB-MCTS, utilizes the most updated UCBvl values to encourage the exploration from the nodes according to the latest rewards a parent node receives. This can improve the accuracy of detecting where to initiate partial backpropagations. The pseudocode for PPB-MCTS algorithm is given in Algorithm 1. PPB-MCTS runs in parallel over a set of processes ranked from 0 to $nprocs-1$, where $nprocs$ is the number of processes. The communication between the processes is asynchronous. Each process has a queue to insert and extract messages it sends and receives. Processes communicate with each other using two types of messages, `MSearch` and `MBackProp`, for forward search and backpropagation, respectively (similar to [14,15]). Processes perform forward or backward searches depending on the message type extracted from the queue. The `MSearch` messages traverse the downward paths, while the `MBackProp` messages update the tree nodes along the upward path. A process stops executing when it reaches the algorithm's assigned time budget (*Time_Budget*). Each tree node corresponds to an entry in a transposition table (as described in Section 2.2.1). Each process keeps only the entries from the transposition table that correspond to the tree nodes it hosts. Tree nodes are represented by their state vector. A node's state vector encodes the environment state represented by the node. A hash function is shared among all processes. The hash function takes a tree node's state vector as input and returns its key in the transposition table and its home-process rank. Rollouts are performed locally in the process that hosts the tree node from which a rollout initiates.

The root home-process inserts a number of initial `MSearch` messages containing the root state into its queue. This number of messages depends on $nprocs$. Existing research [15] recommends setting this number to $3 \times nprocs$ to reduce processes' idle time (lines 2-3). A process extracts received messages from its buffer and inserts them into its queue. The process frequently scans its queue; when there is a message, it extracts it from the queue (lines 6-11). If the message type is `MSearch`, the process extracts the hash key of the message state using the hash function and searches the hash key in its transposition table. If the hash key is not in the table, the process adds it to its transposition table and launches a rollout. If the hash key is in the transposition table, depending on whether the corresponding tree node still has a child node to expand or all its children have been expanded, a node is expanded or selected, respectively, and its state is sent to its home-process as an `MSearch` message. The selection is performed according to the UCBvl given in Equation (2). The process finds the home-process using the hash function (lines 12-30).

Three scenarios exist if the message extracted is of type is `MBackProp`. If the message state belongs to a tree node that is not fully expanded, the process sends the message to the parent of the node. If the tree node is fully expanded, the UCBvl values for all child nodes are computed. If the UCBvl of the child node that sent the `MBackProp` message is the highest, then an `MSearch` message by the state of the child node is sent to the process of the child node. Otherwise, the parent node sends an `MBackProp` message to its parent (lines 31-41). PPB-MCTS terminates when the running time exceeds *Time_Budget* (lines 5 and 42).

In a search tree, a tree node may become reachable from multiple parent nodes. In this case, the tree turns into a DAG. Two methods can be used to handle such nodes. One method is to create two different identical nodes. However, since the hash function takes the state of a node as an input to generate its home-process and key in the transposition table, we need to add other information about the node to the hash function input to make it distinguishable from its replica. One option is to use the state of the node's parent because a set of duplicated nodes cannot have the same parent. Therefore, the transposition-table key for a node in the tree is

$$< \text{s.key, s.process\_id} >= \text{HASH.hash(s.state, s.parent.state)},$$

where HASH is the hash object shared among all processes to extract a tree node's transposition-table key, s.key, and its home-process id, s.process_id, using the method hash. The other approach is that the

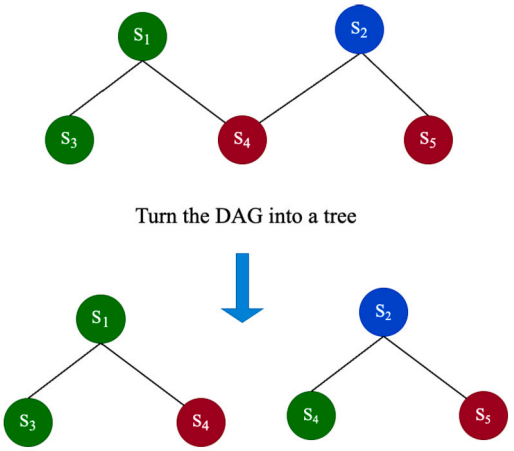


Fig. 7. Node S_4 has two parents. Two duplicate states of S_4 are created using its parents, each with a single parent and a local transposition table. Thus, the DAG turns into a tree.

child sends the same message to all its parents in the backpropagation phase. The problem with this approach is that a duplicated node is explored by one of its parents sooner than others. When one of the other parents wants to expand the node, the node is already in the local transposition table of its home-process. At this point, we need to decide whether to start another rollout from the node regardless of the fact that the node has been explored or consider the node's history and perform expansion or selection. The second approach may incur more communication. Nevertheless, parents can leverage the searches performed by the children in common. Here, we use the first approach and convert the DAG into a tree. This approach is shown in Fig. 7.

4. Using PPB-MCTS to solve JSSP

To assess the effectiveness of our algorithm, PPB-MCTS, we choose as benchmarks two NP-hard problems, the Job Shop Scheduling Problem (JSSP) and the Weighted Set Cover Problem (WSCP). In this section, we discuss JSSP followed by WSCP in the next section.

JSSP is a suitable choice for a benchmark because the number of potential actions (children) in the MCST search tree tends to grow exponentially in relation to the problem's size. This characteristic of the JSSP problem aligns well with the capabilities of PPB-MCTS. Particularly, the algorithm becomes valuable in scenarios where the presence of an information table can lead to significant communication costs. It is worth noting that the versatility of PPB-MCTS extends beyond the JSSP, as it can be applied to any problem where MCTS framework is suitable.

4.1. Job-shop scheduling problem (JSSP)

A standard JSSP instance comprises a set of jobs $\mathcal{J}$ and a set of machines $\mathcal{M}$. A job $j \in \mathcal{J}$ consists of $|j|$ tasks $\{O_i^j\}_{i=1}^{|j|}$ that must be completed in the order $O_1^j \to \cdots \to O_{|j|}^j$. Each task O_i^j is constrained to be executed on the machine $m_i^j \in \mathcal{M}$ for the processing time d_i^j. A machine can execute one job at a time. The number of machines is $|\mathcal{M}|$. The machine m_i^j specifies one of the machines m in the set $\mathcal{M}$ where the task O_i^j must be completed. Another task $O_{i'}^{j'}$ from another job can also have the same constraint to be executed on the same machine m. Given a JSSP instance, the objective is to find a non-preemptive schedule p with the minimum makespan C. A schedule $p \in \mathcal{P}$ determines each task's scheduled time S_i^j. The Makespan of a schedule is denoted by $C(p)$. The optimization problem for standard JSSP is

$$\begin{aligned} &\arg\min_{p\in\mathcal{P}} C(p) \\ \text{s.t.}\quad & C(p) \geq S_i^j + d_i^j && \forall j \in \mathcal{J},\ \forall i \in \{j\} \\ & S_{i+1}^j \geq S_i^j + d_i^j && i,\ i+1 \in \{j\} \\ & S_i^j \geq S_{i'}^{j'} + d_{i'}^{j'} \text{ or } S_{i'}^{j'} \geq S_i^j + d_i^j && \forall j \neq j' \in \mathcal{J},\ m_i^j = m_{i'}^{j'}. \end{aligned} \tag{3}$$

The first constraint states that the makespan cannot be less than the assigned scheduling times S_i^j plus their processing times d_i^j. The second constraint ensures that the tasks of a job are executed according to their priorities. The last constraint implies that a machine can not simultaneously run multiple tasks. Fig. 8 shows an instance of JSSP with three jobs and each job consists of three tasks.

4.2. Markov decision process (MDP)

To apply MCTS to JSSP, we must model it as an MDP. We consider a single-agent discrete MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma \rangle$ where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the set of actions, $\mathcal{R}$ is the reward function, $\mathcal{P}$ is the transition kernel of the environment, and γ is the discount factor. An agent interacts with the environment by taking action a^π at state s and transitions to state s' with probability $p(s'|s,a)$ and receives reward $R_a(s,s')$ under policy π. In deterministic MDPs, $p(s'|s,a) = 1$. The agent aims to learn a policy π^* that maps state s to an action a^π to maximize the value function [37],

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1}) \mid s_t = s \right]. \tag{4}$$

The value function of state s is the expected reward when starting in s and following policy π thereafter. Following an optimal policy π^*, the optimal value of state s is $V^{\pi^*}(s) = \sup_\pi V^\pi(s)$, $\forall s \in \mathcal{S}$.

The MDP for JSSP is deterministic, i.e., $p(s'|s,a) = 1$. Moreover, the agent is not rewarded prior to a terminal state where the rollout is completed. We also assume the discount factor $\gamma = 1$ since all actions have the same importance.

4.3. State representation

To provide an ordering for the tasks of a job j, all its tasks are stored in a stack according to their priority. This means that the task with the highest priority is at the top of the stack. As a result, there are $|\mathcal{J}|$ stacks. We use two vectors to represent a state in JSSP. The first vector contains information on machines and shows when machines become free after completing their assigned jobs. For example, the first element denotes the time instance that the machine 1 finishes the job it is running. This vector's length is $|\mathcal{M}|$. The second vector has information on the jobs and points to the tasks at the top of each stack. Each element in this vector consists of four parameters of a task $(m_i^j, d_i^j, i, S_i^j(t))$, where i is the priority in job j, and $S_i^j(t)$ is the potential scheduling time at decision time t. This variable is discussed in Section 4.6. Fig. 9 shows a JSSP example with three jobs and three machines. The state representation for stacks and machines at time t_k consists of two vectors. The first vector gives information about when the machines become available to accept new jobs. The second vector incorporates information on the tasks at the top of the stacks. The space complexity of this state representation is $\Theta(|\mathcal{M}| + |\mathcal{J}|)$.

4.4. Action representation

An action a_t at time t is a vector of size $|\mathcal{M}|$. The kth element in action a_t indicates the task assigned to the machines k at time t. The space complexity of an action vector is $\Theta(|\mathcal{M}|)$. An element in an action vector can be either `Null` or any of the tasks on the top of the job stacks whose $S_i^j(t) \leq t$, and its required machine m_i^j is not busy at time t. Fig. 9 shows a possible action among all valid actions that can be applied to the current states of stacks and machines. This action takes

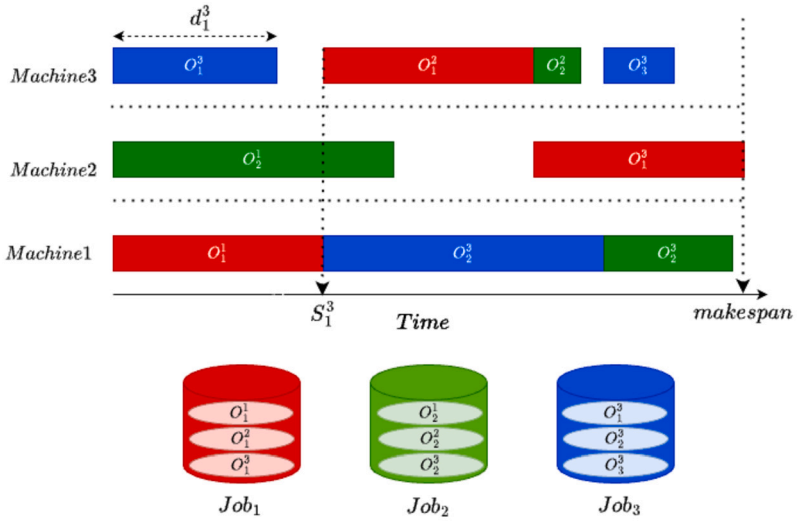


Fig. 8. An instance of JSSP with 3 jobs where each job consists of three tasks. Tasks in a job must be executed according to their priority and on their designated machines. For example task O_2^2 is the second task from the job 2 and its designated machine m_2^2 is machine 3. It runs for d_2^2 time units on machine 2. Task O_2^3 is allowed to run when O_2^2 execution is over.

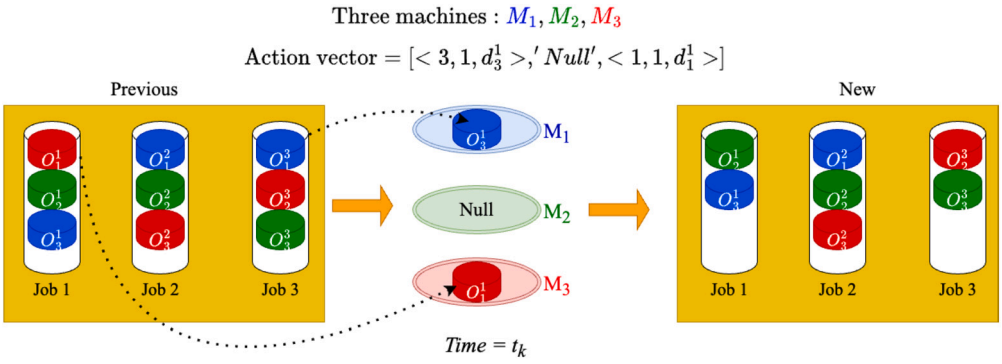


Fig. 9. By applying the action vector to the previous state of stacks, the tasks at the top of the stacks are popped off and assigned to their designated machine. Tasks are color-coded according to the machine they must be executed on. There are three jobs and three machines in this example.

the environment (machines and stacks) from the previous state to the new state. The `Null` operation at index k of a_t means that no task is assigned to machine k. This operation can be used when a machine is busy at time t or assigning another task from another job in some time instances greater than t can lead to a shorter makespan. If an element is not `Null`, it is denoted by a tuple (j, i, d_i^j). A task can be assigned to a machine m if $m_i^j = m$ and $S_i^j \geq t$ when $R_m(t) = 0$. Here, $R_m(t)$ is a binary indicator denoting whether machine m is unavailable, $R_m(t) = 1$, or free, $R_m(t) = 0$, at time t.

An all `Null` action vector is unacceptable in some conditions. For example, at the first decision time $t = 0$, it does not make sense to apply this action. Such actions are penalized with a reward zero, leading MCTS to avoid them. These conditions are discussed in Section 4.6.

4.5. Reward function

In MCTS, a higher reward is usually assumed to be better. However, in JSSP, we aim to find a sequence of actions to minimize makespan. In order to address this issue, each rollout's makespan is subtracted from an upper bound UB on the makespan of the given JSSP instance. Here, we use a simple upper bound defined as

$$UB = \sum_{j=1}^{|J|} \sum_{i=1}^{|j|} d_i^j. \tag{5}$$

It was shown in [14] that for problems where the reward range is $r \in [0,1]$, the hyperparameter c in Equation (2) is usually set to $\sqrt{2}$. This hyperparameter determines the trade-off between exploration and exploitation in MCTS. Higher c values imply more exploration in MCTS. To map the reward in different JSSP instances into the range [0,1], we need to normalize the reward. We define a lower bound LB on makespan as

$$LB = \max_{\forall j \in \mathcal{J}} \left\{ \sum_{i=1}^{|j|} d_i^j \right\}. \tag{6}$$

Now, the reward of a rollout i is defined as

$$r^i = \frac{UB - rollout_i.makespan}{UB - LB}. \tag{7}$$

This reward function maps a rollout's makespan into the range [0,1). The reward function cannot assign reward one to any makespan, including the actual minimum makespan. This is due to the gap between the lower bound (LB) on the minimum makespan and its actual value. As

the distance between these two becomes smaller, it is expected that the MCTS performs better.

4.6. Action-state interaction

When an action a_t is applied to a state s_t, we need to update the $S_i^j(t)$ of the new tasks on top of the job stacks. In addition, the next decision time t' for taking the following action $a_{t'}$ from the current state s_t must be determined.

The potential scheduled time for task i in job j at time t, $S_i^j(t)$, must meet all constraints in (3) if substituted with S_i^j. However, the potential scheduled time $S_i^j(t)$ is different from S_i^j. It indicates the scheduled time for task O_i^j at time t whereas S_i^j is the actual scheduled time for the task. A task can be assigned to a machine when $S_i^j(t) \leq t$. More than one task can satisfy this condition at time t. The action vector at time t can choose only one of these tasks and assign it to machine m_i^j. Hence, $S_i^j(t)$ is a function of t and can be different from the actual time instance S_i^j that is the time instance when task O_i^j is assigned to machine m_i^j. For example, as shown in Fig. 9, $S_1^1(t_k) = S_1^1$ for O_1^1 but $S_1^2(t_k) \neq S_1^2$ for O_1^2.

Depending on the action vector, the update rule for the potential scheduled time at the decision time t_k is

$$S_i^j(t_k) = \max\{(t_{k-1} + d_{i+1}^j) I_{i+1,j}^{t_{k-1}}, (t_{k-1} + d_{i'}^{j'}) I_{i',j'}^{t_{k-1}}, S_i^j(t_{k-1})\}, \tag{8}$$

where $m_i^j = m_{i'}^{j'}$, $I_{i,j}^{t_k}$ is a binary indicator denoting whether O_i^j is assigned to machine m_i^j at decision time t_k, and t_{k-1} is the previous decision time. The $S_i^j(t)$ for all tasks on top of the job stacks are updated at each decision time t.

Enumerating all valid actions at each time unit is impractical since it can lead to a deep search tree with many nodes having merely one single edge of the all `Null` action vector. As a result, we need to find an efficient approach to decide the next decision time with respect to the action applied at the current time t_{k-1}. This procedure for finding the next decision time is described in Algorithm 2. The first rule for determining the next decision time is $t_{k-1} < t_k$. The minimum $S_j^j(t_k)$ among the tasks ready to be popped off from the job stacks is a reasonable choice for determining the next decision time. However, since we allow for `Null` actions in the action vector, the minimum $S_i^j(t_k)$ may remain unchanged at time t_k, and it can break the rule. Hence, we must select the next minimum $S_i^j(t_k)$. First, we sort the job stacks by $S_i^j(t_k)$ in a non-decreasing order (line 2). Then, we begin comparing t_{k-1} with each $S_i^j(t_k)$ according to the sorted job stacks. This process continues to find the first $S_i^j(t_k)$ that is greater than t_{k-1} (lines 4-9). If every $S_i^j(t_k)$ is less than t_{k-1}, then we can conclude that the action we have chosen is unacceptable (lines 10-12). This outcome leads to the termination of the forward search at decision time t_{k-1}, accompanied by the reward UB. This happens by setting the next decision time to UB since we do not allow for a search led to a makespan exceeding UB.

Algorithm 2: Finding Next Decision Time.

```
   Input: Job_Stacks, Current_Time
   Output: Next time decision
1  Function Next_Time()
2      Sorted_Job_stacks = Sort(Job_Stacks);
3      next_time = Current_Time;
4      for S_i^j In Sorted_Job_stacks do
5          if S_i^j > next_time then
6              next_time = S_i^j;
7              Break
8          end
9      end
10     if next_time = Current_Time then
11         next_time = UB ;        /* This signals MCTS to finish the
               rollout */
12     end
13 end
```

5. Using PPB-MCTS to solve WSCP

Another problem that we choose to evaluate our algorithm with is the Weighted Set Cover problem (WSCP). Similar to JSSP, WSCP is also an NP-hard combinatorial optimization problem. Depending on how elements of the universe are distributed among the subsets, weights assigned to the subsets, and the number of subsets in the problem, the quality of rollouts in finding (sub)optimal solutions can become important.

The number of subsets is important because compared to MP-MCTS, PPB-MCTS does not carry an information table. Hence, we expect that PPB-MCTS completes more rollouts and exhibits better performance on average. Another crucial factor determining the difficulty of WSCP lies in the distribution of elements and weights across subsets. For instance, two trivial cases emerge: (i) subsets are disjoint, resulting in identical rewards for each rollout irrespective of the assigned weights, and (ii) each subset covers the entire universe, resulting in a search tree with a depth of one and the optimal solution is the subset with the least weight. An illustrative case of element distribution where partial rollouts prove beneficial occurs when certain subsets encompass others and their weights are less than the accumulated weights of the subsets they subsume. In such cases, growing the search tree from subtrees rooted in these subsets and executing partial backpropagation holds promise. In Section 6, we elaborate on the WSCP instances we design for evaluating the sequential and parallel MCTS algorithms.

5.1. Weighted set cover problem (WSCP)

In the WSCP [38], we are given a universal set $\mathcal{U} = \{e_i\}_{i=1}^m$ consisting of m elements and a collection of n subsets $S = \{S_i\}_{i=1}^n$ where each $S_i \subseteq \mathcal{U}$. Each subset is assigned a nonnegative weight w_i. The goal is to find a minimum weight group of subsets from S such that their union covers all elements in $\mathcal{U}$. WSCP can be formulated as follows:

$$\begin{aligned} &\text{minimize} \sum_{i=1}^{n} w_i x_i \\ \text{s.t.} \sum_{i:e_j \in S_i} x_i \geq 1 \quad &\forall e_j \in \mathcal{U}, \\ x_i \in \{0,1\} \quad &\forall i \in \{1,\ldots,|n|\}, \end{aligned} \tag{9}$$

where x_i is an indicator variable equal to one when the subset S_i is included in the solution, and zero otherwise. The first constraint guarantees that the solution incorporates at least one subset for every element in the universe. The second constraint guarantees that each subset is either included in or excluded from the solution. This implies that selecting a partial group of elements from a subset is prohibited.

When all subsets have equal weights, the WSCP reduces to selecting the minimum number of subsets necessary to cover all elements in $\mathcal{U}$. This problem is known as the Set Cover problem (SCP).

5.2. MDP for WSCP

Like in JSSP, we model WSCP as a deterministic MDP, where transitions between states yield reward zero unless the terminal state is reached. Furthermore, we maintain a discount factor of $\gamma = 1$ similar to JSSP.

5.2.1. State representation

Each state in WSCP is a bipartite graph with the first bipartition composed of m vertices of type *element* and the second bipartition composed of n vertices of type *subset*. An edge between an element vertex and a subset vertex exists if the element belongs to the subset. We use an adjacency list to represent the bipartite graph which requires $O(nm)$ space.

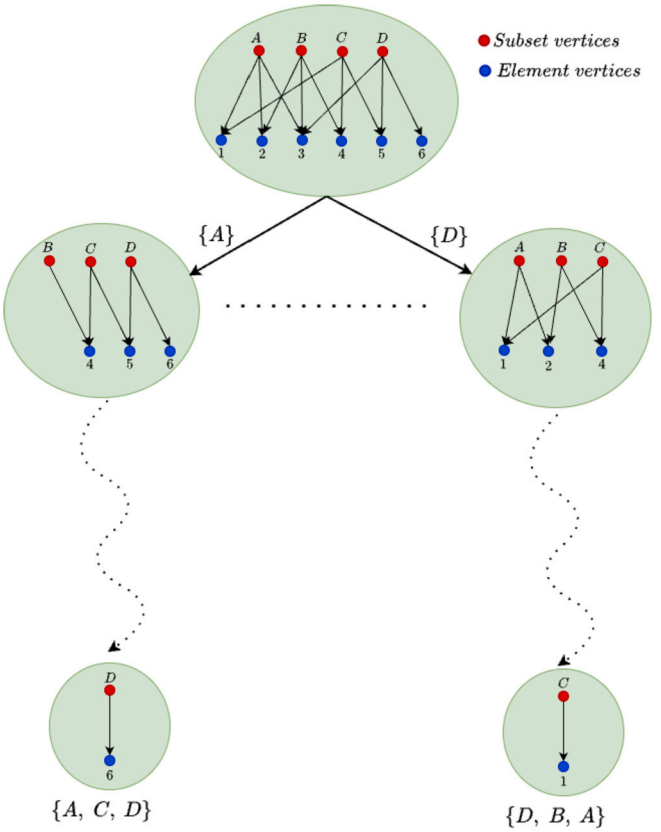


Fig. 10. The state of the root is represented by a bipartite graph consisting of four subset vertices and 6 element vertices. If we select subset A, the corresponding subset vertex A, along with all adjacent edges and element vertices {1, 2, 3}, are removed from the bipartite graph. Conversely, choosing subset vertex D leads to the elimination of element vertices {3, 5, 6} from the bipartite graph.

5.2.2. Action representation

To select an action at a tree node, we choose a vertex from the subset vertices available in the current state of the tree node. Subsequently, we eliminate the selected subset vertex, along with all edges connected to it, and the corresponding element vertices attached to those edges. We reach a terminal state when all element vertices have been removed from the bipartite graph. The selection of vertices at each tree node is contingent upon whether it occurs during the selection phase or the rollout phase. During the selection phase, we adhere to UCB or UCBvl rules, depending on whether we are employing sequential MCTS or parallel MCTS, respectively. In the rollout phase, we randomly sample from the subset vertices available at the tree node. Fig. 10 shows an example of how selecting an action at the root state can transform the bipartite graph, consequently rendering a new child state from the root.

5.2.3. Reward function

In WSCP, the objective is to select a set of subsets that collectively achieve the lowest possible total weight, while ensuring the coverage of all elements within the universe. Hence, similar to JSSP, we establish a lower bound and an upper bound and follow the same structure to keep the rewards within the range [0,1]. The trivial upper bound is

$$UB = \sum_{i=1}^{n} w_i. \tag{10}$$

A trivial lower bound can also be expressed as

$$LB = \min_{i\in\{1,\dots,n\}} w_i. \tag{11}$$

If a search[2] from the initial state to the terminal state is composed of k subsets $\{S^1, \dots, S^k\}$, then the total weight for that search is equal to $\sum_{i=k}^{j} w^j$. The superscript j shows the order of selection. This means each S^j can be one of the subsets in S not chosen up to step j.

The reward function for a search composed of $\{S^1, \dots, S^k\}$ is given by:

$$r^{\{S^1,\dots,S^k\}} = \frac{UB - \sum_{i=1}^{k} w^i}{UB - LB}. \tag{12}$$

The reward can be zero only if subsets are mutually exclusive and one if the subset with the greatest weight covers all elements in the universe.

6. Experimental results

We evaluate our algorithm, PPB-MCTS, by comparing it against two existing tree-parallel based MTCS algorithms, TDS-MCTS [14] and MP-MCTS [15] on several JSSP and WSCP instances. The parallel algorithms are implemented using the MPI library for Python (mpi4py) and executed on a cluster of computational nodes. Each cluster node consists of 2 AMD 74F3 3.2 GHz CPUs. Each CPU consists of 24 cores. A process is allocated to one CPU core with a limit of 16 processes on each CPU. For example, when utilizing 256 processes for a run, we allocate 8 cluster nodes, each with 2 CPUs and 16 cores per CPU (out of 24 cores per CPU). Thus, we assign one process per core.

To compare the performance of the parallel algorithms, we consider four metrics: *average number of rollouts*, *normalized number of backpropagation messages received*, *average of the best rollouts across multiple runs*, and the *distribution of the rollouts*.

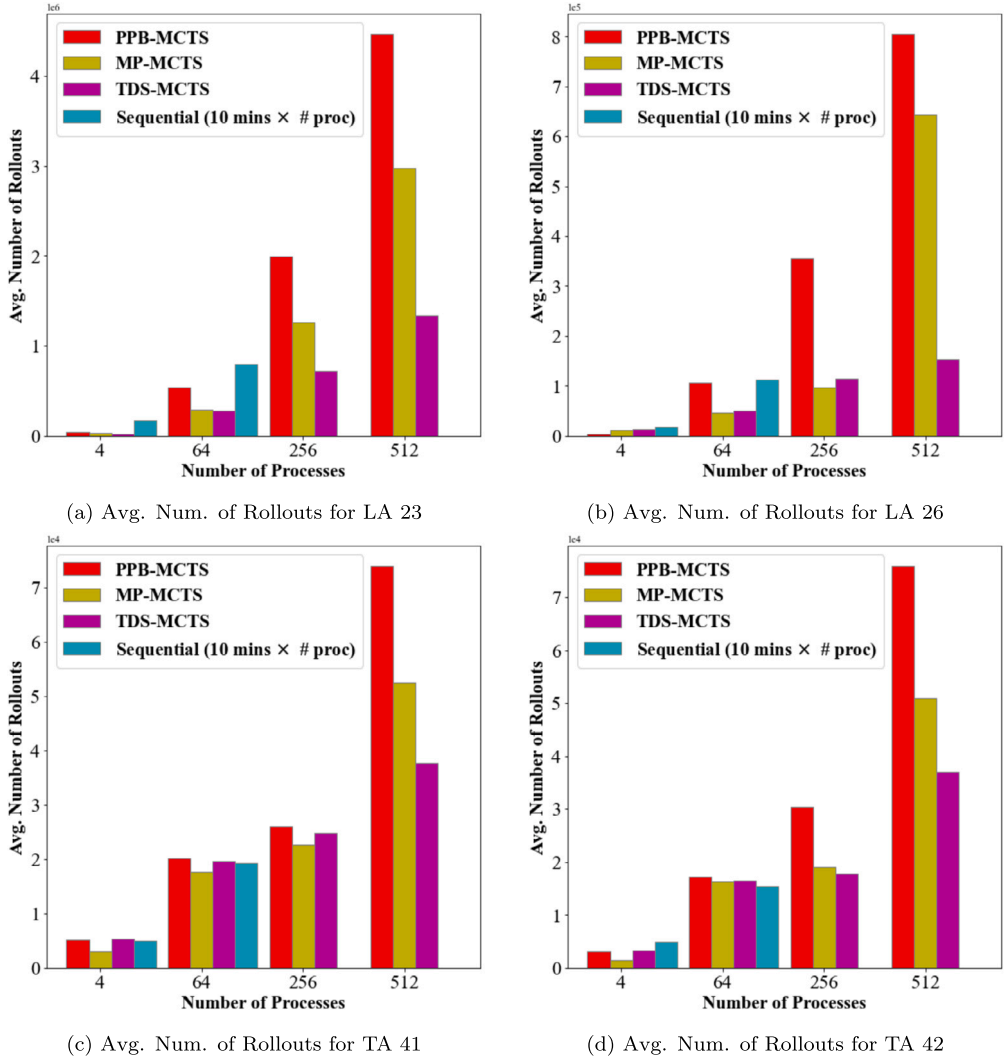
The first metric represents the number of rollouts conducted by each algorithm. Conducting more rollouts can enhance the performance of MCTS and may yield superior results. The second metric gives the number of backpropagation messages received by each process, thus, measuring the load balance among processes in backward paths. To enable the comparison across the three algorithms, we normalized this metric, accounting for potential disparities in the absolute number of `MBackProp` messages under different experimental configurations. The *average of the best rollouts across multiple runs* measures the capability of an algorithm to expand a search tree for uncovering better (sub)optimal solutions. In our study, we execute each algorithm three times and select the best rollout from each run, then calculate the average of these best rollouts. The final metric indicates the quality of rollouts performed by an algorithm, as demonstrated by the distribution of these rollouts. An algorithm with superior rollout quality is anticipated to display a right-skewed distribution, suggesting that most rollouts are in the proximity of the best rollout. It is important to note that due to the asynchronous nature of all these parallel algorithms, comparing the quality of rollouts over time is not feasible.

Reported results for the first three metrics are the average across three trials, each allocated a 10-minute *runtime budget*. In the last metric, each distribution characterizes one run executed for 10 minutes. For comparison fairness, the sequential MCTS was given a 10-minute runtime per process used in the parallel MCTS. This means, for example, a 40-minute runtime for sequential MCTS when compared to a four-process parallel MCTS. We did not experiment with the sequential algorithm for 2560 and 5120 minutes since the results are reported by averaging over three runs, and the running time becomes prohibitively large.

6.1. JSSP results

As the first benchmark, we use JSSP (modeled as an MDP, presented in Section 4). Four JSSP instances, LA 23 (15 × 10), LA 26 (20×10) [39],

[2] Here, we use the term search to refer to both the selection and rollout phases.



(a) Avg. Num. of Rollouts for LA 23

(b) Avg. Num. of Rollouts for LA 26

(c) Avg. Num. of Rollouts for TA 41

(d) Avg. Num. of Rollouts for TA 42

Fig. 11. Average number of rollouts sampled in four JSSP instances for each algorithm. PPB-MCTS shows better scalability compared to the other MCTS algorithms. The averages presented stem from three trials, each constrained by a 10-minute time limit.

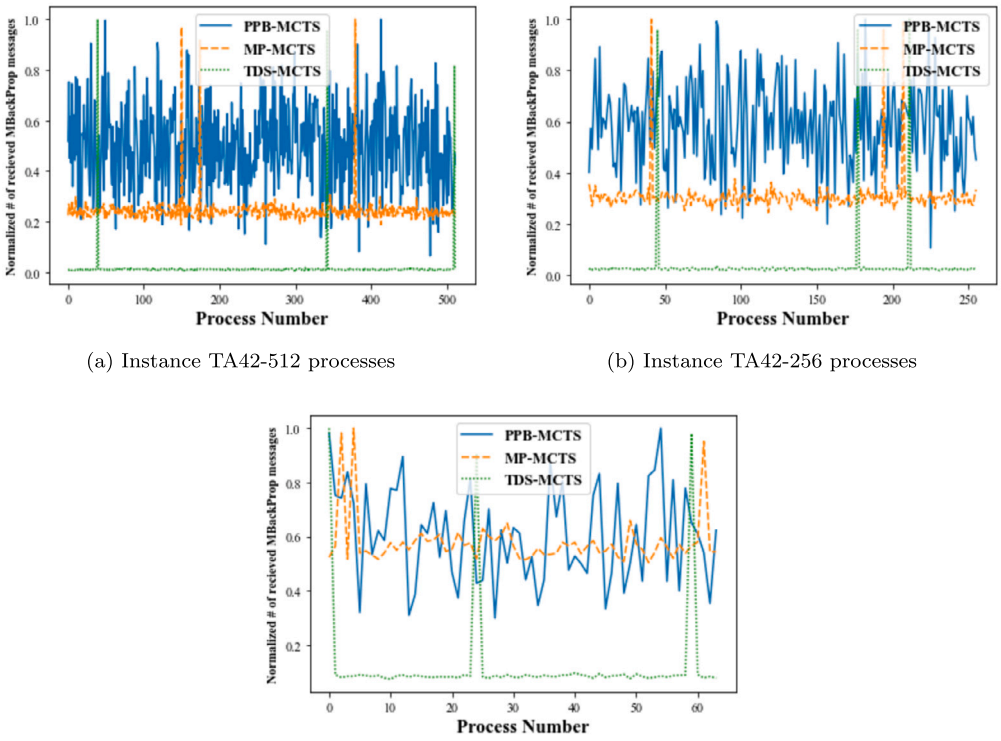
TA 41 (30×20), and TA 42 (30×20) [40], are selected to investigate the performance of the algorithms. Here, the tuple after each instance name $(x, y) = (|\mathcal{J}|, |\mathcal{M}|)$.

In addition, four baseline methods were picked to assess the PPB-MCTS results. We picked two priority-based methods, the longest remaining time machine (LRM) and the longest remaining processing time job (LRPT), and two machine learning methods. The first machine learning method [41] employs a combination of deep graph neural network (DGNN) and deep reinforcement learning (DRL) to uncover nearly optimal policies and approximate the makespan for JSSP. The second method [42] utilizes the dueling double deep queue network (DDDQN) to discover suboptimal policies and determine the makespan for JSSP.

As discussed in Section 2, the transposition-table technique provides load-balance in the forward paths. However, the load imposed by `MBackProp` messages is also important when assessing the load-balance in distributed-memory MCTS algorithms. In TDS-MCTS, the root home-process receives a `MBackProp` message for each rollout due to the complete backpropagations. This incurs a significant load-imbalance over the home-processes of the root and a group of its child nodes that are selected more than others. Moreover, we also compared the algorithms with respect to the *average minimum makespan* to show that PPB-MCTS improves over the previous distributed-memory MCTS algorithm for problems with many actions at tree nodes.

The number of conducted rollouts by each algorithm is shown in Fig. 11. The results show that PPB-MCTS and MP-MCTS accomplished more rollouts for 512 and 256 processes when compared to TDS-MCTS in all four JSSP instances. This is due to the partial backpropagation they perform instead of a complete backpropagation. Moreover, as the number of processes increases, PPB-MCTS also outperformed MP-MCTS for this metric. This is because there is no need to carry an information table while backpropagations are still partial. When 512 processes were employed, PPB-MCTS conducted almost 1.5 times more rollouts than MP-MCTS for all JSSP instances. This pattern can also be observed for instances TA 42, LA 23, and LA 26, for 256 processes. The number of rollouts executed by PPB-MCTS is still slightly higher than those completed MP-MCTS for instance TA 42. For 4 and 64 processes, all three algorithms and the sequential MCTS showed almost the same performance. One reason can be the height of the search tree. Each forward search extends a new tree node to the search tree. If the number of conducted rollouts (forward searches) in those settings is insufficient, then the search tree is not tall, and partial backpropagations are like complete backpropagations.

We used the normalized number of `MBackProp` messages received by each process to compare the load balancing achieved by the algorithms. We normalized the `MBackProp` messages each process obtained by the highest number of `MBackProp` messages received by a process. Fig. 12 shows that the distribution of `MBackProp` messages accepted



(a) Instance TA42-512 processes

(b) Instance TA42-256 processes

(c) Instance TA42-64 processes

Fig. 12. JSSP: Backpropagation messages received by each process are more evenly distributed in PPB-MCTS than TDS-MCTS and MP-MCTS. This becomes more prominent as the number of processes increases.

by a process in PPB-MCTS is more even compared to TDS-MCTS and MP-MCTS. This is because PPB-MCTS uses the most updated information for partial backpropagation, which encourages more exploration of the promising branches in the search tree.

Due to its higher number of rollouts, PPB-MCTS exhibits a greater absolute number of `MBackProp` messages compared to TDS-MCTS and MP-MCTS. However, it is crucial to highlight that, unlike the other algorithms, the message payload in PPB-MCTS does not include an information table. Furthermore, all three distributed-memory tree-parallelized algorithms operate asynchronously, sending new messages as soon as they are available, without waiting for corresponding `MBackProp` messages. Consequently, the increased message exchange in PPB-MCTS does not result in memory or communication bottlenecks, given equivalent resource allocation across all algorithms.

The average minimum makespans obtained by the algorithms for the selected instances, LA 23 (15×10), LA 26 (20×10), TA 41 (30×20), and TA 42 (30×20), are reported in Tables 4, 5, 6, and 7, respectively. The superior performance of PPB-MCTS on the search conducted for finding the smallest makespan can be observed in almost all settings. As an example in Table 6, when 512 processes are employed, the smallest makespan discovered by PPB-MCTS is almost 13 time units and 21 time units less in comparison to the smallest makespans found by MP-MCTS and TDS-MCTS, respectively, on average for TA 41. However, when the number of processes is 4 and 64, similar to the results in Fig. 12, the average smallest makespan by PPB-MCTS is narrowly less than those from other algorithms. It can be inferred that when more rollouts are conducted, the makespans tend to be smaller. Another important factor is to take advantage of the new information about the tree nodes to perform the search tree efficiently. PPB-MCTS provides both fast and efficient MCTS searches.

Increasing the number of rollouts in asymptotic scenarios tends to yield solutions closer to the optimal solution. In sequential MCTS, there exists an $O(\log n)$ bound [1] for identifying the best action from the root, where n denotes the number of rollouts. However, in the context of finite MCTS and the pursuit of optimal solutions through a sequence of actions from the root to a terminal state, no definitive bound has been established, to the best of our knowledge.

Nonetheless, our experiments support the general trend that conducting more rollouts often leads to improved solutions. A notable challenge in finite MCTS involves navigating search trees, favoring the growth of promising subtrees while neglecting exploration in other po-

Table 4
Average makespan: JSSP instance LA23.

Algorithm \ # Processes	4	64	256	512
PPB-MCTS	**1063.6**	1058.3	**1046.6**	**1034.3**
MP-MCTS[15]	1078.3	1061.3	1061.3	1036.3
TDS-MCTS[14]	1080.6	1068.6	1059.6	1057.3
Sequential	1069.6	**1037.3**	-	-

Table 5
Average makespan: JSSP instance LA26.

Algorithm \ # Processes	4	64	256	512
PPB-MCTS	1354.6	1308.3	**1288**	**1271.6**
MP-MCTS[15]	1356.3	**1303.3**	1293.6	1279.3
TDS-MCTS[14]	1353.6	1305.6	1301.6	1288
Sequential	**1350.6**	1312	-	-

Table 6
Average makespan: JSSP instance TA 41.

Algorithm \ # Processes	4	64	256	512
PPB-MCTS	**2454.3**	2402.6	**2399.3**	**2376.6**
MP-MCTS[15]	2456.3	2439.6	2408.3	2390.3
TDS-MCTS[14]	2455.6	2445.3	2415	2398
Sequential	2455.6	**2391.6**	-	-

Table 7
Average makespan: JSSP instance TA 42.

Algorithm \ # Processes	4	64	256	512
PPB-MCTS	2434.3	2354	**2337.6**	**2323.6**
MP-MCTS [15]	2423.3	2366.3	2352.6	2331.6
TDS-MCTS [14]	2394.6	2384.3	2343.6	2336.3
Sequential	**2374**.6	**2348**	-	-

Table 8
JSSP Average makespan: Comparison with other baseline methods.

Method \ Instance	LA23	LA26	TA41	TA42
PPB-MCTS (512)	**1032**	**1268**	**2375**	**2313**
LRM	1214	1498	2679	2701
LRPT	1258	1439	2605	2784
DGNN/DRL [41]	-	-	2667	2664
DDDQN [42]	1053	1327	2450	2351

tentially fruitful areas. This issue is explored in detail in [36]. One plausible explanation for achieving superior solutions with fewer rollouts lies in the stochastic nature of random sampling during the rollout phase, which may facilitate the growth of the search tree from highly promising branches. An example that performing more rollouts does not necessarily lead to better solutions is when we consider the average makespan of the sequential MCTS and that of PPB-MCTS for the case with four processes. However, our results indicate that solutions closer to the optimal solution are typically attained by increasing the number of rollouts.

We also compared the best makespan found by PPB-MCTS over 512 processes with four baseline methods. Table 8 shows that the best makespan obtained by our algorithm PPB-MCTS is smaller than those attained by other methods. This is more noticeable in TA 41 and TA 42, where the problem size is larger. This suggests that as the problem size increases, we expect that PPB-MCTS achieves better results.

To assess the overall pattern of the rewards earned during the execution of any of the tree-parallelized distributed memory algorithms, we analyzed the distribution of rewards. Given the asynchronous operation of these algorithms, depicting the temporal behavior of rollouts might pose challenges, as multiple rollouts can occur simultaneously across various processes. To illustrate the distribution of rollouts, we consider a single trial. Furthermore, we convert distributions into probability distribution functions to standardize the scales of distributions from different algorithms, especially when the number of rollouts conducted by each algorithm varies.

The distributions of rollouts for PPB-MCTS, MP-MCTS, and TDS-MCTS under a 10-minute time constraint and 512 allocated processes on TA 42 and LA 26 are depicted in Fig. 13.

An observable trend is the slightly more pronounced right-skewness in the mass distribution of PPB-MCTS compared to TDS-MCTS and MP-MCTS for LA 26. However, this trend is negligible for TA 42. This characteristic likely stems from the efficient exploration strategy of PPB-MCTS, which prioritizes the most current information during selection. This leads to a focused search tree, growing primarily from subtrees with finer makespan potential and directing rollouts toward the most promising solutions. Consequently, we notice that the majority of makespans are clustered around the vicinity of the minimum makespan found through rollouts.

6.2. WSCP results

To further evaluate the performance of PPB-MCTS, we selected the weighted set cover problem (WSCP), another NP-hard combinatorial optimization problem. As detailed in Section 5.2.1, our approach models WSCP using a bipartite graph. In this model, vertices on one bipartition represent subsets, while vertices on the other bipartition correspond to elements in the universe. For our experimental setup, the bipartite graph $\mathcal{G}$ is generated using the Erdős-Rényi model [43]. The bipartite graph is denoted by $\mathcal{B}(\mathcal{S}, \mathcal{U}, p^{\mathcal{G}})$, where $\mathcal{S}$ represents the set of subsets (i.e., vertices in the first bipartition), $\mathcal{U}$ is the universe (i.e., vertices in the second bipartition), and $p^{\mathcal{G}}$ is the probability that an element belongs to a subset (i.e., the probability of having an edge between two vertices from the two bipartitions).

Table 9
WSCP problem instances.

Instances	$p^{\mathcal{G}}$	$\lvert\mathcal{S}\rvert$	$\lvert\mathcal{U}\rvert$	Avg. Weight	Avg. Subset Size	α
$\mathcal{B}_1$	$\frac{2^8}{2^{16}}$	2^{16}	2^8	2.08	260.94	10^3
$\mathcal{B}_2$	$\frac{2^9}{2^{17}}$	2^{17}	2^9	1.88	514.7	10^3
$\mathcal{B}_3$	$\frac{2^{10}}{2^{18}}$	2^{18}	2^{10}	0.97	1025.7	10^3

Table 10
Average of minimum total weight: WSCP instance $\mathcal{B}_1$.

Algorithm \ # Processes	4	64	256	512
PPB-MCTS	**789.84**	**681.66**	**614.23**	**544.35**
MP-MCTS[15]	809.17	699.84	630.19	602.03
TDS-MCTS[14]	822.68	716.946	672.54	640.11
Sequential	797.77	685.76	-	-

Table 11
Average of minimum total weight: WSCP instance $\mathcal{B}_2$.

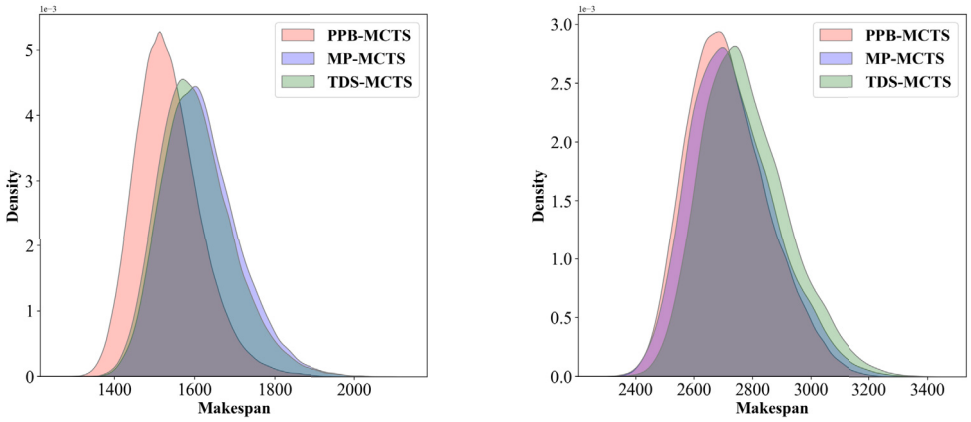
Algorithm \ # Processes	4	64	256	512
PPB-MCTS	694.19	**648.53**	**572.875**	**501.72**
MP-MCTS[15]	698.23	670.82	627.18	559.65
TDS-MCTS[14]	700.41	665.5	651.80	586.15
Sequential	**691.58**	662.22	-	-

Our experiments utilized three Erdős-Rényi bipartite graphs with the parameter $p^{\mathcal{G}}$ set to $\frac{|\mathcal{S}|}{|\mathcal{U}|}$ to generate subsets, where $|\mathcal{S}| < |\mathcal{U}|$. This choice promotes longer search paths by avoiding large subsets, which can terminate the search paths in the vicinity of the root. Longer paths ensure sufficient depth in the search tree, making it suitable for employing partial backpropagations and allowing exploration of subtrees with promising results rather than consistently propagating rewards back to the root which is the case in the short search trees.

Weights were assigned to each subset $s_i \in \mathcal{S}$ with size m_i. Specifically, weight w_i was drawn from an exponential distribution exp(m_i with parameter m_i. This approach promotes the assignment of lower weights to larger subsets. To prevent issues with very small numerical values, each weight w_i was further scaled by a factor of α. Table 9 provides a summary of the three instances employed in our experiments. The average weights and average subset sizes are truncated to two decimal points in this table.

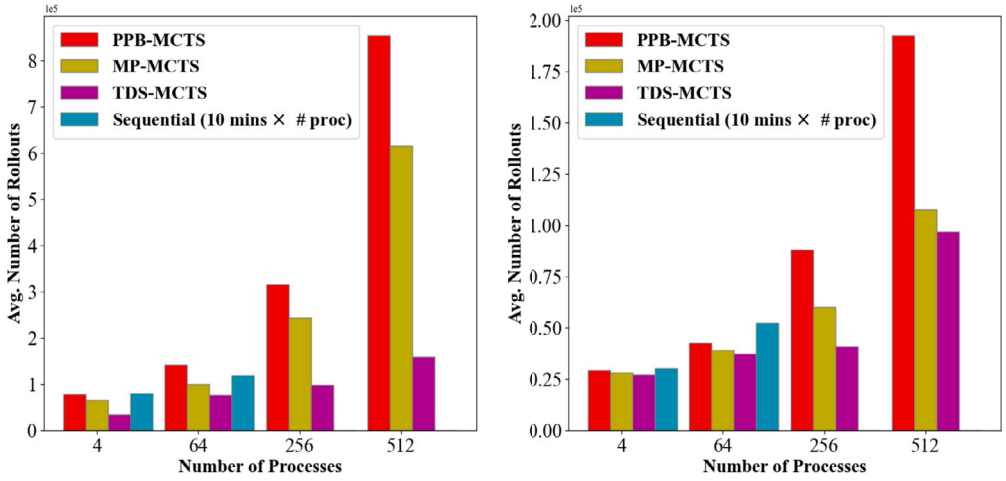
Following the approach in the JSSP experiments, we first compare PPB-MCTS with TDS-MCTS, MP-MCTS, and sequential MCTS using the metric of *average number of rollouts*. Fig. 14 presents the average number of completed rollouts for each algorithm over three 10-minute runs. For the sequential MCTS, we allowed the sequential algorithm to run for 10 minutes multiplied by the number of processes provided to the parallel algorithm. However, we did not conduct the experiments over sequential MCTS for 256×10 minutes and 512×10 minutes due to the large time limit they need.

Similar to JSSP, PPB-MCTS demonstrates superior scalability with an increase in the number of processes. This becomes particularly evident when comparing $\mathcal{B}_1$ with $\mathcal{B}_2$, where both the number of subsets and the universe size double. Increasing the number of subsets is analogous to boosting the branching factor, consequently resulting in larger information tables for MP-MCTS to carry.

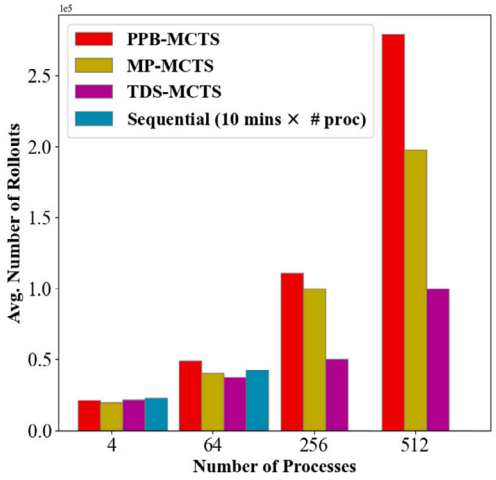


(a) Makespan Distribution for LA 26 (b) Makespan Distribution for TA 42

Fig. 13. The distribution of makespans achieved for PPB-MCTS, MP-MCTS, and TDS-MCTS over TA 42 and LA 26 JSSP instances when 512 processes utilized. We use density distribution to have a standardized scale for all three algorithms.



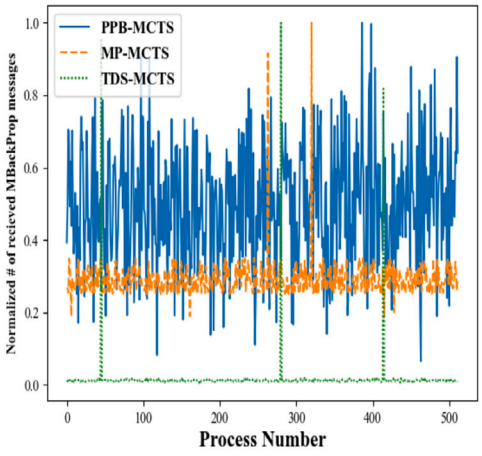
(a) Avg. Num. of Rollouts for $\mathcal{B}_1$ (b) Avg. Num. of Rollouts for $\mathcal{B}_2$



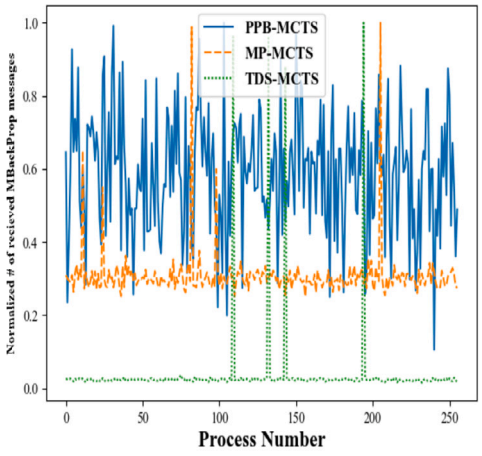
(c) Avg. Num. of Rollouts for $\mathcal{B}_3$

Fig. 14. The average number of rollouts sampled across three WSCP instances for each algorithm. PPB-MCTS demonstrates superior scalability in contrast to the other MCTS algorithms. These averages are derived from three trials, each with a 10-minute time limit.

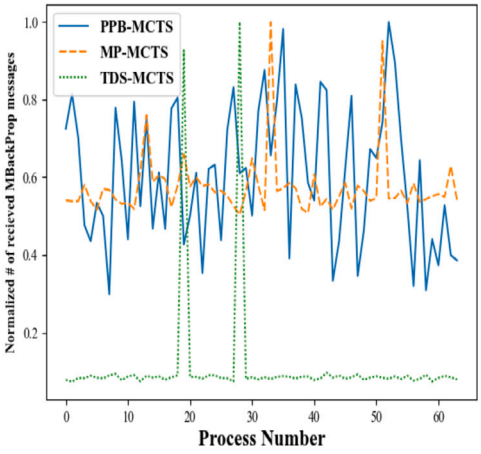
Fig. 15 shows the normalized distribution of backpropagation messages, denoted as `MBackProp`, for instance $\mathcal{B}_2$ across 512, 256, and 64 processes. Analogous to JSSP, it is evident that as the number of processes increases, the distribution of received backpropagation messages



(a) Instance $\mathcal{B}_2$-512 processes



(b) Instance $\mathcal{B}_2$-256 processes



(c) Instance $\mathcal{B}_2$-64 processes

Fig. 15. WSCP: Compared to TDS-MCTS and MP-MCTS, PPB-MCTS exhibits a more balanced distribution of backpropagation messages `MBackProp` across its processes, especially as the number of processes grows.

Table 12
Average of minimum total weight: WSCP instance $\mathcal{B}_3$.

Algorithm \ # Processes	4	64	256	512
PPB-MCTS	382.87	**339.02**	**387.44**	**257.90**
MP-MCTS[15]	385.22	349.21	215.21	320.66
TDS-MCTS[14]	376.91	350.61	322.49	311.15
Sequential	**375.70**	346.94	-	-

Table 13
WSCP average minimum total weight: Comparison with the baseline method.

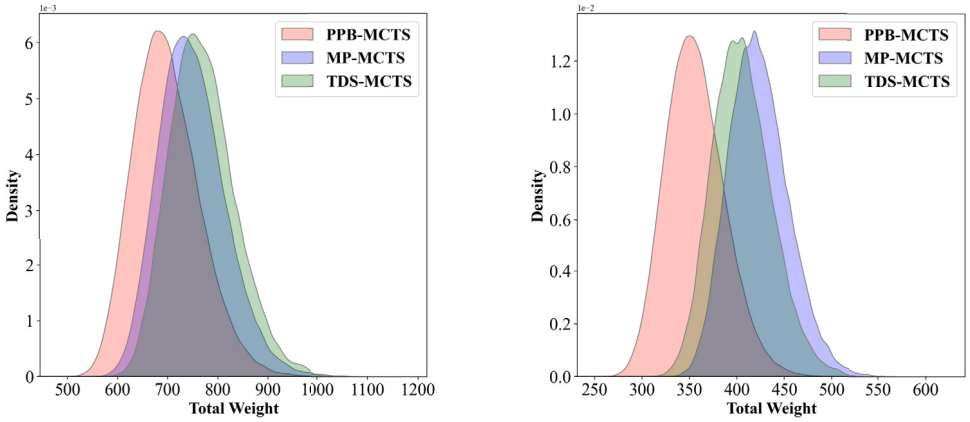
Method \ Instance	$\mathcal{B}_1$	$\mathcal{B}_2$	$\mathcal{B}_3$
PPB-MCTS (512)	**543.93**	**500.55**	**255.09**
Randomized Rounding	608.65	560.76	319.03

across processes in PPB-MCTS demonstrates greater uniformity compared to the patterns observed in MP-MCTS and TDS-MCTS.

Tables 10, 11, and 12 present the average minimum weight achieved by each algorithm over three 10-minute runs. PPB-MCTS consistently outperforms the other algorithms, except in the four-process scenario where the sequential MCTS exhibits a slight advantage.

We also compared the performance of PPB-MCTS over 512 processes with a baseline approximation algorithm for WSCP. We selected the Randomized Rounding algorithm by [38] as our baseline. This approach leverages linear programming to approximate solutions for the WSCP. Unlike the original formulation where x_i in Equation (9) can only be 0 or 1, the Randomized Rounding algorithm allows x_i to take any fractional value within the range $(0, 1)$. Subsequently, subsets are sampled according to these fractional values of x_i, leading to an approximate solution. To ensure complete solutions, we incorporated an additional sampling step for cases where the algorithm cannot find a full solution initially. In the additional sampling step, we first sum the x_i values of the remaining subsets from the initial sampling stage. Then, we normalize their x_i values by dividing each by the total sum. We then iteratively sample from the remaining subsets. The process stops when all elements from the universe are covered by the selected subsets. If elements remain uncovered, we remove the already selected subsets from the pool and repeat the sampling process on the remaining ones.

Table 13 showcases the best results obtained by our methods across three trials for each WSCP instance ($\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{B}_3$) compared to the results achieved by the randomized rounding algorithm. Notably, PPB-MCTS consistently outperforms the randomized rounding algorithm across all three problem instances.



(a) Total Weight Distribution for $\mathcal{B}_2$ (b) Total Weight Distribution for $\mathcal{B}_3$

Fig. 16. The density distribution of total weight obtained by PPB-MCTS, MP-MCTS, and TDS-MCTS algorithms across WSCP instances $\mathcal{B}_1$ and $\mathcal{B}_2$ when utilizing 512 parallel processes.

Fig. 16 demonstrates the promising performance of PPB-MCTS by earning more total weight close to the minimum total weight found through its rollouts. This effect is particularly pronounced for the $\mathcal{B}_3$ problem instance. We can also observe that a modest portion of rollouts conducted by PPB-MCTS still achieved smaller total weight compared to those gained by MP-MCTS and TDS-MCTS for $\mathcal{B}_3$ instance, as evidenced by the density distribution of total weight.

7. Conclusion

We proposed a new distributed-memory MCTS search algorithm, PPB-MCTS. Our algorithm improves upon existing parallel MCTS algorithms for distributed-memory parallel environments by eliminating the use of the required information table to perform partial backpropagation. The design of the algorithm increases the accuracy of deciding the point in the underlying search tree for stopping the backpropagation. We modeled JSSP and WSCP as deterministic MDP and applied our algorithm to find suboptimal solutions for these problems. When solving these problems, the number of actions can grow exponentially, leading to extensive communication if an information table is transmitted as part of the messages between processes. This characteristic can be found in other NP-hard combinatorial optimization problems and we expect that applying the proposed PPB-MCTS algorithm can lead to superior solutions. Results of our experiments showed that PPB-MCTS demonstrates better scalability by performing more rollouts as the number of processes increases while providing better load balance among processes in the backpropagation phase. In future work, we plan to consider applying our algorithm to other NP-hard combinatorial optimization problems. We also plan to combine PPB-MCTS with machine learning algorithms utilized as evaluation functions in the rollout phase, which may lead to faster rollouts.

CRediT authorship contribution statement

Yashar Naderzadeh: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Daniel Grosu:** Conceptualization, Formal analysis, Investigation, Methodology, Project administration, Software, Supervision, Writing – original draft, Writing – review & editing. **Ratna Babu Chinnam:** Conceptualization, Investigation, Methodology, Writing – original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] L. Kocsis, C. Szepesvári, Bandit based Monte-Carlo planning, in: Proc. of the 17th European Conference on Machine Learning (ECML 2006), Springer, Berlin, Germany, 2006, pp. 282–293.
[2] D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al., Mastering the game of go with deep neural networks and tree search, Nature 529 (2016) 484–489.
[3] X. Guo, S. Singh, R. Lewis, H. Lee, Deep learning for reward design to improve Monte Carlo tree search in atari games, arXiv preprint, arXiv:1604.07095, 2016.
[4] C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, A survey of Monte Carlo tree search methods, IEEE Trans. Comput. Intell. AI Games 4 (2012) 1–43.
[5] G.M.B. Chaslot, M.H. Winands, H.J. van Den Herik, Parallel Monte-Carlo tree search, in: Proc. of the 6th International Conference on Computers and Games (CG 2008), Springer, 2008, pp. 60–71.
[6] E. Steinmetz, M. Gini, More trees or larger trees: parallelizing Monte Carlo tree search, IEEE Trans. Games 13 (2020) 315–320.
[7] R.B. Segal, On the scalability of parallel uct, in: Proc. of the 7th International Conference on Computers and Games (CG 2010), Springer, 2010, pp. 36–47.
[8] S.A. Mirsoleimani, H.J. van den Herik, A. Plaat, J. Vermaseren, A lock-free algorithm for parallel mcts, in: Proc. of the 10th International Conference on Agents and Artificial Intelligence (ICAART 2018), vol. 2, 2018, pp. 589–598.
[9] A. Skrynnik, A. Andreychuk, K. Yakovlev, A. Panov, Decentralized Monte Carlo tree search for partially observable multi-agent pathfinding, in: Proc. of the AAAI Conference on Artificial Intelligence, vol. 38, 2024, pp. 17531–17540.
[10] G. Best, O.M. Cliff, T. Patten, R.R. Mettu, R. Fitch, Dec-mcts: decentralized planning for multi-robot active perception, Int. J. Robot. Res. 38 (2019) 316–337.
[11] G. Best, R. Fitch, Probabilistic maximum set cover with path constraints for informative path planning, in: Proc. of the Australasian Conference on Robotics and Automation, ARAA, 2016.
[12] E. Scheide, G. Best, G.A. Hollinger, Behavior tree learning for robotic task planning through Monte Carlo dag search over a formal grammar, in: Proc. of the IEEE International Conference on Robotics and Automation (ICRA), IEEE, 2021, pp. 4837–4843.
[13] M. Enzenberger, M. Müller, A lock-free multithreaded Monte-Carlo tree search algorithm, in: Proc. of the 12th International Conference Advances in Computer Games (ACG 2009), Springer, 2010, pp. 14–20.
[14] K. Yoshizoe, A. Kishimoto, T. Kaneko, H. Yoshimoto, Y. Ishikawa, Scalable distributed Monte-Carlo tree search, in: Proc. of the International Symposium on Combinatorial Search (SoCS 2011), vol. 2, 2011, pp. 180–187.
[15] X. Yang, T.K. Aasawat, K. Yoshizoe, Practical massively parallel Monte-Carlo tree search applied to molecular design, arXiv preprint, arXiv:2006.10504, 2020.

[16] T. Graf, U. Lorenz, M. Platzner, L. Schaefers, Parallel Monte-Carlo tree search for hpc systems, in: Proc. of the 17th International European Conference on Parallel Processing (Euro-Par 2011), vol. 2, Springer, 2011, pp. 365–376.
[17] L. Schaefers, M. Platzner, Distributed Monte Carlo tree search: a novel technique and its application to computer go, IEEE Trans. Comput. Intell. AI Games 7 (2014) 361–374.
[18] J.W. Romein, A. Plaat, H.E. Bal, J. Schaeffer, Transposition table driven work scheduling in distributed search, in: Proc. of the 16th National Conference on Artificial Intelligence (AAAI, 1999, 1999, pp. 725–731.
[19] E. Leurent, O.-A. Maillard, Monte-Carlo graph search: the value of merging similar states, in: Proc. of the 12th Asian Conference on Machine Learning (ACML 2020), 2020, pp. 577–592.
[20] J. Czech, P. Korus, K. Kersting, Improving alphazero using Monte-Carlo graph search, in: Proc. of the 31st International Conference on Automated Planning and Scheduling (ICAPS 2021), vol. 31, 2021, pp. 103–111.
[21] A. Saffidine, T. Cazenave, J. Méhat, Ucd: upper confidence bound for rooted directed acyclic graphs, Knowl.-Based Syst. 34 (2012) 26–33.
[22] F.-Y. Wang, J.J. Zhang, X. Zheng, X. Wang, Y. Yuan, X. Dai, J. Zhang, L. Yang, Where does alphago go: from church-Turing thesis to alphago thesis and beyond, IEEE/CAA J. Autom. Sin. 3 (2016) 113–120.
[23] S.D. Holcomb, W.K. Porter, S.V. Ault, G. Mao, J. Wang, Overview on deepmind and its alphago zero ai, in: Proc. of the International Conference on Big Data and Education (ICBDE 2018), ACM, 2018, pp. 67–71.
[24] S. Gelly, L. Kocsis, M. Schoenauer, M. Sebag, D. Silver, C. Szepesvári, O. Teytaud, The grand challenge of computer go: Monte Carlo tree search and extensions, Commun. ACM 55 (2012) 106–113.
[25] A.R. Kan, Machine Scheduling Problems: Classification, Complexity and Computations, Springer Science & Business Media, 2012.
[26] K. Gao, Z. Cao, L. Zhang, Z. Chen, Y. Han, Q. Pan, A review on swarm intelligence and evolutionary algorithms for solving flexible job shop scheduling problems, IEEE/CAA J. Autom. Sin. 6 (2019) 904–916.
[27] A.S. Manne, On the job-shop scheduling problem, Oper. Res. 8 (1960) 219–223.
[28] K. Kurzer, C. Hörtnagl, J.M. Zöllner, Parallelization of Monte Carlo tree search in continuous domains, arXiv preprint, arXiv:2003.13741, 2020.
[29] T. Cazenave, N. Jouandeau, On the parallelization of uct, in: Proc. of the Computer Games Workshop (CGW 2007), 2007, pp. 165–174.
[30] M. Enzenberger, M. Müller, A lock-free multithreaded Monte-Carlo tree search algorithm, in: Proc. of the 12th International Conference on Advances in Computer Games (ACG 2009), Springer, 2009, pp. 14–20.
[31] T.P. Runarsson, M. Schoenauer, M. Sebag, Pilot, rollout and Monte Carlo tree search methods for job shop scheduling, in: Proc. of the 6th International Conference on Learning and Intelligent Optimization (LION 6), Springer, 2012, pp. 160–174.
[32] M. Saqlain, S. Ali, J. Lee, A Monte-Carlo tree search algorithm for the flexible job-shop scheduling in manufacturing systems, Flex. Serv. Manuf. J. (2022) 1–24.
[33] P.-T. Lin, K.-S. Tseng, Maximal coverage problems with routing constraints using cross-entropy Monte Carlo tree search, Auton. Robots 48 (2024) 1–22.
[34] R.E. Korf, Depth-first iterative-deepening: an optimal admissible tree search, Artif. Intell. 27 (1985) 97–109.
[35] A. Kishimoto, J. Schaeffer, Distributed game-tree search using transposition table driven work scheduling, in: Proc of the 31st International Conference on Parallel Processing (ICPP 2002), IEEE, 2002, pp. 323–330.
[36] R. Munos, et al., From bandits to Monte-Carlo tree search: the optimistic principle applied to optimization and planning, Found. Trends Mach. Learn. 7 (2014) 1–129.
[37] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, MIT Press, 2018.
[38] D.P. Williamson, D.B. Shmoys, The Design of Approximation Algorithms, Cambridge University Press, 2011.
[39] S. Lawrence, Resouce Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques (Supplement), Graduate School of Industrial Administration, Carnegie-Mellon University, 1984.
[40] E. Taillard, Benchmarks for basic scheduling problems, Eur. J. Oper. Res. 64 (1993) 278–285.
[41] C. Zhang, W. Song, Z. Cao, J. Zhang, P.S. Tan, X. Chi, Learning to dispatch for job shop scheduling via deep reinforcement learning, Adv. Neural Inf. Process. Syst. 33 (2020) 1621–1632.
[42] B.-A. Han, J.-J. Yang, Research on adaptive job shop scheduling problems based on dueling double dqn, IEEE Access 8 (2020) 186474–186495.
[43] V. Batagelj, U. Brandes, Efficient generation of large random networks, Phys. Rev. E 71 (2005) 036113.

Yashar Naderzadeh earned his Master of Science in Electrical Engineering from the University of Tehran, Iran, in 2015. Currently, he is actively pursuing dual Master of Science and Ph.D. degrees in Computer Science and Industrial Engineering at Wayne State University, Detroit, USA. His research focuses on several areas, including reinforcement learning, parallel and distributed computing, and information theory.

Daniel Grosu received the Diploma in engineering (automatic control and industrial informatics) from the Technical University of Iaşi, Romania, in 1994 and the MSc and PhD degrees in computer science from the University of Texas at San Antonio in 2002 and 2003, respectively. Currently, he is an associate professor in the Department of Computer Science, Wayne State University, Detroit. His research interests include parallel and distributed computing, algorithms, and topics at the border of computer science, game theory and economics. He has published more than one hundred peer-reviewed papers in the above areas. He is an *IEEE Computer Society Distinguished Contributor*. He serves as a Senior Associate Editor for *ACM Computing Surveys* and as an Associate Editor for *IEEE Transactions on Parallel and Distributed Systems*, and *IEEE Transactions on Cloud Computing*. He is a Distinguished Member of the ACM, and a senior member of the IEEE, and the IEEE Computer Society.

Ratna Babu Chinnam received his B.S. degree in Mechanical Engineering from Manipal Institute of Technology of Mangalore University (India) in 1988 and the M.S. and Ph.D. degrees in Industrial Engineering from Texas Tech University (U.S.A.) in 1990 and 1994, respectively. He is a Professor in the Industrial & Systems Engineering Department at Wayne State University. He is the author of over 150 technical publications (journal articles, conference proceedings, and research reports). His research interests include machine learning, business analytics, big data, supply chain management, sustainability, healthcare, and smart engineering systems.