Operation Parallelism in Large Neighborhood Search for Anytime Multi-Agent Path Finding

Shao-Hung Chan[1], Zhe Chen[2], Dian-Lun Lin[3], Yue Zhang[2], Daniel Harabor[2], Sven Koenig[4], Tsung-Wei Huang[3], and Thomy Phan[5]

2025 IEEE International Symposium on Multi-Robot and Multi-Agent Systems (MRS) | 979-8-3315-9359-9/25/$31.00 ©2025 IEEE | DOI: 10.1109/MRS66243.2025.11357253

***Abstract*— Multi-Agent Path Finding (MAPF) is the problem of finding a set of collision-free paths for multiple agents in a shared environment while minimizing the sum of travel time. Since solving the MAPF problem optimally is NP-hard, anytime algorithms based on Large Neighborhood Search (LNS) are promising for finding good-quality solutions in a scalable manner by iteratively destroying and repairing paths. We propose *Destroy-Repair Operation Parallelism for LNS (DROP-LNS)*, a parallel framework that performs multiple destroy and repair operations concurrently to explore more regions of the search space within a limited time budget. Unlike classic MAPF approaches, DROP-LNS can leverage multi-threading hardware to enhance solution quality. We also formulate two variants of parallelism and conduct experimental evaluations. The empirical results show that DROP-LNS significantly outperforms the leading MAPF-LNS and LaCAM*, as well as other parallelism variants.**

I. INTRODUCTION

A wide range of real-world applications can be formulated as the *Multi-Agent Path Finding (MAPF)* problem, such as autonomous warehouses [1] and autonomous vehicles [2]. MAPF aims to find a set of collision-free paths, each from an assigned start location to a goal location, for multiple agents in a shared environment while minimizing the sum of travel time [3]. However, solving MAPF optimally is NP-hard, which limits the scalability of many algorithms [4].

Anytime algorithms are promising approaches to scaling up MAPF to hundreds of agents by iteratively optimizing a set of collision-free paths until a user-specified time budget is exhausted. Based on *Large Neighborhood Search (LNS)*, *MAPF-LNS* is the current leading anytime MAPF algorithm [5]. MAPF-LNS starts with an initial collision-free solution computed by a fast but suboptimal algorithm. It iteratively selects a subset of agents, known as *neighborhood*, and performs destroy and repair operations to replan their paths while keeping the other paths fixed. The repair operations can be done with any fast algorithm such as Prioritized Planning (PP) [6]. However, as the number of agents and the size of the environment increase, PP becomes a bottleneck during the search because the underlying single-agent path-finding algorithm slows down due to more temporal obstacles and longer path lengths. This predominantly affects the speed of the replan operations. Thus, MAPF-LNS may converge to poor-quality solutions in large-scale MAPF instances.

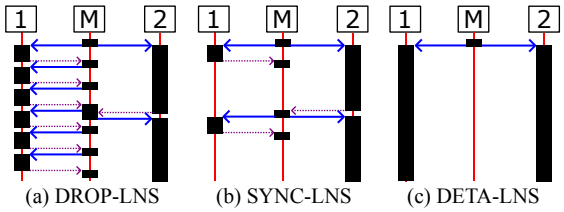

Fig. 1. Timelines of parallelism variants with a main thread "M" and two worker threads "1" and "2". Black blocks indicate the *productivity* (the time when threads are processing), and red lines indicate the idle time. Blue arrows indicate events where a worker thread receives tasks, and purple dotted arrows indicate events where a worker thread returns a solution.

Despite the rapid progress in the field of MAPF that resulted in many sophisticated algorithms [7], [8], few exploit parallelism to improve the anytime MAPF for better scalability and solution quality [9]. Given the availability of parallelism, there is a lot of potential to fully exploit the available time budget. However, parallelism is not trivial as the search algorithm needs to balance between *productivity* and *synchronization*. The former describes the concurrent execution of tasks, and the latter describes the access to the best-known solution to guide the search toward better quality. Any imbalanced approach can lead to inefficient parallel search, where worker threads are either less productive due to *synchronization overhead*, i.e., the idle time of the threads (see Fig. 1(b)) or redundant because of exploring similar regions in the search space (see Fig. 1(c)).

In this paper, to bridge the gap between anytime MAPF algorithms and parallelism, we propose *Destroy-Repair Operation Parallelism for LNS (DROP-LNS)*, a parallelism framework that concurrently performs destroy and repair operations to explore more solution candidates with high-quality solutions within a limited time budget. Our contributions are as follows:

- We propose DROP-LNS, which performs pairs of destroy and repair operations via parallelism. The solution is updated asynchronously to achieve high productivity, i.e., a low synchronization time.
- We propose numeric metrics to evaluate the efficiency and effectiveness of anytime MAPF algorithms. We also demonstrate that DROP-LNS achieves better solution quality and scalability than both the other parallelism variants and the leading anytime MAPF algorithms, namely MAPF-LNS and LaCAM*.

[1]University of Southern California, USA. shaohung@usc.edu
[2]Monash University, Australia.
{Zhe.Chen,yue.zhang,daniel.harabor}@monash.edu
[3]University of Wisconsin-Madison, USA.
{dianlun.lin, tsung-wei.huang}@wisc.edu
[4]University of California, Irvine, USA. svenk@uci.edu
[5]University of Bayreuth, Germany. thomyp@uci.edu

II. PRELIMINARIES

A. Problem Definition

A MAPF problem [3] consists of an undirected and unit-cost graph and a set of k agents $A = \{a_1...a_k\}$. Each agent $a_i \in A$ is assigned a start vertex s_i and a goal vertex g_i. Time is discretized into timesteps. A *state* of an agent is represented as a tuple (v, t) indicating its location at vertex v at timestep t. At each timestep, an agent can either *wait* at its current vertex or *move* to an adjacent vertex. A *collision* between two paths occurs when two agents occupy the same vertex or pass through the same edge in opposite directions at the same timestep. A *solution* is a set of paths $P = \{p_1, ..., p_k\}$, with one path $p_i \in P$ for each agent $a_i \in A$. A solution is *feasible* if the set of paths is collision-free. The *cost* $c(p_i)$ of a path p_i is the number of timesteps or travel time to get from start vertex s_i to goal vertex g_i. In this paper, our goal is to find a feasible solution while minimizing the *sum of costs (SOC)* $c(P) = \sum_{i=1}^{k} c(p_i)$.

B. Large Neighborhood Search for MAPF

Starting from a feasible solution, anytime MAPF aims to minimize the SOC of iteratively [10]. Based on the *Large Neighborhood Search (LNS)*, a meta-heuristic search algorithm for combinatorial optimization, *MAPF-LNS* is the leading algorithm for anytime MAPF, which can scale up to large-scale scenarios [5], [11]. Starting with an initial feasible solution P found by fast but suboptimal algorithms like PP [6], MAPF-LNS iteratively modifies P by heuristically selecting a subset of N agents $A' \subset A$ as the *neighborhood* with their corresponding paths $P' = \{p_i \in P | a_i \in A'\}$ from P, where $N < k$ is a user-specified parameter. MAPF-LNS then repairs the paths P' with a new set of paths P'_{new} generated by PP within a limited repair time budget. Since PP finds the paths sequentially according to a priority ordering of A', each new path is supposed to avoid any collisions with the set of *already-planned paths* $(P \setminus P') \cup P'_{new}$ to ensure the new solution is still feasible. If MAPF-LNS finds such a path successfully, it adds the path to paths P'_{new}. If each agent in neighborhood A' has a corresponding path in paths P'_{new} and the SOC of the paths P'_{new} is lower than that of paths P', then MAPF-LNS "destroys" the previous paths P' from P and "repairs" them with P'_{new}. During the search, we call the feasible solution with the lowest SOC found so far the *best-known solution*. MAPF-LNS continues searching for solutions with lower SOCs until the time budget runs out.

MAPF-LNS uses a set $\mathcal{H}$ of three *destroy heuristics* for selecting neighborhoods, namely a random-based heuristic, an agent-based heuristic, and a map-based heuristic. To select a destroy heuristic $H \in \mathcal{H}$ at each iteration, MAPF-LNS maintains a set of updatable *weights* w_H, one for each destroy heuristic and uses a *roulette wheel selection* mechanism, where each destroy heuristic H is selected with the probability of $\frac{w_H}{\sum_{H \in \mathcal{H}} w_H}$ [12]. After destroying and repairing paths of agents in the neighborhood, MAPF-LNS updates the value of weight w_H corresponding to the selected destroy heuristic H:

$$w_H \leftarrow \gamma \max\{c(P') - c(P'_{new}), 0\} + (1 - \gamma) w_H, \quad (1)$$

where $\gamma \in [0, 1]$ is a user-specified reaction factor indicating how fast the weight value changes in reaction to the improvement of the solution quality.

C. Lazy Constraint Addition Search for MAPF

Lazy Constraint Addition Search for MAPF (LaCAM) [13] is a suboptimal algorithm that performs a search on the joint-state space, where each node in its search tree is a *joint-state* of the agents on the graph, i.e., a sequence of non-repeated vertices, one for each agent. To efficiently generate a suboptimal solution, LaCAM uses Priority Inheritance with Backtracking (PIBT) [14], a rule-based algorithm for solving MAPF suboptimally to determine movements for all agents step-wise. With a systematic search and the invocation of a pattern-based swap operation [15], [16], LaCAM computes high-quality plans while retaining its performance advantages. Based on LaCAM, *LaCAM** [13] is an anytime MAPF algorithm that minimizes either the makespan (the maximum individual cost) or the sum of loss (the number of agents that have not reached their goals yet). Once LaCAM finds a solution, LaCAM* continues the search for better-quality solutions while keeping track of the minimum makespan (or sum of loss). We use LaCAM* as one of our baselines.

III. RELATED WORK

Despite significant progress in MAPF in recent years, there has been limited work on parallelization to scale MAPF algorithms with the growing availability of low-cost parallel hardware. Most works focus on some form of task decomposition, where parallelism is done at the level of the subproblems. For example, Lee et al. decompose the task into subproblems solved in parallel and merge them into larger subproblems until the original task is completely solved [17]. In contrast to these works that focus on parallelization in finding one-shot solutions, we focus on iterative optimization in an anytime manner. Furthermore, we propose parallelism on the *operation level* of the MAPF algorithm, which is less dependent on particular map structures and the number of agents. The closest work to our focus is a variant, called DETA-LNS, where multiple MAPF-LNS processes are performed on independent threads [9]. However, this approach causes wasteful overlaps in the independent search beams as it lacks a synchronization mechanism to focus on more promising solutions.

Apart from MAPF, a previous work called *Parallel Adaptive Large Neighborhood Search (PALNS)* solves the Traveling Salesman Problem with Pickup and Delivery and the Capacitated Vehicle Routing Problem [18]. The tasks for parallelization comprise pairs of destroy and repair operators executed simultaneously on a given solution. We adopt PALNS for the MAPF setting, resulting in our DROP-LNS. We use the destroy heuristics proposed in [5] and PP with randomized priorities as a repair operator.

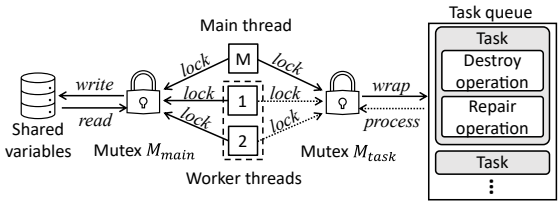

Fig. 2. Illustrative example of the DROP-LNS framework with a main thread "M" and two worker threads "1" and "2". Arrows are the actions from each thread.

Algorithm 1 Main Function of DROP-LNS

```
1:  procedure MAIN(I, T, N, m)
2:      Initialize
3:          w_H ← 1, ∀H ∈ ℋ
4:          task queue Q_task ← ϕ
5:          mutexes M_main and M_task
6:      P_min = {p_1, ..., p_k} ← Find initial solution with I, T
7:      Activate m worker threads
8:      while runtime not exceeds T do
9:          [lock M_task]
10:         if Q_task is empty then
11:             Wrap DESTROYANDREPAIR tasks to Q_task
12:         end if
13:         [unlock M_task]
14:     end while
15:     Stop all the worker threads
16:     return P_min
17: end procedure
```

IV. Methodology

A. Destroy-Repair Operation Parallelism

Fig. 2 shows an illustrative example of the DROP-LNS framework. DROP-LNS uses a *main thread* and m *worker threads* to parallelize the search. The core idea is to wrap pairs of destroy and repair operations as *tasks* via the main thread and assign these tasks to idle worker threads. DROP-LNS maintains *shared variables*, denoted in bold, that can be modified by any thread, including the *best-known solution* $\mathbf{P}_{min}$ with the minimum SOC found so far, the weights $\mathbf{w}_H$ for each destroy heuristic $H \in \mathcal{H}$, and a task queue $\mathbf{Q}_{task}$ with a fixed capacity. It also maintains *shared constants* that are read-only and cannot be modified after initialization, including the given MAPF instance I and three user-specified parameters: the neighborhood size N, the time budget T, and the reaction factor γ. To synchronize the shared variables, DROP-LNS uses two *mutexes*: M_{task} to ensure that only one thread is allowed to modify the task queue at a time, and M_{main} for modifying any other shared variables. Any thread should acquire or *lock* the mutex to modify the corresponding shared variables and release that mutex after the modification. For example, once a worker thread finishes its current task, it tries to acquire mutex M_{task} to pop the next task out of task queue $\mathbf{Q}_{task}$. If mutex M_{task} is locked by another thread, the worker thread has to wait for it to be released. Analogously, mutex M_{main} has to be acquired to access or modify the other shared variables.

Like MAPF-LNS, DROP-LNS first uses the main thread to initialize a feasible solution via LaCAM [13]. The main thread then fills task queue $\mathbf{Q}_{task}$ with the fixed capacity of tasks until the time budget runs out. The process is formulated in Algorithm 1, where I is the MAPF instance to be solved, T is the time budget, N is the neighborhood size, and m is the number of threads. We mark lines requiring access to shared variables in blue.

An idle worker thread tries to access task queue $\mathbf{Q}_{task}$ by acquiring mutex M_{task}. If successful, the thread locks M_{task} and pops a task from the queue. The task function of DROP-LNS is formulated in Algorithm 2, where each worker thread performs a task with private variables that can only be accessed and modified by the corresponding worker thread. Before executing a task, the worker thread tries to acquire mutex M_{main} in order to copy the shared variables to its private variables, including a copy P of the currently best-known solution $\mathbf{P}_{min}$ and a copy w_H of weights $\mathbf{w}_H$, for all $H \in \mathcal{H}$. The worker thread then releases mutex M_{main} to enable access or modification of the shared variables by other threads and computes the SOC C of paths P [Lines 2-7]. To perform destroy operations, the worker thread samples a destroy heuristic H' using the roulette wheel selection mechanism with weights w_H. Then, the worker thread selects a subset of N agents $A' \subseteq A$ as the neighborhood along with their paths $P' \subseteq P$ according to destroy heuristic H', and removes P' from P [Lines 8-11].

To perform the repair operation, the worker thread uses PP to generate paths for agents in neighborhood N. If PP fails to find such paths within the repair time budget or the repaired paths P'_{new} has a higher SOC than that of destroyed paths P', the worker thread acquires mutex M_{main} and lowers weight $\mathbf{w}_{H'}$ by a factor of $1-\gamma$ without updating paths $\mathbf{P}_{min}$, which indicates a failed iteration. After that, the worker thread terminates the task function and tries to fetch the first task in task queue $\mathbf{Q}_{task}$ again [Lines 12-18]. Otherwise, the worker thread repairs paths P with P'_{new} and acquires mutex M_{main} in order to access the shared variables, namely paths $\mathbf{P}_{min}$ and weight $\mathbf{w}$. The worker thread first updates the value of weight $\mathbf{w}_{H'}$ to

$$\mathbf{w}_{H'} \leftarrow \gamma[c(P) - c(P \setminus P' \cup P'_{new})] + (1-\gamma)\mathbf{w}_{H'}, \quad (2)$$

as it modifies the SOC from $c(P)$ to $c(P \setminus P' \cup P'_{new})$ after processing the task. Then, the worker thread updates path $\mathbf{P}_{min}$ if SOC $c(P \setminus P' \cup P'_{new}))$ is lower than that of $c(P_{min})$, and releases the mutex so that other threads can access the shared variables [Lines 19-27]. We define *synchronization* as a two-step process that (1) compares the SOCs between the solution P generated by the worker thread and the best-known solution $\mathbf{P}_{min}$ and (2) updates the best-known solution and the weight of each destroy heuristic accordingly.

B. Parallelism Variants of MAPF-LNS

1) SYNC-LNS: SYNC-LNS keeps track of the best-known solution $\mathbf{P}_{min}$. At each iteration, SYNC-LNS selects the same number of neighborhoods as the worker threads. Each worker

Algorithm 2 Task Function of DROP-LNS

procedure DESTROYANDREPAIR
  Initialize
    [lock M_{main}]
    $P \leftarrow \mathbf{P}_{min}$
    $w_H \leftarrow \mathbf{w}_H, \forall H \in \mathcal{H}$
    [unlock M_{main}]
    $C \leftarrow c(P)$
  $H' \leftarrow$ Select a destroy heuristic with $w_H, \forall H \in \mathcal{H}$
  $A' \leftarrow$ Select a neighborhood with H'
  $P' \leftarrow \{p_i \in P | a_i \in A'\}$
  $P \leftarrow P \setminus P'$ ▷ Destroy the subset of paths
  $P'_{new} \leftarrow$ Run PP within a repair time budget
  if P'_{new} **not** found **or** $c(P'_{new}) \geq c(P')$ **then**
    [lock M_{main}]
    $\mathbf{w}_{H_s} \leftarrow (1-\gamma) \cdot \mathbf{w}_{H_s}$
    [unlock M_{main}]
    return
  end if
  if runtime **not** exceeds T **then**
    $P \leftarrow P \cup P'_{new}$ ▷ Repair the subset of paths
    [lock M_{main}]
    $\mathbf{w}_{H_s} \leftarrow \gamma \cdot (C - c(P)) + (1-\gamma) \cdot \mathbf{w}_{H_s}$
    if $c(P) < c(\mathbf{P}_{min})$ **then**
      $\mathbf{P}_{min} \leftarrow P$
    end if
    [unlock M_{main}]
  end if
  return
end procedure

thread then performs a pair of destroy and repair operations individually in parallel. After all the worker threads complete their own destroy and repair operations, SYNC-LNS selects the worker thread that contains the solution P_{min} with the lowest SOC among all solutions generated by all worker threads. It also records the destroy heuristic $H' \in \mathcal{H}$ that the selected worker thread uses. SYNC-LNS compares the SOC between the solution P_{min} and the best-known solution $\mathbf{P}_{min}$, and then updates the value of weight $w_{H'}$ to

$$w_{H'} \leftarrow \gamma \max\{c(\mathbf{P}_{min}) - c(P_{min}), 0\} + (1-\gamma) w_{H'}. \quad (3)$$

SYNC-LNS replaces solution $\mathbf{P}_{min}$ with P_{min} if the latter has a lower SOC than the former. That is, it synchronizes solutions and weights at each iteration by selecting the one with the lowest SOC, as illustrated in Fig. 1(b).

2) DETA-LNS: DETA-LNS "detaches" one MAPF-LNS process individually on a worker thread in parallel with equal weights of each destroy heuristic [9]. When the time budget runs out, it selects the solution with the lowest SOC among all feasible solutions generated by the worker threads. That is, DETA-LNS never synchronizes solutions and weights developed by each thread until the time budget runs out, as illustrated in Fig. 1(c).

C. Conceptual Discussion

DROP-LNS parallelizes the search by wrapping pairs of destroy and repair operations as concurrently executable tasks while maintaining the best-known solution and the weights for destroy heuristic selection. Compared to performing destroy and repair operations sequentially on a single worker thread, parallelism can efficiently exploit high-quality solutions for further improvement and explore different regions in the search space. Both aspects increase the chance of finding solutions with lower SOCs than MAPF-LNS.

Fig. 1 shows the conceptual timelines during the search of DROP-LNS, SYNC-LNS, and DETA-LNS, respectively. SYNC-LNS performs a focused search by pruning solutions with higher SOCs at each iteration, allowing all worker threads to concentrate on higher-quality solutions. However, SYNC-LNS must wait until all worker threads complete their tasks before moving on to the next iteration, which limits the productivity of fast worker threads and thus increases the synchronization overhead. DETA-LNS only synchronizes solutions at the end when the time budget runs out, meaning that its worker threads do not need to wait for one another. Thus, DETA-LNS theoretically exhibits the highest possible productivity of worker threads. Since all worker threads process independently, they may explore overlapping regions in the search space, which leads to finding similar solutions and thus limits the effectiveness due to a lack of exploitation of high-quality solutions. Meanwhile, DROP-LNS synchronizes solutions on the fly. Once a worker thread completes its task, it synchronizes without waiting for others. Thus, DROP-LNS maintains higher productivity than SYNC-LNS. Unlike DETA-LNS, DROP-LNS continues to synchronize its solutions. Thus, DROP-LNS trades off between pruning bad-quality solutions and the synchronization overhead.

V. EMPIRICAL EVALUATION

A. Configurations and Algorithm Implementation

We use 4-connected grid maps from the MAPF benchmark suite [3]. Two of them are small maps of size 32×32: a `Room` map (*room-32-32-4*) and a `Random` map with 20% static obstacles (*random-32-32-10*). The other two are large maps: a `Warehouse` map (*warehouse-10-20-10-2-1*) and a `Den520d` map from the game Dragon Age: Origins. Fig. 3 shows the configuration and the size of each map. We conduct all experiments on the 25 randomly generated scenarios for each map. Since the benchmark only provides instances with at most 1,000 agents for `Den520d` map, we use those 25 instances when the number of agents is set to 1000 and create 25 instances when the number of agents is set to 2000 and 3000. All the experiments are conducted with a time budget $T = 60$ seconds.

We implement DROP-LNS, SYNC-LNS, and DETA-LNS as parallelism variants with fixed neighborhood size $N = 16$ and reaction factor $\gamma = 0.01$ and use $m = 8$ worker threads unless mentioned otherwise. We modify LaCAM* by tracking the total number of timesteps agents took before their last visit to their respective goal vertices so that it becomes an anytime algorithm that optimizes SOC.

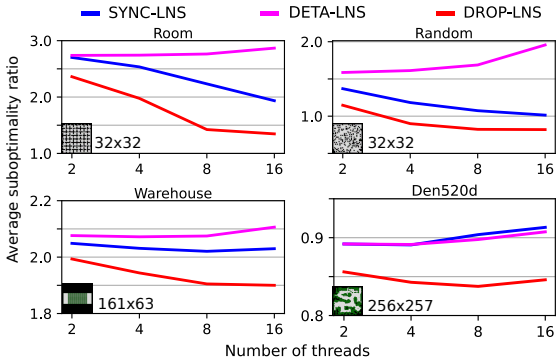

Fig. 3. Average solution quality among all instances with the highest number of agents on each map solved by SYNC-LNS, DETA-LNS, and DROP-LNS with 2, 4, 8, and 16 threads, respectively. The lower the average suboptimality ratio, the better the solution quality.

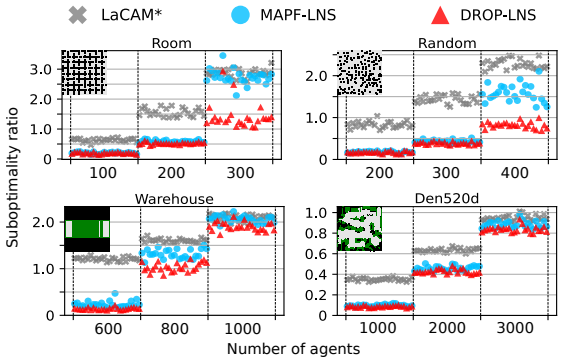

Fig. 4. Solution quality of instances solved by LaCAM*, MAPF-LNS, and DROP-LNS. Instances are grouped by the number of agents. The lower the average suboptimality ratio, the better the solution quality.

B. Evaluation Metrics

1) Solution Quality: To evaluate the solution quality among instances, we define *the shortest path distance* d_i of an agent a_i as the minimum timesteps needed to move from start vertex s_i to goal vertex g_i. Thus, the sum of the shortest path distances provides a *lowerbound* of the optimal SOC. We then define the *delay* of an agent a_i as the difference between its path cost $c(p_i)$ and its shortest path distance d_i. To evaluate the solution quality of different algorithms, we compare the *suboptimality ratio* between the *sum of delays* over all agents and the lowerbound, i.e.,

$$\textit{suboptimality ratio} = \frac{\sum_{a_i \in A} (c(p_i) - d_i)}{\sum_{a_i \in A} d_i}. \tag{4}$$

Since the sum of the shortest path distances is a constant given an instance, comparing the suboptimality ratios is equivalent to comparing the SOCs of solutions. The lower the suboptimality ratio, the better the solution quality is.

2) Effectiveness: To evaluate the effectiveness of algorithms, we focus on the following aspects:

(1) How fast an algorithm can find a high-quality solution.
(2) How productive the worker threads are.
(3) How often an algorithm exploits the best-known solution during the search.

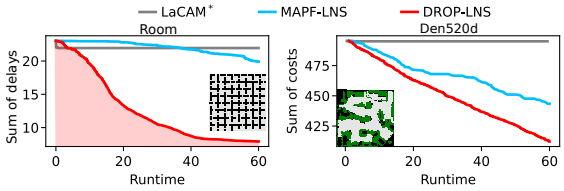

Fig. 5. Sum of delays (in thousands) versus runtime (in seconds) in two instances with 300 agents in `Room` map and with 3000 agents in `Den520d` respectively solved by LaCAM*, MAPF-LNS, and DROP-LNS. The pink region indicates the AUC of DROP-LNS.

(4) How often an algorithm explores different solutions.

For aspect **(1)**, we measure the *area under the curve* (AUC), which is defined as the integral of the sum of delays of the best-known solution versus runtime (see Fig. 5). The lower the AUC, the faster an algorithm converges to a high-quality solution. For aspect **(2)**, we measure the *number of pairs of destroy and repair operations* (NPO) during the search, where NPO* denotes the total NPO within the time budget. The higher the NPO*, the more productive the worker threads are (i.e., less synchronization overhead). For aspect **(3)**, we measure the *depth* of the final solution (DP), which is defined as the NPO from the initial solution to the final best-known solution. The higher the DP, the more frequently the algorithm improves its best-known solution. For aspect **(4)**, we measure the *exploration ratio* (EXP), which is defined as the ratio between the NPO that does not lead to the final best-known solution and the total NPO, i.e., EXP $=$ (NPO* $-$ DP)/NPO*. The higher the EXP, the more frequently the algorithm explores in the search space.

C. Empirical Results

Fig. 3 shows the average solution quality of different parallelism variants versus the number of threads, where DROP-LNS outperforms SYNC-LNS and DETA-LNS while maintaining its performance as the number of threads increases. Fig. 4 shows the solution quality of DROP-LNS along with the leading anytime MAPF algorithms, namely LaCAM* and MAPF-LNS, where DROP-LNS outperforms the state of the art, especially in small and congested instances like `Room` or `Random` maps with increasing numbers of agents. Fig. 5 shows the changes in the sum of delays versus the runtime while solving an instance where DROP-LNS converges to a good-quality solution faster than the state of the art, resulting in lower AUC. Table I shows the average AUC, NPO*, DP, and EXP among all compared algorithms, where `RO`, `RA`, `W`, and `D` in the first column stand for maps `Room`, `Random`, `Warehouse`, and `Den520d`, respectively. For AUC (w.r.t. NPO*, DP, and EXP), numbers in bold are the minimum (w.r.t. maximum) among all in the same row. The AUC and DP of DETA-LNS are obtained from the worker thread that has the best-known solution when the search ends. DROP-LNS typically has a lower AUC, especially in small and congested instances. Table II shows the average memory usage over MAPF instances with the highest number of agents on each map, where LaCAM* can be more memory-consuming than all the MAPF-LNS variants by two orders

TABLE I

AVERAGE AUC (IN MILLIONS), NPO* (IN THOUSANDS), DP, AND EXP OVER ALL INSTANCES WITH THE SAME NUMBER OF AGENTS PER MAP

	k	LaCAM*	MAPF-LNS				SYNC-LNS				DETA-LNS				DROP-LNS			
		AUC	AUC	NPO*	DP	EXP	AUC	NPO*	DP	EXP	AUC	NPO*	DP	EXP	AUC	NPO*	DP	EXP
RO	100	0.10	**0.03**	26.6	137.1	**0.99**	**0.03**	72.0	127.5	**0.99**	**0.03**	**139.4**	**142.5**	**0.99**	**0.03**	129.2	134.9	**0.99**
	200	0.48	0.24	16.9	547.3	0.96	0.20	62.7	562.7	**0.99**	0.25	86.6	522.1	**0.99**	**0.19**	**104.6**	**628.7**	**0.99**
	300	1.32	1.32	8.0	68.9	**0.99**	1.23	27.4	323.0	**0.99**	1.34	47.2	52.0	**0.99**	**1.00**	**62.0**	**1194.1**	0.98
RA	200	0.22	**0.05**	21.1	372.1	0.98	**0.05**	57.1	308.7	**0.99**	0.06	**116.4**	**386.3**	**0.99**	**0.05**	108.5	366.9	**0.99**
	300	0.57	0.23	15.7	885.2	0.94	0.21	50.7	763.2	0.98	0.26	80.8	835.0	**0.99**	**0.19**	**95.1**	**976.6**	**0.99**
	400	1.22	1.08	6.5	497.4	0.92	0.83	31.4	1158.8	0.96	1.12	36.3	375.9	**0.99**	**0.69**	**70.1**	**2063.7**	0.97
W	600	3.48	2.02	0.8	560.7	0.26	2.52	1.1	144.76	**0.88**	2.69	1.6	221.3	0.87	**1.51**	**8.0**	**968.6**	0.87
	800	6.19	5.78	0.1	124.9	0.13	5.88	0.4	54.9	0.87	6.00	0.6	74.2	**0.88**	**5.20**	**1.2**	**227.7**	0.80
	1000	10.09	10.11	0.1	53.4	0.31	10.06	0.3	36.1	0.88	10.19	0.4	38.0	**0.90**	**9.74**	**0.6**	**85.6**	0.85
D	1000	3.61	1.68	0.7	**514.1**	0.25	1.99	1.7	206.6	0.88	2.03	**3.5**	372.5	**0.89**	**1.66**	3.3	414.5	0.87
	2000	13.21	11.46	0.2	**220.1**	0.03	12.30	0.5	64.2	**0.88**	12.07	1.0	144.0	0.86	**11.04**	**1.3**	199.2	0.85
	3000	29.41	28.31	0.1	100.3	0.06	28.76	0.3	37.52	**0.88**	28.47	0.6	77.0	0.87	**27.73**	**0.7**	**114.0**	0.84

TABLE II

AVERAGE MEMORY USAGE (IN MB) OF ALGORITHMS OVER INSTANCES

	RO	RA	W	D
LaCAM*	3723.4	4567.2	2491.4	2664.0
MAPF-LNS	8.3	10.8	118.7	802.0
SYNC-LNS	8.4	9.5	67.0	786.9
DETA-LNS	13.0	15.7	137.1	1096.3
DROP-LNS	9.8	11.1	145.5	1294.1

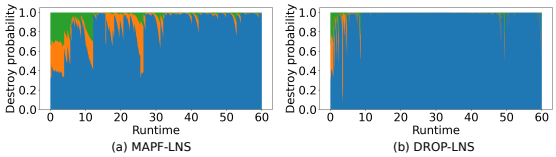

Fig. 6. Probability distributions of destroy heuristics of DROP-LNS with 8 worker threads and MAPF-LNS over an instance with 300 agents in Room map. The blue, orange, and green indicate the probabilities of random-based, agent-based, and map-based heuristics.

of magnitude when solving congested instances. Meanwhile, DROP-LNS, even with 8 worker threads, uses significantly less memory and reaches better solution quality and AUC than LaCAM*.

D. Empirical Discussion

DETA-LNS theoretically runs several MAPF-LNS independently; however, in comparison to MAPF-LNS, its NPO* and memory usage do not grow in proportion to the number of threads due to the limited memory bandwidth. Along with poor exploitation of the best-known solutions, DETA-LNS results in poor solution quality when the number of threads increases. Still, DETA-LNS can reach higher NPO* and EXP than MAPF-LNS, demonstrating its productivity. Meanwhile, due to synchronization at each iteration, SYNC-LNS exploits good-quality solutions and thus results in higher DP than MAPF-LNS and DETA-LNS in congested instances. If the solution selected by SYNC-LNS during the synchronization always has a lower SOC than the best-known solution, then its EXP becomes $1 - \frac{1}{m}$ since the best-known solution is selected from one of the m worker threads. However, since SYNC-LNS requires all worker threads to wait until each of them finishes its operations, it is less efficient in MAPF instances with large maps, such as Den520d, where the runtime of repair operations can have large standard deviations.

DROP-LNS strikes a balance between productivity and synchronization by updating the best-known solution in real-time. Compared to SYNC-LNS, DROP-LNS requires less synchronization overhead, typically resulting in a higher NPO*. Compared to DETA-LNS, DROP-LNS has a lower AUC due to the exploitation of the synchronized best-known solution. Also, as shown in Fig. 6, both DROP-LNS and MAPF-LNS converge to the random-based destroy heuristic during the search. However, due to its higher NPO*, DROP-LNS has a higher chance of escaping from the local minimum than MAPF-LNS and thus results in a lower AUC.

VI. CONCLUSION

We presented DROP-LNS, a parallel framework that performs multiple destroy and repair operations concurrently, while the best-known solution is updated asynchronously to maintain the productivity of worker threads. DROP-LNS strikes a balance between productivity and synchronization to achieve better performance. It also keeps the idle time per worker thread low while focusing the search on more promising solutions. The empirical evaluations show that DROP-LNS is more effective than other parallelism variants and the leading anytime algorithms, e.g., MAPF-LNS and LaCAM*. Future work includes developing parallelism on GPUs with more sophisticated mechanisms for synchronization.

VII. ACKNOWLEDGMENTS

The research at the University of California, Irvine and University of Southern California was supported by the National Science Foundation (NSF) under grant numbers 2434916, 2346058, 2321786, 2121028, and 1935712 as well as gifts from Amazon Robotics.

REFERENCES

[1] P. R. Wurman, R. D'Andrea, and M. Mountz, "Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses," in *AI Magazine*, 2008, pp. 9–20.

[2] J. Li, T. A. Hoang, E. Lin, H. L. Vu, and S. Koenig, "Intersection Coordination with Priority-Based Search for Autonomous Vehicles," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2023, pp. 11 578–11 585.

[3] R. Stern, N. R. Sturtevant, A. Felner, S. Koenig, H. Ma, T. T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. S. Kumar, R. Barták, and E. Boyarski, "Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks," in *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, 2019, pp. 151–159.

[4] J. Yu and S. M. LaValle, "Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2013, pp. 1443–1449.

[5] J. Li, Z. Chen, D. Harabor, P. J. Stuckey, and S. Koenig, "Anytime Multi-Agent Path Finding via Large Neighborhood Search," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2021, pp. 4127–4135.

[6] D. Silver, "Cooperative Pathfinding," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2005, pp. 117–122.

[7] E. Boyarski, A. Felner, R. Stern, G. Sharon, D. Tolpin, O. Betzalel, and S. E. Shimony, "ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2015, pp. 740–746.

[8] J. Li, D. Harabor, P. J. Stuckey, H. Ma, and S. Koenig, "Symmetry-Breaking Constraints for Grid-Based Multi-Agent Path Finding," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2019, pp. 6087–6095.

[9] F. Laurent, M. Schneider, C. Scheller, J. Watson, J. Li, Z. Chen, Y. Zheng, S.-H. Chan, K. Makhnev, O. Svidchenko, V. Egorov, D. Ivanov, A. Shpilman, E. Spirovska, O. Tanevski, A. Nikov, R. Grunder, D. Galevski, J. Mitrovski, G. Sartoretti, Z. Luo, M. Damani, N. Bhattacharya, S. Agarwal, A. Egli, E. Nygren, and S. Mohanty, "Flatland Competition 2020: MAPF and MARL for Efficient Train Coordination on a Grid World," in *Proceedings of the NeurIPS 2020 Competition and Demonstration Track*, 2021, pp. 275–301.

[10] L. Cohen, M. Greco, H. Ma, C. Hernández, A. Felner, T. S. Kumar, and S. Koenig, "Anytime Focal Search with Applications," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2018, pp. 1434–1441.

[11] T. Huang, J. Li, S. Koenig, and B. Dilkina, "Anytime Multi-Agent Path Finding via Machine Learning-Guided Large Neighborhood Search," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2022, pp. 9368–9376.

[12] S. Ropke and D. Pisinger, "An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows," in *Transportation science*, 2006, pp. 455–472.

[13] K. Okumura, "Improving LaCAM for Scalable Eventually Optimal Multi-Agent Pathfinding," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2023, pp. 243–251.

[14] K. Okumura, M. Machida, X. Défago, and Y. Tamura, "Priority Inheritance with Backtracking for Iterative Multi-agent Path Finding," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2019, pp. 535–542.

[15] R. Luna and K. E. Bekris, "Push and Swap: Fast Cooperative Path-Finding with Completeness Guarantees," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2011, pp. 294–300.

[16] B. De Wilde, A. W. Ter Mors, and C. Witteveen, "Push and Rotate: A Complete Multi-Agent Pathfinding Algorithm," in *Journal of Artificial Intelligence Research (JAIR)*, vol. 51, 2014, pp. 443–492.

[17] H. Lee, J. Motes, M. Morales, and N. M. Amato, "Parallel Hierarchical Composition Conflict-Based Search for Optimal Multi-Agent Pathfinding," in *IEEE Robotics and Automation Letters*, 2021, pp. 7001–7008.

[18] S. Ropke, "Parallel Large Neighborhood Search – A Software Framework," in *Proceedings of the Metaheuristic International Conference (CDROM)*, 2009.