A Distributed Solver for Multi-Agent Path Finding Problems

Poom Pianpak, Tran Cao Son, Z. O. Toups
{ppianpak, tson2}@cs.unm.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 optimality, performance, or a tradeoff between them. However, there is not much work on solving them in a distributed manner, and even less through 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 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 the system against our 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.

KEYWORDS

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

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 sub-optimal 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-dnapf 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-dnapf 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-dnapf. We conclude the paper with a discussion on possible improvements of ros-dnapf 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 \( \alpha = (v_1, \ldots, v_n) \) 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 (ie, \( (v_i, v_{i+1}) \in E \)) or they are the same vertex (ie, \( v_i = v_{i+1} \)).

A robot \( r \) completes an order \( o \) via a path \( \alpha = (v_1, \ldots, v_n) \) 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 \( \Pi \) is a set of rules of the form
\[
an_0 \leftarrow a_1, \ldots, a_m, \text{ not } a_{m+1}, \ldots, \text{ not } a_n
\]
where \( 0 \leq m \leq n \), each \( a_i \) is an atom of a propositional language and \( \text{not} \) represents (default) negation. An atom is of the form \( p(c_1, \ldots, c_k) \), 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 \( \text{not} a_i \) 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 form \( a : b_1, \ldots, b_m \), the latter can be written as \( s(d_1, \ldots, d_n) t \), where \( a \) and \( b_1 \) 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 \( \text{count} \), \( \text{sum} \), etc. are also introduced. For example, \( \text{count}(X : a(X)) \) 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 multi-agent systems [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 \( \text{ag}(a, l) \) atoms (\( a \): agent identifier, \( l \): starting location). Orders are specified by \( \text{order}(i, l) \) atoms (\( i \): order identifier, \( l \): destination).

MAPF ASP Rules. For an integer \( i \in \{0, 1, \ldots, n\} \), \( \text{sta}(i) \) 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, \( \text{stay}(r, l, s) \)) 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 \( l' \) (\( \text{mv}(r, l', s) \)) or staying at \( l \) (\( \text{stay}(r, l, s) \)). The next rule generates an action for an agent \( r \) at timestep \( s \) with this restriction:

\[
\begin{align*}
1 \{ \text{mv}(r, l', s) \mid (e(L, L'), \text{stay}(r, L, s)) \} & \leftarrow \text{ag}(R, s), at(r, L, S) < n. \\
\text{The starting location of an agent is specified by:} & \\
& at(r, L, 0) \leftarrow \text{ag}(R, L).
\end{align*}
\]

The next two rules encode the effects of the action \( \text{mv} \) and \( \text{stay} \) and the constraints forbidding agents to collide or exchange their places:

\[
\begin{align*}
& at(r, L', S) \leftarrow at(r, L, S-1), \text{mv}(r, L', S-1), e(L, L'). \\
& at(r, L, S) \leftarrow at(r, L, S-1), \text{stay}(r, L, S-1). \\
& at(r, L', S) \leftarrow at(r, L', S-1), \text{stay}(r', L', S), R \neq R'. \\
& at(r, L, S) \leftarrow at(r', L', S), R \neq R', e(L', L'). \\
& at(r, L, S-1) \leftarrow at(r', L', S), R \neq R', e(L', L').
\end{align*}
\]

Order Allocation. The next rule assigns an order to an agent and guaranteed that each agent is assigned at most one order and each order has at least one agent assigned to it.

\[
\begin{align*}
& \{ \text{goal}(R, T) \mid \text{ag}(R, _) \} & \leftarrow & \text{order}(T, L). \\
& \text{goal}(R, T), \text{goal}(R, T') \leftarrow & \text{order}(T, L), T \neq T'.
\end{align*}
\]

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

\[
\begin{align*}
& \text{finished}(R, T, S) \leftarrow \text{goal}(R, T), \text{order}(T, L), at(R, L, S). \\
& \text{finished}(R, T, S + 1) \leftarrow \text{finished}(R, T, S). \\
& \text{not} \text{finished}(R, T, n). \\
\end{align*}
\]

Let \( \Pi(\mathcal{P}, n) \) be the 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 $A_r(A) = (v_0, \ldots, v_n)$ where $at(r, v_j, j) \in A$ for $j = 0, \ldots, n$. It can be shown that $S = \{A_r(A) \mid r \in R\}$ is a solution of $P$ (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 2.1 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 the 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.

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 a robot 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 iteration, $s$ 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 iteration.

(2) Movement Planning: In this phase, each sub-solver executes the ASP code that is referred to in Section 2.3 to realize the 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 iteration $i$ in the abstract planning phase, $L_i^s$, $M_i^s$, and $J_i^s$ 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.
We now discuss related work from the MAPF literature that also 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 \((robot_id, location(current_soler), location(next_soler))\). Upon receiving the set of jumps from a neighbor, a sub-solver \(s\) extracts all tuples with \(s = next_soler\) and adds these tuples to its list of jumps for the current iteration. Algorithm 2 computes \((L^s_{t+1}, M^s_{t+1}, J^s_{t+1})\).

Lines 16–26 compute the solution of the problem. For each iteration \(i\) of the abstract planning phase, a MAPF problem \((L_i^s \cup M_i^s, G^s, T_i^s)\) is defined by setting, for each \(r \in L_i^s \cup M_i^s\), a destination as described earlier: If \(r \in M_i^s\) and \((r, l(s), l'(n)) \in J^s_i\), then \(r \) is the destination of \(r\); if \(r \in L_i^s\) and there is no tuple with \(r\) as first element appeared in \(J^s_i\), then \(r\) is chosen as its destination; otherwise, \(r\) is chosen as its 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 iterations presented in the jump list is successful, the solver will return to the client node of \(ros\-\)map a sequence \(sol^s\) of moves where \(sol^s[t]\) is of the form \(move(r, l, t)\) (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 \(\bigcup_{t \in \{1, ..., n\}} sol^s[t]\).

We note that additional book-keeping of robots achieving their goals, the migrating actions (move between neighbors), or synchronization between sub-solvers for the horizon of the ASP computation are implemented to ensure that the 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).

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 \textit{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 \textit{Flow Annotated Replanning} (FAR) [33], which combines the reservation table from WHCA*
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 it executes 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 (MAPF) [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, MAPF 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. generalize their 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.

## 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 grid-based, obstacle-free, and rectangular maps in our experiments. We would have used asprilo but the its code does not generate instances large enough for the 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.

<table>
<thead>
<tr>
<th>Maps</th>
<th>Robots</th>
<th>Time(s)</th>
<th>Span</th>
<th>Moves</th>
</tr>
</thead>
<tbody>
<tr>
<td>20x20</td>
<td>16</td>
<td>0.278/0.413</td>
<td>39</td>
<td>231</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>0.279/0.846</td>
<td>41</td>
<td>453</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>0.284/1.610</td>
<td>55</td>
<td>745</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>0.276/2.602</td>
<td>71</td>
<td>1148</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>0.271/3.340</td>
<td>68</td>
<td>1318</td>
</tr>
<tr>
<td>40x40</td>
<td>128</td>
<td>1.895/4.179</td>
<td>111</td>
<td>3613</td>
</tr>
<tr>
<td></td>
<td>192</td>
<td>2.003/7.594</td>
<td>127</td>
<td>5954</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>2.001/13.076</td>
<td>150</td>
<td>8857</td>
</tr>
<tr>
<td></td>
<td>320</td>
<td>1.952/18.887</td>
<td>175</td>
<td>12077</td>
</tr>
<tr>
<td>60x60</td>
<td>144</td>
<td>2.217/6.432</td>
<td>149</td>
<td>5996</td>
</tr>
<tr>
<td></td>
<td>288</td>
<td>2.415/14.809</td>
<td>185</td>
<td>12934</td>
</tr>
<tr>
<td></td>
<td>432</td>
<td>2.001/29.361</td>
<td>230</td>
<td>21627</td>
</tr>
<tr>
<td></td>
<td>576</td>
<td>2.018/45.256</td>
<td>264</td>
<td>30704</td>
</tr>
<tr>
<td></td>
<td>720/3</td>
<td>2.343/71.045</td>
<td>310</td>
<td>43740</td>
</tr>
<tr>
<td>80x80</td>
<td>256</td>
<td>2.393/15.224</td>
<td>212</td>
<td>14312</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>2.616/42.121</td>
<td>284</td>
<td>31643</td>
</tr>
<tr>
<td></td>
<td>768</td>
<td>2.498/78.041</td>
<td>362</td>
<td>55510</td>
</tr>
<tr>
<td></td>
<td>1024/8</td>
<td>2.448/146.054</td>
<td>433</td>
<td>82165</td>
</tr>
<tr>
<td></td>
<td>1280/29</td>
<td>2.740/232.130</td>
<td>484</td>
<td>116210</td>
</tr>
<tr>
<td>100x100</td>
<td>400</td>
<td>4.366/31.702</td>
<td>272</td>
<td>28138</td>
</tr>
<tr>
<td></td>
<td>800</td>
<td>5.080/79.964</td>
<td>371</td>
<td>62987</td>
</tr>
<tr>
<td></td>
<td>1200/2</td>
<td>4.451/154.601</td>
<td>459</td>
<td>106516</td>
</tr>
<tr>
<td></td>
<td>1600/26</td>
<td>4.182/253.609</td>
<td>538</td>
<td>158392</td>
</tr>
<tr>
<td></td>
<td>2000/44</td>
<td>3.889/386.903</td>
<td>598</td>
<td>225924</td>
</tr>
</tbody>
</table>

Table 1: Experimental Results from Running 10x10 Fixed-Size Sub-Solvers on Different Map Sizes and Robot Densities
We choose these systems because they are a good representative of WHCA* that allows diagonal moves, and choose the window total number of moves, averaged from 10 runs on each problem instance. Among these algorithms, ros-dmapf 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.

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


<table>
<thead>
<tr>
<th>Solvers</th>
<th>Time (s)</th>
<th>Span</th>
<th>Time (s)</th>
<th>Span</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1.903/35.085</td>
<td>210</td>
<td>1.769/181.429</td>
<td>286</td>
</tr>
<tr>
<td>25</td>
<td>2.037/24.496</td>
<td>201</td>
<td>2.840/91.471</td>
<td>313</td>
</tr>
<tr>
<td>35</td>
<td>1.882/15.409</td>
<td>188</td>
<td>2.540/46.170</td>
<td>267</td>
</tr>
<tr>
<td>40</td>
<td>2.743/15.295</td>
<td>177</td>
<td>1.359/35.142</td>
<td>269</td>
</tr>
<tr>
<td>50</td>
<td>1.981/11.290</td>
<td>196</td>
<td>3.092/27.728</td>
<td>283</td>
</tr>
<tr>
<td>75</td>
<td>2.857/11.168</td>
<td>181</td>
<td>3.884/25.112</td>
<td>253</td>
</tr>
<tr>
<td>90</td>
<td>4.810/13.404</td>
<td>224</td>
<td>6.154/26.296</td>
<td>308</td>
</tr>
<tr>
<td>100</td>
<td>6.292/9.5560</td>
<td>167</td>
<td>5.863/22.671</td>
<td>215</td>
</tr>
</tbody>
</table>

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

and 20% of the respective map size, and the number of instances, 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 subgraphs, 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 it might lead to significant performance improvement.

Table 3 shows comparisons between ros-dmapf, 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 sub-optimal, 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 have 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.
Table 3: Comparisons between ros-dmapf, WHCA*, aspipro, and CBSH-RM [timeout set to 100 seconds]