A Distributed Solver for Multi-Agent Path Finding Problems

Poom Pianpak, Tran Cao Son, Z O. Toups
{ppianpak,tson,z}@cs.nmsu.edu
New Mexico State University
Las Cruces, New Mexico

William Yeoh
wyeoh@wustl.edu
Washington University in St. Louis
St. Louis, Missouri

ABSTRACT

Multi-Agent Path Finding (MAPF) problems are traditionally solved in a centralized manner. There are works focusing on completeness, optimality, performance, or a tradeoff between them. However, there are only a few works based on spatial distribution. In this paper, we introduce `ros-dmapf`, a distributed MAPF solver. It consists of multiple MAPF sub-solvers, which—besides solving their assigned sub-problems—interact with each other to solve a given MAPF problem. In the current implementation, the sub-solvers are answer set planning systems for multiple agents, and are created based on spatial distribution of the problem. Interactions between components of `ros-dmapf` are facilitated by the *Robot Operating System* (ROS). The highlights of `ros-dmapf` are its scalability and a high degree of parallelism. We empirically evaluate `ros-dmapf` using the move-only domain of the *asprilo* system and results suggest that `ros-dmapf` scales up well. For instance, `ros-dmapf` gives a solution of length around 600 for a MAPF problem with 2000 robots in randomly generated 100x100 obstacle-free maps—a problem beyond the capability of a single sub-solver—within 7 minutes on a consumer laptop. We also evaluate `ros-dmapf` against some other MAPF solvers and results show that the system performs well. We also discuss possible improvements for future work.

CCS CONCEPTS

• **Computing methodologies** → **Distributed artificial intelligence**; **Multi-agent systems**; **Cooperation and coordination**.

KEYWORDS

Distributed Multi-Agent Path Finding, Scalability, Robot Operating System, Answer Set Programming

ACM Reference Format:
Poom Pianpak, Tran Cao Son, Z O. Toups and William Yeoh. 2019. A Distributed Solver for Multi-Agent Path Finding Problems. In *First International Conference on Distributed Artificial Intelligence (DAI '19), October 13–15, 2019, Beijing, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3356464.3357702

DAI '19, October 13–15, 2019, Beijing, China

ACM ISBN 978-1-4503-7656-3/19/10…$15.00
https://doi.org/10.1145/3356464.3357702

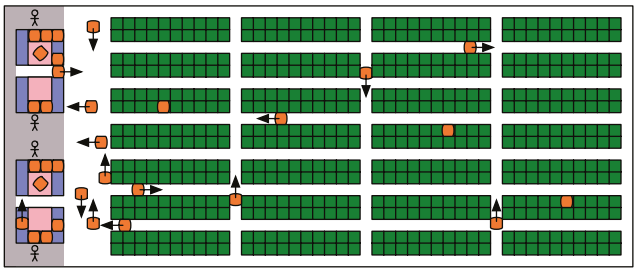

Figure 1: Layout of an Autonomous Warehouse System

1 INTRODUCTION

Multi-Agent Path Finding (MAPF) deals with agents that need to find a collision-free path from their starting to goal locations on a graph. This model can be applied to a number of applications including autonomous aircraft towing vehicles [22], autonomous warehouse systems [36], office robots [31], and video games [26]. For example, in an autonomous warehouse system (illustrated by Figure 1), robots (in orange) navigate around a warehouse to pick up inventory pods from their storage locations (in green) and drop them off at designated inventory stations (in purple).

In MAPF, the objective is to find a collision-free path for agents to move to their goal locations while minimizing either the makespan or the total path cost. Researchers have proposed various optimal and bounded-suboptimal algorithms [1, 4, 12, 25, 32] as well as suboptimal ones [5, 19, 34]. While most of them are search-based, there are also approaches that reformulate the problem using *answer set programming* (ASP) [6, 23], *mixed-integer programming* (MIP) [37], and satisfiability testing [30]. Stern *et al.* presents a short survey on different MAPF formulations and extensions [29].

The general consensus within the MAPF community has been that declarative approaches, such as those based on ASP and MIP, outperform imperative search-based approaches in small complex problems (e.g., problems where agents need to repeatedly swap locations) [23]. However, most declarative approaches fail to scale to large problems due to memory limitations—they often need to ground or instantiate a large number of variables that is proportional to the entire search space of the problem.

To remedy this limitation, we propose a distributed system, called `ros-dmapf`, that combines both imperative search-based and declarative ASP-based approaches. `ros-dmapf` first spatially partitions the MAPF graph into subgraphs. The pathing problem of agents within each subgraph is solved by a software agent running an ASP solver called *asprilo* [10]. To differentiate between the agents that are running the solvers and the agents whose paths we need to find, we will refer to the former agents as agents and the latter agents as robots from here on.

For a robot to get to its goal, it may need to traverse through several subgraphs. Therefore, `ros-dmapf` searches over the abstract graph—a graph where nodes are the subgraphs of the actual MAPF graph and vertices connect neighboring subgraphs—to find a path from the starting subgraph to the goal subgraph of each robot. The agents of these subgraphs then coordinate to identify where and when the robot should cross between subgraphs.

Our empirical results show that `ros-dmapf` scales significantly better compared to many similar systems as well as the original single-agent solver. Furthermore, the size of the subgraphs does affect the performance of `ros-dmapf`. We conclude the paper with a discussion on possible improvements of `ros-dmapf`as part of our plans for future work.

2 BACKGROUND

2.1 Multi-Agent Path Finding Problem (MAPF)

A *Multi-Agent Path Finding Problem* (MAPF) problem is given by a triple $P = (G, R, T)$, where $G = (V, E)$ is an undirected connected graph, where V and E correspond to locations and ways of moving between them for the agents; R is a set of robots; and T is a set of orders. For each robot $r \in R$, the starting location of r is specified by a location $l_r \in V$. Each order $o \in T$ is specified by a location $l_o \in V$.

Robots can move between the vertices along the edges of G, one edge at a time, under the restrictions: (a) two robots cannot swap locations in a single timestep; and (b) each location can be occupied by at most one robot at any time. A path for a robot r is a sequence of vertices $\alpha = \langle v_1, \ldots, v_n \rangle$ if (i) the robot starts at v_1; and (ii) for any two subsequent vertices v_i and v_{i+1}, there is an edge between them (i.e., $(v_i, v_{i+1}) \in E$) or they are the same vertex (i.e., $v_i = v_{i+1}$).

A robot r completes an order o via a path $\alpha = \langle v_1, \ldots, v_n \rangle$ if $l_o \in \{v_1, \ldots, v_n\}$. A *solution* of a MAPF problem P is a collection of paths $S = \{\alpha_r \mid r \in R\}$ for the robots in R so that all orders in T are completed.

2.2 Answer Set Programming (ASP)

A logic program Π is a set of rules of the form

$$a_0 \leftarrow a_1, \ldots, a_m,\ not\ a_{m+1}, \ldots,\ not\ a_n$$

where $0 \leq m \leq n$, each a_i is an atom of a propositional language and *not* represents (default) negation. An atom is of the form $p(c_1, ..., c_k)$, where p is a k-ary predicate, also written as p/k, and each c_j is a constant. Intuitively, a rule states that if all positive literals a_i are believed to be true and no negative literal $not\, a_i$ is believed to be true, then a_0 must be true. Semantically, a program induces a collection of so-called *answer sets*, which are distinguished models determined by answer sets semantics; see [11] for details.

To facilitate the use of *Answer Set Programming* (ASP) in practice, several extensions have been developed. First of all, rules with variables are viewed as shorthands for the set of their ground instances. Further language constructs include conditional literals and cardinality constraints [27]. The former are of the form $a : b_1, \ldots, b_m$, the latter can be written as $s\{d_1, \ldots, d_n\}t$, where a and b_i are possibly default negated literals, and each d_j is a conditional literal; s and t provide optional lower and upper bounds on the number of satisfied literals within the cardinality constraints. We refer to $b_1, \ldots, b_m$ as a *condition*. Aggregate functions such as *count*, *sum*, etc. have also been introduced. For example, *count* $\{X : a(X)\}$ computes the number of different objects X such that $a(X)$ is true.

2.3 ASP and MAPF

ASP has been employed effectively in planning for single and multiple agents [28] and a generalized version of MAPF [23]. The general idea is that to solve a MAPF problem $\mathcal{P}$ using ASP, we translate it to a logic program $\pi(\mathcal{P}, n)$ where n is an integer representing the maximum length (or makespan) of solutions to $\mathcal{P}$ that we are interested in. For the self-containedness of the paper, we summarize the main rules of the program $\pi(\mathcal{P}, n)$ below. Let $\mathcal{P} = (G, R, T)$ be a MAPF problem.

MAPF Input Representation. Edges and vertices in a graph G are encoded by $e(x, y)$ and $v(r)$ atoms, respectively. Agents are specified by $ag(a, l)$ atoms (a: agent identifier, l: starting location). Orders are specified by $order(i, l)$ atoms (i: order identifier, l: destination).

MAPF ASP Rules. For an integer $i \in \{0, 1, \ldots, n\}$, $st(i)$ denotes a timestep in a solution of $\mathcal{P}$. $at(r, l, s)$ encodes that "agent r is at location l at timestep s."

Action Generation. $mv(r, l, s)$ (respectively, $stay(r, l, s)$) denotes that agent r moves to (respectively, stays at) the vertex l in timestep s. At any timestep S, an agent R at location L executes exactly one action of either moving to a connected location L' ($mv(R, L', S)$) or staying at L ($stay(R, L, S)$). Rule (1) generates an action for an agent R at timestep S with this restriction:

$$1\{mv(R, L', S) : e(L, L'); stay(R, L, S)\}1 \leftarrow ag(R, \_), at(R, L, S), S < n. \quad (1)$$

The starting location of an agent is specified by:

$$at(R, L, 0) \leftarrow ag(R, L). \quad (2)$$

Rules (3) and (4) encode the effects of the action mv and $stay$ while constraints (5) and (6) forbid agents to collide or exchange their places, respectively:

$$at(R, L', S) \leftarrow at(R, L, S-1), mv(R, L', S-1), e(L, L'). \quad (3)$$

$$at(R, L, S) \leftarrow at(R, L, S-1), stay(R, L, S-1). \quad (4)$$

$$\leftarrow at(R, L, S), at(R', L, S), R \neq R'. \quad (5)$$

$$\leftarrow at(R, L, S), at(R', L', S), R \neq R', e(L, L'), at(R, L', S-1), at(R', L, S-1). \quad (6)$$

Order Allocation. Rule (7) assigns an order to an agent and constraint (8) guarantees that each agent is assigned at most one order and each order has at least one agent assigned to it.

$$1\{goal(R, T) : ag(R, \_)\}1 \leftarrow order(T, L). \quad (7)$$

$$\leftarrow goal(R, T), goal(R, T'), T \neq T'. \quad (8)$$

Solution Verification. The next rules verify that each robot has completed the order assigned to it at step n.

$$finished(R, T, S) \leftarrow goal(R, T), order(T, L), at(R, L, S). \quad (9)$$

$$finished(R, T, S+1) \leftarrow finished(R, T, S). \quad (10)$$

$$\leftarrow goal(R, T),\ not\ finished(R, T, n). \quad (11)$$

Let $\Pi(\mathcal{P}, n)$ be a program consisting of the input and Rules (1)—(11) and A be an answer set of $\Pi(\mathcal{P}, n)$. It is easy to see that for

each agent r, A must contain some atoms of the form $at(r, v_s, s)$ for each timestep $s = 0, \ldots, n$ due to Rules (2)–(4). Define $\alpha_r(A) = \langle v_0, \ldots, v_n \rangle$ where $at(r, v_j, j) \in A$ for $j = 0, \ldots, n$. It can be shown that $S = \{\alpha_r(A) \mid r \in R\}$ is a solution of $\mathcal{P}$ (see [23] for details).

2.4 Robot Operating System (ROS)

The *Robot Operating System* (ROS) is an open-source distributed framework that is geared toward building robotics systems [24]. We adopt ROS since it provides services for the development of heterogeneous clusters of software agents written in C++, Python, Lisp, etc., which ensures that `ros-dmapf` is not limited to ASP, but could potentially be used with other MAPF solvers.

For the purpose of this paper, it suffices to say that a ROS system consists of individual *ROS nodes* and the *ROS master*. Each node does not know about other nodes. For nodes to communicate, they first have to locate one another via the *ROS master*. Once they have located each other, the nodes can then use peer-to-peer communication.

There are mainly two forms of communication between nodes:

(1) In *publish/subscribe*, nodes are connected through a *topic*, which is a named bus. A node sending messages to a topic is called a *publisher*. A node listening to a topic for messages is called a *subscriber*. One node can be both a publisher and a subscriber on the same or multiple topics. A topic may have zero or more publishers and/or subscribers.
(2) In *request/response*, two nodes follow an RPC interaction through a *service*. A node that provides/calls a service is called a *service server*/*service client*. There can only be exactly one server and client at a time for the same service (subsequent service calls are put on a queue).

3 A DISTRIBUTED SYSTEM FOR SOLVING MAPF PROBLEMS

This section describes a ROS-based distributed system for solving MAPF problems. In the following, a *solver* or a *client*—or generically an agent—refers to a ROS node, unless otherwise specified. The solvers use the ASP-code in Subsection 2.3 with C++ wrapper. The description of the system focuses on its use in solving one MAPF problem. However, the system could be used for solving several related MAPF problems (e.g., when the problems share some subgraphs, robots, etc.).

3.1 Overview

Figure 2 illustrates the overall architecture of `ros-dmapf`. A map is divided into *areas* $1, 2, 3, \ldots, n$, and each area is assigned to a *sub-solver*. The division of a map can be done arbitrarily. However, a reasonable way of dividing could be helpful for the scalability of the system. We will discuss some considerations for the division of a map in Section 5. The *client* supplies the sub-solvers with orders, which are destinations for robots. A sub-solver is said to be *responsible* for a robot if the robot is located in the area assigned to the sub-solver. In essence, a sub-solver works with a *modified MAPF problem* in which the destinations of some robots in its area might not be on the map. The sub-solvers communicate with each other to find their neighboring sub-solvers and create a *neighboring map* (or *abstract map*). This map is used for computing an *abstract path* for each robot by the sub-solver that is responsible for the robot. An abstract path is a sequence of sub-solvers a robot needs to visit to reach its destination. The current implementation uses breadth-first search for computing the abstract paths.

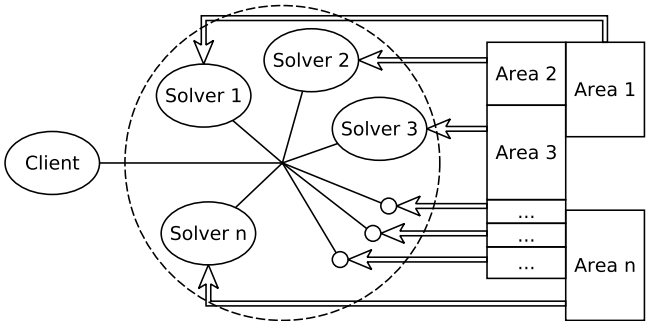

Figure 2: Overall architecture of `ros-dmapf`

To describe the system, we need some additional terminologies. Given a robot r and a sub-solver s, we say that r is a *local robot* (to s) if its destination is on the map assigned to s; otherwise, r is a *migrating robot*. The abstract map and the destination assigned by the client for a robot r will be denoted by $A(r)$ and $goal(r)$, respectively. Obviously, a migrating robot needs to go to a neighbor—following its abstract path—to reach its destination. A border location that enables r to migrate to the next sub-solver on its abstract path is called a *jump*. Each sub-solver runs the following two phases:

(1) *Abstract Planning*: In this phase, each sub-solver s repeatedly (*i*) adds the robots that it has agreed to accommodate to its set of local or migrating robots; and (*ii*) negotiates with its neighbors to identify jumps for its migrating robots. To identify a jump for a robot r, s will select the nearest free border location l to r and sends a request to the neighbor, based on the abstract map of r, indicating that r wants to migrate to a location connecting to l. The neighbor will respond with a new location l', indicating that it accepts the request and l' is the cross border location, or *false* to reject the request. At the end of each round, s will record all migrating robots that could move to its neighbors. This process stops when every sub-solver has only local robots. The result of this phase is that each sub-solver contains a *jump list* which encodes the jumps for the migrating robots at each round.
(2) *Movement Planning*: In this phase, each sub-solver executes the ASP code that is referred to in Section 2.3 to realize detailed steps of the jump list generated by the abstract planning phase.

We next describe the algorithm in more details. For a sub-solver s, let R^s denote the set of robots assigned to s by the client node; and for each round i in the abstract planning phase, L_s^i, M_s^i, and J_s^i denote the set of local robots, the migrating robots, and the set of jumps for migrating robots to/from s at i, respectively. We begin with the overall algorithm for each sub-solver (Algorithm 1):

Intuitively, Algorithm 1 implements the two phases as described earlier. Lines 3–6 initialize the necessary variables.

Lines 7–15 implements the abstract planning phase. For each round, the termination check (Lines 8–11) is executed and is done as follows: Each robot broadcasts $(|L_s^i|, |L_s^i| + |M_s^i|)$ (the number of local robots and total robots currently on its map) and waits to receive the numbers of local robots and total robots from other sub-solvers. If the total number of local robots on all sub-solvers equals the total number of robots on all sub-solvers, then the function returns true; otherwise, it returns false. It then continues with an update of local and migrating robots from the set of jumps from the solver's neighbors. Each jump is a tuple $(robot\_id, location(current\_solver), location'(next\_solver))$. Upon receiving the set of jumps from a neighbor, a sub-solver s extracts all tuples with $s = next\_solver$ and adds these tuples to its list of jumps for the current round. Algorithm 2 computes $(L_s^{i+1}, M_s^{i+1}, J_s^{i+1})$.

Lines 16–26 compute the solution of the problem. For each round i of the abstract planning phase, a MAPF problem $(L_s^i \cup M_s^i, G^s, T_s^i)$ is defined by setting, for each $r \in L_s^i \cup M_s^i$, a destination as described earlier: If $r \in M_s^i$ and $(r, l(s), l'(n)) \in J_s^{i+1}$ (see Algorithm 2), then l is the destination of r; if $r \in M_s^i$ and there is no tuple with r as first element appeared in J_s^{i+1}, then select[1] a location in G^s as the destination of r; if $r \in L_s^i$ and $goal(r)$ does not appear in the second element of any tuple in J_s^{i+1}, then $goal(r)$ is set as its destination; otherwise, select[1] a location in G^s as r's destination. We note that this is necessary to ensure that the call to a MAPF solver asks for a solution of a MAPF problem. If the computation of the plan for all rounds presented in the jump list is successful, the solver will return to the client node of `ros-dmapf` a sequence sol^s of moves where $sol^s[t]$ is of the form $mv(r, l, t)$ (or $stay(r, l, t)$). The solution to the original MAPF problem P is defined as follows: For each time step t, the actions executed by the actions of P is given by $\bigcup_{s \in \{1,\ldots,n\}} sol^s[t]$.

We note that additional bookkeeping of robots achieving their goals, the migrating actions (move between neighbors), or synchronization between sub-solvers for the horizons of the ASP computation are implemented to ensure that plans generated by the planning algorithm (Line 21, Algorithm 1) have the same length for each step in the jump list (different steps may have different plan lengths). We omit this detail due to space limitations.

To conclude the section, we would like to briefly discuss some properties of Algorithm 1. It is obvious that if the abstract planning phase cannot complete, then Algorithm 1 would never stop (and times out!). This can happen if there are deadlocks between the sub-solvers. This situation would not occur if at every negotiation cycle, the sub-solvers manage to have at least one robot moving forward on its abstract path, i.e., closer to its destination. This condition might be satisfied when the density of the robots is low (see Section 5).

Algorithm 1 solver(G^s, R^s, T^s)

1: **Input:** A modified MAPF problem (G^s, R^s, T^s)
2: **Output:** Partial solutions for the general problem
3: $L_s^0 = \{r \in R^s \mid goal(r) \in G^s\}$
4: $M_s^0 = R^s \setminus L_s^0$ and $J_s^0 = \emptyset$
5: $termination = false$
6: $i = 0$
7: **while** *not termination* **do**
8: $termination = check()$
9: **if** $termination = true$ **then**
10: break
11: **end if**
12: update J_s^i by $\bigcup_{n \in N(s)} J_n^i$
where $N(s)$ is the set of neighbors of s
13: $(L_s^{i+1}, M_s^{i+1}, J_s^{i+1})$=$ab\_plan(L_s^i, M_s^i, J_s^i, R^s, G^s)$
14: $i = i + 1$
15: **end while**
16: $sol^s = []$
17: **for** $k = 0$ to i **do**
18: $R_k = L_s^k \cup M_s^k$
19: create T_K for R_k using J_s^{k+1}
20: create a MAPF (R_k, G^s, T_k)
21: $sol_s^k = planning(R_k, G^s, T_k)$
22: **if** $sol_s^k = false$ **then**
23: **return** false
24: **end if**
25: $sol^s = sol^s \circ sol_s^k$
26: **end for**
27: **return** sol^s [sending to client node]

Algorithm 2 $ab\_plan(L, M, J, R, G)$

1: compute $X = \{r \mid r$ is migrating to the solver via $J\}$
2: $L' = L \cup \{r \in X \mid goal(r) \in G\}$
3: $J' = \emptyset$
4: $M' = M \cup X \setminus L'$
5: **for** each $r \in M'$ **do**
6: identify $l \in G$ bordering the neighbor n that r needs to go to
some neighbor l' of l that belongs to n
7: negotiate with n on l
8: **if** the negotiation success with some l' **then**
9: add to J' the tuple $(r, l(self), l'(n))$
10: **end if**
11: **end for**
12: **return** (L', M', J')

4 RELATED WORK

We now discuss related work from the MAPF literature that also draws inspiration from solving MAPF problems in a distributed manner. In the early years of MAPF research, researchers proposed a number of distributed approaches, where each robot plans its own path to get to its goal and resolves conflicts with other robots as and when these conflicts are detected. One of the pioneering algorithms in this category is *Windowed Hierarchical Cooperative A** (WHCA*) [26], which finds collision-free paths for all robots for their next window of moves. It shares the paths of all robots up to the given move limit through a reservation table, which adds a time dimension to the search space and thus results in expensive searches.

Another early key algorithm is *Flow Annotated Replanning* (FAR) [33], which combines the reservation table from WHCA*

[1] In the implementation, we use a heuristic called '*nearest destination*' that selects the nearest available location to robot r.

with flow annotations that make its searches less expensive since no time dimension has to be added to the search space. Each robot has to reserve its next moves before executing them. Robots do not incorporate these reservations into their search but simply wait until other robots that block them have moved away, similar to waiting at traffic lights. FAR attempts to break deadlocks (where several robots wait on each other indefinitely) by pushing some robots temporarily off their goal nodes. However, robots can still get stuck in some cases.

A key limitation for both WHCA* and FAR is that they are incomplete approaches. *Multi-Agent Path Planning* (MAPP) [34] tackles this limitation by proposing a more systematic approach to resolve conflicts by imposing a total ordering of all robots, resulting in completeness guarantees for a subclass of MAPF problems. Experimentally, MAPP is shown to be able to solve more problem instances than FAR, but at the cost of larger runtimes.

Chouhan and Niyogi followed up on the idea of using priority ordering to guarantee completeness for their *Distributed Multi-Agent Path Planning* (DMAPP) algorithm [3]. DMAPP consists of three steps: (*i*) Each robot finds a plan to reach its goal independently, using the FastForward planner [13] with Euclidean distances as a heuristic; (*ii*) they then decide the priority of robots based on plan lengths (longer plans have higher priorities); and (*iii*) the robot with the highest priority passes its plan to the robot with the second highest priority. This robot combines its plan with the plan received, re-plans if the plans conflict, and then sends the combined plan to the next robot, and so on. In the worst case, the lowest priority robot will have to plan for all the robots in a centralized manner, thereby significantly limiting scalability.

Sharon *et al.* generalizes this approach with their *Conflict-Based Search* (CBS) algorithm [25]. Like MAPP and DMAPP, robots in CBS also plan their individual paths to their goals. However, they resolve conflicts in an even more systematic fashion—by enumerating over the space of possible conflicts in a *conflict tree*. When a conflict is detected, the conflicting robots evaluate all the possible ways of resolving the conflict and chooses the way that results in the least overall cost. This higher-level search on the conflict tree is complete, exhaustively going through all possible conflicts in the worst case, thereby guaranteeing that CBS is also complete. Researchers have proposed a number of improvements to CBS over the years [2, 7, 8, 15, 16] as well as extensions to solve other MAPF variants [14, 17, 18, 20, 21].

A key difference between between `ros-dmapf` and the algorithms described above is that `ros-dmapf` distributes the problem *spatially* into sub-problems while the algorithms described above distributes the problem into sub-problems of *individual agents*. Nonetheless, other MAPF researchers have also investigated spatial decompositions. A key algorithm in this category is *Spatially Distributed Multiagent Planner* (SDP) [35]. It spatially divides the MAPF map into high-congestion areas (e.g., narrow passages) and low-congestion areas with an agent (called "controller" in the paper) assigned to each area. Like `ros-dmapf`, each agent is in charge of planning the paths of robots in its area and can do so independently of other areas to improve scalability. Also, like `ros-dmapf`, adjacent agents communicate with each other in identifying how robots can move from one area to the next neighboring area. However, the key differences between `ros-dmapf` and SDP are the following: (*i*) High-congestion areas in SDP cannot contain starting positions and goals of robots. In contrast, there is no such limitation in all areas in `ros-dmapf`. (*ii*) Agents in SDP plans the movement of robots between adjacent areas one robot at a time and must replan when there is a conflict. In contrast, `ros-dmapf` plans for the movement of multiple robots collectively, thereby eliminating conflicts and the need for replanning. (*iii*) Finally, SDP runs a modified version of *Cooperative A** (CA*) [26] as the underlying multi-agent path finding algorithm while `ros-dmapf` relies on ASP to find the paths.

Maps	Robots	Time (s)	Span	Moves
20 x 20	16	0.278/0.413	39	231
	32	0.279/0.846	41	453
	48	0.284/1.610	55	745
	64	0.276/2.602	71	1148
	80	0.271/3.340	68	1318
40 x 40	64	2.191/1.786	94	1770
	128	1.895/4.179	111	3613
	192	2.003/7.594	127	5954
	256	2.001/13.076	150	8857
	320	1.952/18.887	175	12077
60 x 60	144	2.217/6.432	149	5996
	288	2.415/14.809	185	12934
	432	2.001/29.361	230	21627
	576	2.018/45.256	264	30704
	720/**3**	2.343/71.045	310	43740
80 x 80	256	2.393/15.224	212	14312
	512	2.616/42.121	284	31643
	768	2.498/78.041	362	53510
	1024/**8**	2.448/146.054	433	82165
	1280/**29**	2.740/232.130	484	116210
100 x 100	400	4.366/31.702	272	28138
	800	5.080/79.964	371	62987
	1200/**2**	4.451/154.601	459	106516
	1600/**26**	4.182/253.609	538	158392
	2000/**44**	3.889/386.903	598	225924

Table 1: Experimental Results from Running 10x10 Fixed-Size Sub-Solvers on Different Map Sizes and Robot Densities

5 EXPERIMENTS

We conduct three sets of experiments to (*i*) evaluate the scalability of `ros-dmapf`; (*ii*) evaluate the impact of the size of the subgraphs on the performance of `ros-dmapf`; and (*iii*) compare `ros-dmapf` with other MAPF solvers.

Inspired by *asprilo*, we generate gird-based, obstacle-free, and rectangular maps in our experiments.[2] The robots have distinct starting and goal locations that are randomly generated. The experiments are run on a laptop with an Intel Core i7-7700HQ processor and 16 GB of memory with ROS Kinetic on Ubuntu 16.04.

Table 1 shows the results of `ros-dmapf` when subgraphs are of the size 10x10. The first column shows the map size. The second column shows the numbers of robots, which are 4%, 8%, 12%, 16%, and 20% of the respective map size, and the number of instances,

[2]We would like to use the problem generator of *asprilo* but its code does not generate instances large enough for the experiments.

Solvers	288 robots		576 robots	
	Time (s)	Span	Time (s)	Span
20	1.903/35.085	210	1.769/181.429	286
25	2.037/24.496	201	2.840/91.471	313
30	1.864/21.587	217	2.512/79.326	307
36	1.882/15.409	188	2.540/46.170	267
45	2.743/15.295	177	1.359/35.142	269
60	1.981/11.290	196	3.092/27.728	283
75	2.857/11.168	181	3.884/25.112	253
90	4.810/13.404	224	6.154/26.296	308
100	6.292/9.5560	167	5.863/22.671	215

Table 2: Experimental Results from Running Varying-Size Sub-Solvers on a 60x60 Map with 288 and 576 Robots

indicated by a boldface number following '/', that `ros-dmapf` was not able to find a solution. The third column shows runtime in seconds spent on setup/solving periods. The setup period is when all the clients and sub-solvers are launched and waiting for ROS to set up their communication channels. The solving period is when `ros-dmapf` is working on solving the problem. The fourth column shows the makespan. The last column shows the combined total number of moves of all the robots. We generate 5 problem instances for each pair of map and number of robots, and run each instance 10 times to give the averaged result of 50 runs in each row.

Table 1 shows that: (*i*) the setup times stay relatively the same in the same map size even with different number of robots and only increase slightly as the map gets bigger; (*ii*) the solving times increase in relatively the same rate as the the number of robots, impervious to map size; and (*iii*) the makespan and the number of moves tend to grow faster in bigger maps than in smaller ones.

Table 2 shows the results of varying the size of subgraphs on two problem instances—60x60 map with 288 and 576 robots. We choose these instances for the experiment as they seem, from Table 1, reasonably difficult for `ros-dmapf`. The first column shows the number of sub-solvers, which come from varying the size of subgraphs to 15x12, 12x12, 15x8, 10x10, 10x8, 10x6, 8x6, 10x4, and 6x6, respectively. The second and the last column show runtime (setup/solving) in seconds and makespan averaged from 10 runs on the two problem instances, respectively. The number of moves are similar among the different numbers of sub-solvers and, so, we omit them for brevity. Table 2 shows that the size of subgraphs affects runtime and makespan, and being able to identify a good size of subgraphs relative to a problem might lead to significant performance improvement.

Table 3 shows comparisons between `ros-dmapf`,[3] WHCA*, *asprilo*, and CBSH-RM in terms of runtime in seconds, makespan, and the total number of moves, averaged from 10 runs on each problem instance. Among these algorithms, `ros-dmapf` and WHCA* are suboptimal, while *asprilo* and CBSH-RM are optimal MAPF solvers. We choose these systems because they are a good representative of state-of-the-art approaches and are available to download and compile on our machine. We use an improved implementation of WHCA* that allows diagonal moves, and choose the window size to be 8 because it gives a good tradeoff between runtime and solution quality in our experiments. asprilo [10] is an ASP encoding for robotic intra-logistics domain. We use the move-only domain of asprilo with a slight modification that assigns each robot to a specific goal. CBSH-RM [16] is the most recent improved version of CBS [25].

The results show that `ros-dmapf` has the best scalability with competitive number of moves and acceptable makespan. The disparity of the quality between the number of moves and makespan tells us that some robots had to spend more time than the others just to wait to migrate. WHCA* is the second best in terms of scalability and also gives the best makespan in a few cases because diagonal moves are allowed. However, the runtime and solution quality get worse much quicker than `ros-dmapf` as the problem gets bigger. In some cases (e.g., 40x40(192)), it gives the largest number of moves. *asprilo* is the slowest in most cases and also seems to be mostly affected by map size, but it is optimal and does not get worse as fast as CBSH-RM. This is noticeable by the difference in runtime between the 40x40(64) and 40x40(128) problem instances. CBSH-RM performs extremely fast on small problem instances, but has the worst scalability, demonstrated by the difference in runtime between the 40x40(64) and 40x40(128) problem instances.

[3]We only consider the solving time of `ros-dmapf`.

6 CONCLUSIONS AND FUTURE WORK

We have presented an initial version of `ros-dmapf`, a distributed MAPF solver, where empirical results show that it has promising scalability. To the best of our knowledge, `ros-dmapf` is one of the first MAPF solvers in which the components are distributed and work together with a high degree of parallelism. In the future, we plan to (*i*) identify a better way to handle dense maps; (*ii*) investigate a new implementation of the sub-solvers using multi-shot ASP [9]; and (*iii*) experiment with `ros-dmapf` on maps with obstacles.

ACKNOWLEDGMENTS

This research is partially supported by NSF grants 1345232, 1619273, 1757207, and 1812628. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, or the U.S. government.

REFERENCES

[1] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Solomon Eyal Shimony. 2015. ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding. In *International Joint Conference on Artificial Intelligence (IJCAI)*. 740–746.

[2] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Solomon Eyal Shimony. 2015. ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding. In *International Joint Conference on Artificial Intelligence (IJCAI)*. 740–746.

[3] Satyendra Singh Chouhan and Rajdeep Niyogi. 2015. DMAPP: A Distributed Multi-agent Path Planning Algorithm. In *Australasian Joint Conference on Artificial Intelligence (AI)*. 123–135.

[4] Liron Cohen, Tansel Uras, T. K. Satish Kumar, Hong Xu, Nora Ayanian, and Sven Koenig. 2016. Improved Solvers for Bounded-Suboptimal Multi-Agent Path Finding. In *International Joint Conference on Artificial Intelligence (IJCAI)*. 3067–3074.

[5] Boris de Wilde, Adriaan ter Mors, and Cees Witteveen. 2014. Push and Rotate: a Complete Multi-Agent Pathfinding Algorithm. *Journal of Artificial Intelligence Research (JAIR)* 51 (2014), 443–492.

[6] Esra Erdem, Doga Gizem Kisa, Umut Öztok, and Peter Schüller. 2013. A General Formal Framework for Pathfinding Problems with Multiple Agents. In *AAAI Conference on Artificial Intelligence*. 290–296.

Maps	Robots	ros-dmapf			WHCA*			asprilo			CBSH-RM		
		Time (s)	Span	Moves	Time (s)	Span	Moves	Time (s)	Span	Moves	Time (s)	Span	Moves
20 x 20	16	0.399	35	204	0.207	19	140	3.203	24	209	0	24	197
	32	0.900	44	434	0.373	18	272	10.239	28	399	0.001	28	386
	48	1.817	53	775	0.882	27	659	14.190	27	703	0.004	27	673
	64	2.340	66	1037	1.998	56	1475	22.451	29	890	-	-	-
	80	3.108	71	1283	1.250	36	1183	29.220	29	1106	-	-	-
40 x 40	64	1.864	87	1873	8.407	45	1503	-	-	-	0.006	53	1772
	128	4.115	114	3739	24.255	74	4564	-	-	-	45.307	63	3391
	192	7.669	124	5855	65.153	104	11113	-	-	-	-	-	-
	256	13.238	146	8687	-	-	-	-	-	-	-	-	-
	320	18.005	175	12409	-	-	-	-	-	-	-	-	-
60 x 60	144	6.401	146	5876	84.773	89	6310	-	-	-	0.034	87	5621
	288	15.306	185	12677	-	-	-	-	-	-	-	-	-
	432	26.658	222	21197	-	-	-	-	-	-	-	-	-
	576	45.655	266	30667	-	-	-	-	-	-	-	-	-
	720	64.186	304	41668	-	-	-	-	-	-	-	-	-

Table 3: Comparisons between ros-dmapf, WHCA*, asprilo, and CBSH-RM [timeout is set to 100 seconds]

[7] Ariel Felner, Jiaoyang Li, Eli Boyarski, Hang Ma, Liron Cohen, T. K. Satish Kumar, and Sven Koenig. 2018. Adding Heuristics to Conflict-Based Search for Multi-Agent Path Finding. In *International Conference on Automated Planning and Scheduling (ICAPS)*. 83–87.

[8] Graeme Gange, Daniel Harabor, and Peter J. Stuckey. 2019. Lazy CBS: Implicit Conflict-Based Search Using Lazy Clause Generation. In *International Conference on Automated Planning and Scheduling (ICAPS)*. 155–162.

[9] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2019. Multi-Shot ASP Solving with Clingo. *Theory and Practice of Logic Programming (TPLP)* 19, 1 (2019), 27–82.

[10] Martin Gebser, Philipp Obermeier, Thomas Otto, Torsten Schaub, Orkunt Sabuncu, Van Nguyen, and Tran Cao Son. 2018. Experimenting with Robotic Intra-Logistics Domains. *Theory and Practice of Logic Programming (TPLP)* 18, 3-4 (2018), 502–519.

[11] Michael Gelfond and Vladimir Lifschitz. 1988. The Stable Model Semantics for Logic Programming. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium (ICLP/SLP)*. 1070–1080.

[12] Meir Goldenberg, Ariel Felner, Roni Stern, Guni Sharon, Nathan R. Sturtevant, Robert C. Holte, and Jonathan Schaeffer. 2014. Enhanced Partial Expansion A*. *Journal of Artificial Intelligence Research (JAIR)* 50 (2014), 141–187.

[13] Jörg Hoffmann and Bernhard Nebel. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research (JAIR)* 14 (2001), 253–302.

[14] Wolfgang Hönig, Scott Kiesel, Andrew Tinka, Joseph W. Durham, and Nora Ayanian. 2018. Conflict-Based Search with Optimal Task Assignment. In *International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. 757–765.

[15] Jiaoyang Li, Daniel Harabor, Peter J. Stuckey, Hang Ma, and Sven Koenig. 2019. Disjoint Splitting for Multi-Agent Path Finding with Conflict-Based Search. In *International Conference on Automated Planning and Scheduling (ICAPS)*. 279–283.

[16] Jiaoyang Li, Daniel Harabor, Peter J. Stuckey, Hang Ma, and Sven Koenig. 2019. Symmetry-Breaking Constraints for Grid-Based Multi-Agent Path Finding. In *AAAI Conference on Artificial Intelligence*. 6087–6095.

[17] Jiaoyang Li, Pavel Surynek, Ariel Felner, Hang Ma, T. K. Satish Kumar, and Sven Koenig. 2019. Multi-Agent Path Finding for Large Agents. In *AAAI Conference on Artificial Intelligence*. 7627–7634.

[18] Minghua Liu, Hang Ma, Jiaoyang Li, and Sven Koenig. 2019. Task and Path Planning for Multi-Agent Pickup and Delivery. In *International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. 1152–1160.

[19] Ryan Luna and Kostas E. Bekris. 2011. Push and Swap: Fast Cooperative Path-Finding with Completeness Guarantees. In *International Joint Conference on Artificial Intelligence (IJCAI)*. 294–300.

[20] Hang Ma, Wolfgang Hönig, T. K. Satish Kumar, Nora Ayanian, and Sven Koenig. 2019. Lifelong path planning with kinematic constraints for multi-agent pickup and delivery. In *AAAI Conference on Artificial Intelligence*, Vol. 33. 7651–7658.

[21] Hang Ma, Glenn Wagner, Ariel Felner, Jiaoyang Li, T. K. Satish Kumar, and Sven Koenig. 2018. Multi-Agent Path Finding with Deadlines. In *International Joint Conference on Artificial Intelligence (IJCAI)*. 417–423.

[22] Robert Morris, Corina S. Pasareanu, Kasper Søe Luckow, Waqar Malik, Hang Ma, T. K. Satish Kumar, and Sven Koenig. 2016. Planning, Scheduling and Monitoring for Airport Surface Operations. In *AAAI Workshop on Planning for Hybrid Systems*.

[23] Van Nguyen, Philipp Obermeier, Tran Cao Son, Torsten Schaub, and William Yeoh. 2017. Generalized Target Assignment and Path Finding Using Answer Set Programming. In *International Joint Conference on Artificial Intelligence (IJCAI)*. 1216–1223.

[24] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. 2009. ROS: an Open-source Robot Operating System. In *ICRA Workshop on Open Source Software in Robotics*.

[25] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. 2015. Conflict-Based Search for Optimal Multi-Agent Pathfinding. *Artificial Intelligence (AIJ)* 219 (2015), 40–66.

[26] David Silver. 2005. Cooperative Pathfinding. In *Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*. 117–122.

[27] Patrik Simons, Ilkka Niemelä, and Timo Soininen. 2002. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence (AIJ)* 138, 1-2 (2002), 181–234.

[28] Tran Cao Son and Marcello Balduccini. 2018. Answer Set Planning in Single- and Multi-Agent Environments. *Künstliche Intelligenz (KI)* 32, 2-3 (2018), 133–141.

[29] Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma andw Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Roman Barták, and Eli Boyarski. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *International Symposium on Combinatorial Search (SOCS)*. 151–159.

[30] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. 2016. Efficient SAT Approach to Multi-Agent Path Finding Under the Sum of Costs Objective. In *European Conference on Artificial Intelligence (ECAI)*. 810–818.

[31] Manuela M. Veloso, Joydeep Biswas, Brian Coltin, and Stephanie Rosenthal. 2015. CoBots: Robust Symbiotic Autonomous Mobile Service Robots. In *International Joint Conference on Artificial Intelligence (IJCAI)*. 4423–4429.

[32] Glenn Wagner and Howie Choset. 2015. Subdimensional Expansion for Multi-robot Path Planning. *Artificial Intelligence (AIJ)* 219 (2015), 1–24.

[33] Ko-Hsin Cindy Wang and Adi Botea. 2008. Fast and Memory-Efficient Multi-Agent Pathfinding. In *International Conference on Automated Planning and Scheduling (ICAPS)*. 380–387.

[34] Ko-Hsin Cindy Wang and Adi Botea. 2011. MAPP: a Scalable Multi-Agent Path Planning Algorithm with Tractability and Completeness Guarantees. *Journal of Artificial Intelligence Research (JAIR)* 42 (2011), 55–90.

[35] Christopher Makoto Wilt and Adi Botea. 2014. Spatially Distributed Multiagent Path Planning. In *International Conference on Automated Planning and Scheduling (ICAPS)*. 332–340.

[36] Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. 2008. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. *AI Magazine* 29, 1 (2008), 9–20.

[37] Jingjin Yu and Steven M. LaValle. 2016. Optimal Multirobot Path Planning on Graphs: Complete Algorithms and Effective Heuristics. *IEEE Transactions on Robotics (T-RO)* 32, 5 (2016), 1163–1177.