Improving the Performance of Multi-Robot Systems by Task Switching

Cynthia Sung, Nora Ayanian, Daniela Rus

Abstract— **We consider the problem of task assignment for a multi-robot system where each robot must attend to one or more queues of tasks. We assume that individual robots have no knowledge of tasks in the environment that are not in their queue. Robots in communication with each other may share information about active tasks and exchange queues to achieve lower cost for the system. We show that allowing this kind of task switching causes tasks to be completed more efficiently. In addition, we present conditions under which queues can be guaranteed to make progress, and we support these claims with simulation and experimental results. This work has potential applications in manufacturing, environmental exploration, and pickup-delivery tasks.**

I. Introduction

Efficient coordination is a requirement for success in many applications of multi-robot systems. In general, a complex task given to a multi-robot system consists of a sequence of simpler subtasks that can be allocated to individual robots. For example, a multi-robot system assembling a bookcase must execute part and tool delivery subtasks (e.g. bring shelves, screws, and screw drivers to the assembly location), then perform grasping, manipulation, and assembly subtasks (e.g. add the next shelf to the subassembly). To be robust against failures, allocation of these tasks to robots in the system must occur in a distributed way.

A large body of literature exists that addresses the problem of decentralized task allocation. For instance, considerable work in dynamic vehicle routing [1]–[3] is concerned with determining least-cost assignments for multiple vehicles servicing stochastic demands. The most popular approach is a divide-and-conquer [1] approach, in which the vehicles partition the environment among themselves, and each vehicle individually attends to the tasks in its own region. This approach has the added benefit that since each vehicle stays within its own region, the challenge of inter-vehicle collision avoidance is averted. Although the divide-and-conquer approach is decentralized in that no negotiation about task assignment between vehicles is necessary once the partitioning has been decided, it requires a vehicle to be aware of every demand that appears within its region, which is not always feasible.

Support for this project has been provided in part by the Future Urban Mobility project of the Singapore-MIT Alliance for Research and Technology (SMART) Center, with funding from Singapore's National Research Science Foundation, by the Foxconn Company, by ONR MURI Grants N00014-09-1-1051 and N00014-09-1-1031, and by NSF award IIS-1117178. C. Sung was supported by the Department of Defense (DoD) through the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program.

C. Sung, N. Ayanian, and D. Rus are with the Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA emails: {crsung, nayanian, rus}@csail.mit.edu

For example, consider a door-to-door mail delivery system (e.g., [4]). Packages are left at various locations in an environment, with their destinations specified. Robots in the process of delivering a package (completing a task) may find new packages awaiting delivery. The destinations of these new packages correspond to new task locations, but those locations may not be inside the region of the robot who discovered the task. If the robot responsible for the region containing the destination is not within communicating distance, the discovering robot has no way to inform the robot responsible for the task or to hand over the package. In this case, the divide-and-conquer approach cannot solve the problem. Similar examples can be constructed for applications such as manufacturing, environmental monitoring, and transportation.

The key challenge of coordination for complex multi-robot tasks in these applications is that robots have only partial knowledge of the tasks, which limits their ability to satisfy the global objective. Market-based approaches [5], [6] have recently gained attention as a method of dealing with this challenge. As individual robots come into contact, they bid for the privilege of performing a task and thereby reduce the total system cost. For the most part, however, the cost of performing a task can only be estimated, and the resulting inaccurate bids may lead to situations where tasks cannot be completed.

This paper presents an approach to distributed task assignment in a perfectly known environment, where individual robots with partial knowledge about the locations of tasks can exchange tasks with other robots that they encounter. A similar approach is used in [7], where tasks can be exchanged among communicating robots in partially known environments. While that work ensures that all tasks are completed, the results are restricted to situations where every robot has exactly one task. It does not allow robots to serve queues or multiple tasks, which could result in some tasks being ignored. This paper contributes the following:

- a decentralized task-switching algorithm that extends the results in [7] to the case where robots have queues of tasks (rather than single tasks), including infinite queues,
- the introduction of an aging function for timelier task completion,
- conditions for guarantees of forward progress in every task queue, and
- experimental evaluation across various switching policies and discussion of the cost-computation tradeoff.

The outline of the paper is as follows. Section II formally defines the problem and objectives. Section III summarizes the general approach towards task assignment, our switching policy, and control. In Sections IV and V, we provide guarantees on task completion and forward progress in the system. Finally, Section VI discusses simulation results of various switching policies.

II. Definitions and Problem Statement

The problem formulation and notation used in this paper builds on [7] and are illustrated in Fig. 1. Specifically, consider a bounded, connected workspace $\mathcal{W} \subset \mathbb{R}^d$. The workspace contains a team of N agents (robots) $\mathcal{V}_A = \left\{a^i | i = 1, \ldots, N\right\}$ with state

$$\mathbf{x} = [\mathbf{x}_1^\intercal \; \mathbf{x}_2^\intercal \; \cdots \; \mathbf{x}_N^\intercal]^\intercal, \; \mathbf{x}_i = [x_i \; y_i \; z_i \; \cdots]^\intercal \in \mathbb{R}^d$$

and dynamics

$$\dot{\mathbf{x}}_i = \mathbf{u}_i$$

We assume agents can accurately localize themselves in the environment. The agents must collectively perform a collection of tasks $\mathcal{Q}$ in the form of N queues $\mathcal{Q} = \{Q^k | k = 1, \ldots, N\}$.

The *communication graph* on the team of agents is a dynamic undirected graph $G_N = (\mathcal{V}_A, \mathcal{E}_N)$ where $\mathcal{E}_N = \left\{(a^i, a^j) | a^i \text{ and } a^j \text{ can communicate}\right\}$. As in [7], we assume that the workspace $\mathcal{W}$ has been tessellated into a finite number of convex, non-overlapping polytopes p^m such that any polytope and its neighbor intersect along exactly one hyperplane. This allows us to represent the environment as a graph $G^p = (\mathcal{V}^p, \mathcal{E}^p)$ where $\mathcal{V}^p = \{p^1, p^2, \ldots\}$ are the polytopes and $\mathcal{E}^p = \{(p^m, p^{m'}) | p^m \text{ shares a facet with } p^{m'}\}$. Each edge is associated with a positive weight. A pair of agents in a pair of polytopes $(p^m, p^{m'})$ can communicate if an agent at any position in p^m would be able to communicate with an agent at any position in $p^{m'}$. Therefore, the polytopes must be sufficiently small that agents in adjacent polytopes are able to communicate; the polytopes can easily be subdivided if this is not the case.

A set of agents form a *group* $\mathcal{G} \subseteq \mathcal{V}_A$ if the subgraph of G_N induced by $\mathcal{G}$ is connected. Agents must be in the same group to share information about task locations. Finally, we assume all agents are identical and, specifically, that they are all equally able to perform any task.

A *task queue* is a list of positions $Q^k = \{q_1^k, \ldots, q_{n^k}^k\}$, $n^k > 0$, in $\mathcal{W}$ that must be visited in order. We say task q_b^k is *active* if it is currently at the front of the queue, and we let $\sigma^k \in \{1, \ldots, n^k\}$ be the queue's state (i.e., the current position in the queue, or the current value of b). An active task q_b^k is *completed* if the agent assigned to Q^k enters the ϵ-neighborhood of q_b^k, for some small characteristic $\epsilon > 0$ of the workspace, at which time the next task q_{b+1}^k becomes active. Each agent is aware of only the currently active tasks in its group, and has no knowledge of any tasks that appear later in the queue or that belong to other agents outside of communication range. Finally, agents who have completed their assigned task queues remain at the site of the last task. This can be achieved, for example, by causing the last task in the queue to remain active forever despite being completed. For succinctness, we say that the last task in a queue is *final*.

We define a *task assignment* as a bijective function $\pi : \mathcal{Q} \to \mathcal{V}_A$ between the task queues and the agents. We say agent a^i is *assigned* to Q^k if $\pi\left(Q^k\right) = a^i$ and correspondingly that Q^k *belongs* to a^i. Our goal is to find a task assignment that minimizes the cost (time, distance traveled, etc.) of completing the tasks.

In order to simplify our problem and decouple task assignment from control, we use a cost function that is independent of the equations of motion. Let $\mathcal{P}^i(q_b^k)$ be the path in the polytope graph G^p that agent a^i has traveled during the time it has been assigned task q_b^k, and let $w^i(q_b^k)$ be the sum of weights of the edges traversed along that path. The total distance that *all* agents assigned to q_b^k have traveled is $w_b^k = \sum_{i=1}^N w^i(q_b^k)$. For a task that is not yet active, $w_b^k \equiv 0$. Then, we define a cost function

$$\text{cost} = \frac{\text{total travel weight}}{\text{number of tasks}} = \frac{\sum_{k=1}^N \sum_{b=1}^{n^k} w_b^k}{\sum_{k=1}^N n^k} \qquad (1)$$

which is the eventual mean travel weight per task once all tasks have been completed. Minimizing this value is equivalent to minimizing the total travel weight for all agents to complete all the tasks.

The problem addressed in this paper is as follows:

Problem 2.1 (Finite Queue): For a given initial state $\mathbf{x}_0$, find a task assignment π, which may vary with time, such that:

(a) all tasks are completed in finite time and
(b) the cost as defined in Eq. (1) is minimized.

Note that due to the task completion condition 2.1(a), this optimization problem is only well-defined when all queues are of finite length. For the case of an infinite queue, we instead consider queues' progress. We say a queue is *making progress* if the currently active task in the queue is completed in a finite amount of time. If this is not the case, we say the queue has *stalled*.

The cost function given in Eq. (1) also requires task completion. In order to translate this function into the infinite queues case, we define instead a running cost

$$\text{cost}_r(t) = \frac{\sum_{k=1}^N \sum_{b=1}^{\sigma^k} w_b^k}{\sum_{k=1}^N \sigma^k} \qquad (2)$$

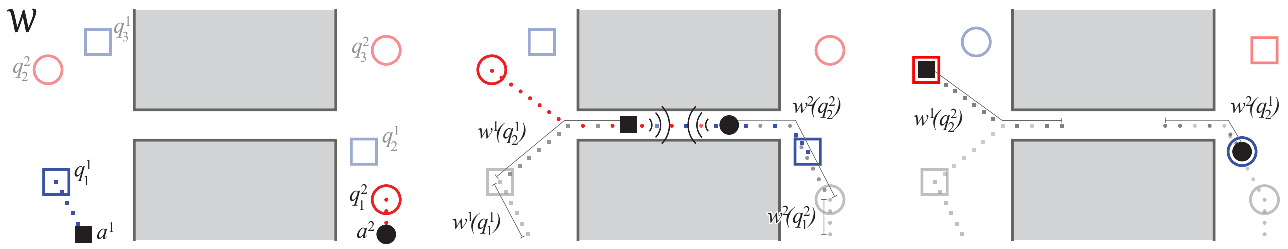
and define the infinite queues cost function to be

$$\text{cost} = \lim_{t\to+\infty} \text{cost}_r(t) = \lim_{t\to+\infty} \frac{\sum_{k=1}^N \sum_{b=1}^{\sigma^k} w_b^k}{\sum_{k=1}^N \sigma^k} \qquad (3)$$

The modified problem statement is as follows:

Problem 2.2 (Infinite Queue): For a given initial state $\mathbf{x}_0$, find a task assignment π, which may vary with time, such that

1) all queues make progress and
2) the cost as defined in Eq. (3) is minimized.



(a) $\pi(Q^1) = a^1, \pi(Q^2) = a^2$ (b) $\pi(Q^1) = a^1, \pi(Q^2) = a^2$ (c) $\pi(Q^1) = a^2, \pi(Q^2) = a^1$

Fig. 1. Relevant variables for an example problem case. (a) Agents a^1 and a^2 (solid black) live in workspace $\mathcal{W}$ and attend to queues Q^1 (blue) and Q^2 (red). The agents individually complete the first task and continue to second when (b) they encounter each other and form a group. The cost of completing a task is the total distance traveled by any agent assigned to that task; i.e., $w_2^1 = w^1(q_2^1) + w^2(q_2^1)$.

III. Methods

Given the communication constraints, the existence of decentralized, optimal solutions to Problems 2.1 and 2.2 is unlikely. Agents must rendezvous to coordinate and share information about task locations; but in order to know they must coordinate, agents must already have some information about the tasks they will exchange. Rather than optimizing the global task assignment, then, we focus on cost improvements that can be made during local interactions that occur naturally as agents move towards their currently active tasks.

A. Algorithm

Each group of agents coordinates independently of other groups to navigate to its active tasks, according to Algorithm 1. The path planning step (line 4) is performed by the group of agents as a whole, and may take advantage of parallel speedup [8], [9].

Figure 1 shows an example of Algorithm 1 on a 2 agents, 2 queues system. At the beginning (Fig. 1(a)), agents a^1 and a^2 are not in communication, so each forms its own group. Each agent independently plans to complete the tasks in its task queue. In Fig. 1(b), the agents come within communication range of each other and form a single group. After exchanging position and task location data (line 3), the agents determine that the least cost assignment is that in which they exchange queues (line 4). In Fig. 1(c), the agents move towards their new task assignments (lines 6-9). They break communication and again form individual groups. The agents retain information only about the new task queue that they have been assigned.

Algorithm 1: Algorithm Run by Agent a^i

```
1  COMMUNICATIONS: Determine $a^i$'s group:
       $\mathcal{G} \leftarrow \{a^j | a^j$ and $a^i$ can communicate$\}$;
2  while not all active tasks belonging to $\mathcal{G}$ completed do
3      COMMUNICATIONS: Share agent/task positions;
4      PATH PLANNING (Sec. III-B): Find the least-cost
       in-group task assignment and corresponding path in
       the group configuration space;
5      SWITCHING: Exchange queues according to line 4;
6      while $\mathcal{G}$ unchanged AND no tasks completed do
7          CONTROL: Calculate control inputs (Sec. III-C);
8          COMMUNICATIONS: Update $\mathcal{G}$, agent positions;
9      end
10 end
```

B. Switching and Path Planning

The decision to switch task queues and the least-cost path to achieve the new task assignment is determined using an A* search over the *group task configuration space* for the least-cost path between the group's starting positions and task locations [10]. A* is used, as opposed to the bipartite matching of [11], since the cost of an agent-goal assignment depends on the other assignments and whether an agent must coordinate with others to avoid collision. For each group $\mathcal{G}$, we construct a space consisting of all transformations of the agents in the group such that no agents in the group collide with each other. For groups that are a single agent, this space is equivalent to the workspace $\mathcal{W}$ itself. For larger groups, it is a subset of the Cartesian power of $\mathcal{W}$ (points corresponding to inter-agent collisions are removed). The tessellation of $\mathcal{W}$ induces a tessellation on the group task configuration space, allowing us to define a discrete representation of the group task configuration space as a graph, $G^P = (\mathcal{V}^P, \mathcal{E}^P)$, where vertices $\mathcal{V}^P = \{P^1, P^2, \ldots\}$ are polytopes in this Cartesian product space and edges in $\mathcal{E}^P$ connect pairs of polytopes that share a supporting hyperplane. Each edge is weighted with a heuristic distance designed so that the optimal path through this group polytope graph is truly the optimal path with respect to our defined cost function (refer [7] for more details). Applying A* to the resulting graph yields a discrete path on the polytopes between the group's current configuration and its goal. The final positions of the agents give the least cost task assignment for the given starting and task locations.

C. Control Policy

Once the path on the polytopes is found, local navigation functions, described in [10], on each pair of sequential polytopes drive the system toward the goal. A *navigation function* on a polytope P is a twice differentiable Morse function $\psi : P \to [0, 1]$ with a unique minimum of 0 at the goal point in P and that evaluates uniformly to 1 on the boundary. Using the control law $\mathbf{u} = -\nabla\psi(\mathbf{x})$, we can drive the system to successive intermediate goals inside polytopes along the computed path. Note that while theoretically the state's descent along the gradient to the goal is infinite in time, the time to enter within an ϵ-ball of it is finite. Practically, it is impossible to attain an exact position for a real system, and getting within a small distance of a goal is acceptable. Therefore, this control law will allow the system to reach a goal in finite time.

IV. Analysis

The switching policy functions entirely online and requires no centralized entity, making a globally optimal task assignment impossible. It is simple to construct a set of task queues for which this approach will perform arbitrarily badly when compared to an optimal centralized policy with full knowledge of the queue contents. Despite this, our switching policy does guarantee a decrease in global cost for every exchange of tasks that occurs, and that progress is made on the global level. Of greater interest, however, is that all task queues make progress.

A. Finite-Length Queues

In the case that all task queues are of finite length, the system is stable and all tasks will complete. This result is a straightforward extension of the proofs in [7].

Theorem 4.1: If all task queues are of finite length, every task in $\mathcal{Q}$ will be completed.

Proof: Partition time into intervals $\{e_1 = (t_0, t_1), e_2 = (t_1, t_2), \ldots\}$, which we call *epochs*, during which the set of active tasks remains constant. An epoch ends when a task is completed and the next task becomes active. The last epoch extends until $t \to +\infty$.

Now consider an epoch e_i. During this time interval, the active tasks in the system are constant, so the proof in [7] holds, and the system will converge asymptotically to the task locations. Because we require agents to get within only ϵ distance of a task, some task will be completed after only a finite amount of time. This task can be one of two types:

I) *The task is not final.* A new task will become active, and the epoch will end. At least one queue will make progress.

II) *The task is final.* Since no new task will become active, the epoch will continue and the system will continue to converge to the given task locations. If tasks remain that are not yet completed, the next task to be completed will again fall into one of these two types. If all tasks in the epoch are completed, then they all are the final tasks in their queues, or else the epoch will have ended before this time. In that case, all tasks in $\mathcal{Q}$ are complete.

Therefore, any epoch containing active tasks that are not final will end after some finite amount of time. The only epoch that can last forever, i.e., the last epoch, is one in which all tasks are final. Since the system will converge to these tasks, every task in $\mathcal{Q}$ will be completed. ■

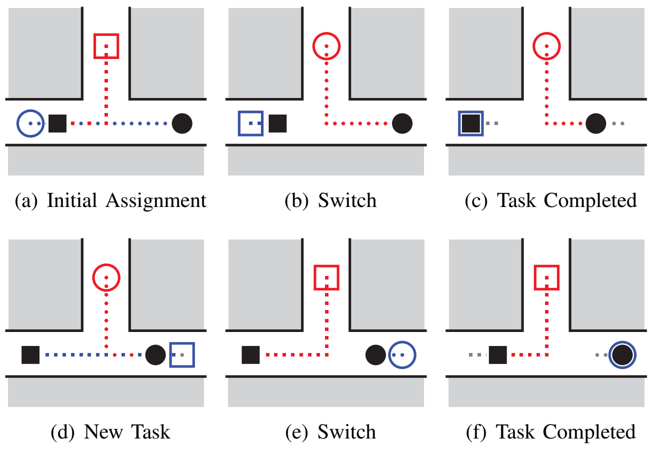


Fig. 2. Example of a 2 agent, 2 queues situation where the red queue will not make progress. Agents are shown as a solid black square and circle. They are always in communication with each other. The active tasks to which they are assigned are the empty square and circle respectively. The colors indicate the queue for each task. The blue queue alternates between a task in the bottom left and in the bottom right. Repeated changes of task assignment for the two agents cause the red queue to stall.

B. Infinite-Length Queues

When task queues are infinite-length, the switching policy in Section III-B cannot guarantee completion of all tasks in a finite amount of time. Figure 2 shows a simple example of a task queue that is unable to progress. The blue queue alternates task locations between the bottom left and bottom right locations, while the red queue (stalled) has an active task at the top. Every time a new task in the blue queue becomes active, the agents exchange tasks to achieve a better assignment, and all previous progress towards the red task is lost. Since the blue queue is infinite in length, neither agent will ever complete the red task. Therefore, for infinite-length queues, using the switching policy described in Section III-B only guarantees that *some* queue will make progress.

Theorem 4.2: At least one queue in $\mathcal{Q}$ will make forward progress.

Proof: The proof for this is identical to that of Theorem 4.1. In particular, we partition time into (an infinite number of) epochs. During each epoch, the set of active tasks is constant, so by [7], the system will converge asymptotically to the task locations. Some task will be completed in a finite amount of time, and the queue to which that task belongs will make forward progress. ■

While making progress along some queue is beneficial, we would like to guarantee that all queues will make progress. To do so, we borrow ideas from job scheduling literature, specifically in *aging* [12], [13] to force stalled queues to progress. Aging techniques are heuristics that monitor how long a task has been waiting (active) and force execution of the task when the waiting time becomes too long.

We modify the cost objective to

$$\text{cost} = \frac{\sum_{k=1}^{N} \sum_{b=1}^{n^k} C_t(w_b^k)}{\sum_{k=1}^{N} n^k} \tag{4}$$

in the case of finite queues and

$$\text{cost} = \lim_{t \to +\infty} \frac{\sum_{k=1}^{N} \sum_{b=1}^{\sigma^k} C_t(w_b^k)}{\sum_{k=1}^{N} \sigma^k} \tag{5}$$

for infinite queues, where C_t is an aging function.

Theorem 4.3: If the aging function C_t is an increasing function with strictly increasing first derivative that satisfies

$$\lim_{\hat{t} \to +\infty} \left. \frac{dC_t(t)}{dt} \right|_{t=\hat{t}} = +\infty, \tag{6}$$

then every queue in the system will make progress.

Proof: It suffices to show that no task will take infinitely long to complete. Consider any active task $q^k_{\sigma^k}$. If no queue permutations involving Q^k ever occur, this is true by virtue of the path planning and control policy, and of the discussion in [10]. Therefore, we need only consider what happens if a task permutation occurs. Let the agent that used to be assigned to Q^k be a^{old}, and the new agent be a^{new}. One of two cases will occur: (In the following analysis, the terms *closer* and *farther* are in terms of cost, rather than shortest distance)

I) *a^{new} is closer to $q^k_{\sigma^k}$ than a^{old}.* If permutations of only this type occur, then the cost of completing $q^k_{\sigma^k}$ can only decrease. The time until completion of $q^k_{\sigma^k}$ will be less than that if there were no permutation, and it will therefore be finite as well.

II) *a^{new} is farther from $q^k_{\sigma^k}$ than a^{old}.* In this case, the cost of completing $q^k_{\sigma^k}$ increases, and some previous progress towards $q^k_{\sigma^k}$ is lost. For a finite completion time, the number of permutations of this type must itself be finite.

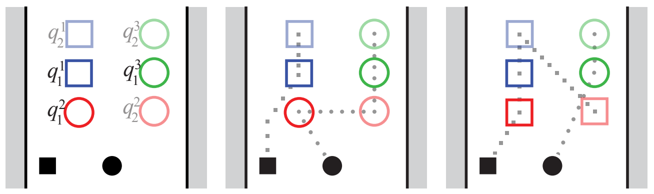
In order for a permutation to occur, the total cost to completion within a group must decrease, even though the cost to completion of $q^k_{\sigma^k}$ increases. However, due to assumption (6), eventually the increase in cost associated with further stalling completion of $q^k_{\sigma^k}$ will dominate any potential decrease in cost to other group tasks, and it will be impossible to decrease the group cost without decreasing the cost associated with the stalling queue. At this point, any future task permutations will be of type I, and task $q^k_{\sigma^k}$ will complete in finite time.

■

Intuitively, using an aging function that grows faster than $w^k_{\sigma^k}$ causes groups of agents to gradually shift from minimizing total cost to minimizing maximum cost when considering possible task permutations.

V. Multiple Queues per Agent

The assumptions in the previous section are restrictive in that there must be exactly as many task queues as agents. In a manufacturing or exploration setting, it is possible for the number of task queues to be greater than the number of agents and for agents to simultaneously perform several task queues. Even when the number of task queues is specifically designed to equal the number of agents, events such as agent failure may force individual agents to pick up more task queues than originally intended. In this section, we extend previous analysis to the case of multiple queues per agent.



(a) Initial Assignment (b) Without Switching (c) With Switching

Fig. 3. Example of a 2 agent, 3 queues situation where switching to the locally optimal task assignment increases the global cost. Agents are shown as a solid black square and circle. They are initially in communication with each other. The tasks to which they are assigned are the empty squares and circles respectively. Tasks are colored according to queue. Faded tasks are inactive. (b) and (c) show the paths that the agents would take given two different task assignments. The path resulting from the initial task assignment (b) covers less distance than the path after switching (c), even though the assignment in (c) yields a lower local cost.

A. Problem Definition

Similarly to Section II, N agents live in a bounded, connected workspace. Now, however, $\mathcal{Q}$ is a collection of $M > N$ queues. Our aim is still to minimize the mean distance traveled per task, and the problem definitions remain the same.

Since $M \neq N$, the task assignment π is no longer bijective, and multiple task queues may be assigned to the same agent. Since an agent can only make progress in one queue at a time, we call the queue that is currently making progress the *active queue* and the other queues *inactive*.

B. Switching Policy

Assigning queues to an agent now requires additional consideration of the order in which those queues will be completed. Thus, the decision of whether to exchange queues becomes more complex than can be solved by simply running A*. This problem of finding the locally optimal task assignment amounts to solving a Multiple Depot Hamiltonian Path Problem (MDHPP) with collision constraints every time a new group forms. Even if this is done, we cannot guarantee that the cost incurred by the switching system will be no greater than that incurred by the non-switching system using the initial assignment. Unlike the case of a single queue per agent, lack of knowledge about future tasks in this case can hurt. Figure 3 shows a situation where a switch that optimizes local cost (given the currently known task locations) increases the final global cost.

On the other hand, the guarantees on queue progress from Section IV still hold. In particular:

Corollary 5.1: If all task queues are finite-length, then every task in $\mathcal{Q}$ will be completed in finite time. If more than one task queue is infinite-length, then we guarantee that at least one queue will make progress. If aging techniques are used, then all queues will make progress.

The proofs are similar to those in Section IV. Note that for this case of Problems 2.1 and 2.2, in addition to dealing with the possibility of a queue stalling (i.e., the queue is active but does not make progress), we must also deal with the possibility of *starvation*, when a queue is always inactive. This can occur if whenever a new task becomes active, the agent plans to perform it first (before other waiting tasks). Similarly to the stalling problem, an aging function satisfying (6) will prevent the occurrence of starvation.

While MDHPPs with collision constraints solve the multiple queues problem, they are computationally expensive [14]. To our knowledge, few solutions exist in the literature, and even then only heuristics [15]. Therefore, in order to address the multiple queues problem, we decouple task assignment and path planning. We underestimate the distance between agents and tasks using the shortest path without considering collisions. Using these distances, we solve for a task assignment and an order of execution for the tasks assigned to an agent. Finally, we calculate the collision-free paths for each agent in the group to its next task using A*.

Since this heuristic is based on estimates of cost and does not reflect the true distances-to-goal as determined by A* in the joint space, we cannot make any guarantees on queue progress. Instead, we present experimental results for a variety of switching heuristics in the following section.

VI. Experimental Results

A. Simulations

We simulated the system in MATLAB, using the MultiParametric Toolbox [16] for polytope computations. We used two two-dimensional environments, shown in Fig. 4: (Plant) an area with large open spaces punctuated by obstacles of varying size and shape, as is characteristic of a manufacturing plant; and (Street) a grid of long narrow corridors, similar to the streets that a delivery vehicle would use. Although these environments were designed to be consistent with our motivating examples in Section I, we can use the results to draw general conclusions about our algorithm.

We placed six agents in each of the two environments and generated 18 infinite-length queues with tasks uniformly randomly distributed throughout the free space. The initial task assignment was random, with each agent assigned three queues. We used an aging function of $C_t(t) = t^2$.

We tested the following five switching policies. In all five, once the task assignment has been determined, agents individually solve the HPP for their assigned tasks, then perform A* within their groups to find collision-free paths to their next task.

1) *NoSwitch.* The task assignment does not change.
2) *MDHPP.* Agents in the group collectively solve the MDHPP on the graph of agents and task locations, using the length of the shortest path as the edge weights, ignoring collision constraints. The computational complexity of this policy is exponential in the number of agents and tasks in a group, but it will give the lowest cost task assignment out of all the policies tested.
3) *DivideAndConquer.* The agents in the group equitably partition the workspace among themselves, assuming a uniform distribution of tasks. Each agent assumes responsibility of one region of the workspace and is assigned all tasks that lie within it. In the case of infinite communication range, where a group of agents will consist of the entire team, this policy reduces to the multi-vehicle Divide & Conquer policy, which has been proven to perform within a factor N of optimal for the dynamic vehicle routing problem [1].
4) *Greedy.* Agents take turns choosing the next closest active task to their current location.
5) *A*First.* Given the initial task assignment, all queues assigned to an agent are treated as a single set. Each agent orders its set of queues individually and determines the first task to complete. Agents in the group then share their first tasks and use A* search to determine the least cost permutation of these sets of task queues. With this policy, we test whether regrouping of task queues actually improves performance. When $M = N$, this policy reduces to that described in Section III.

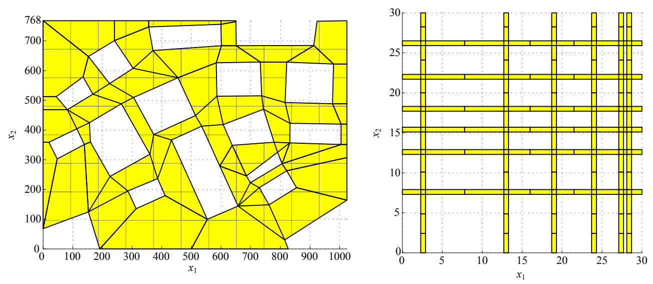
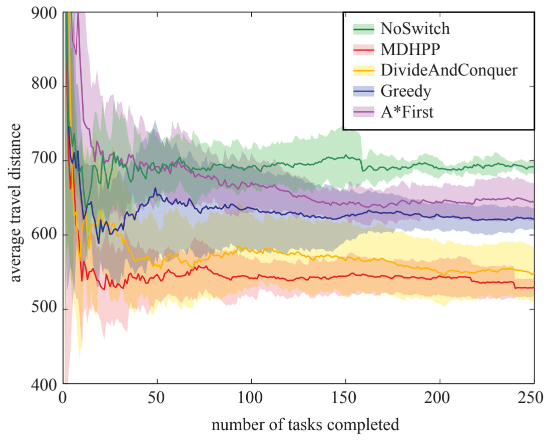


Fig. 4. Environments used in simulations, partitioned into polytopes.

Figures 5 and 6 show the results for 10 trial runs of each switching policy. At the beginning, the cost of completing a task is high since the initial allocation is random. As agents move around the environment, they encounter each other and exchange tasks, so that the per-task distance traveled quickly reaches some steady state. In both environments, solving an MDHPP yields the lowest cost of all the policies, and NoSwitch yields the highest cost, as expected. As for the other three policies, relative cost varies with the environment; the DivideAndConquer approach performs best out of the three in the Plant environment and the Greedy policy performs best in the Street environment.

Given the structure of the environments tested, this can be expected. The Plant environment has large open spaces relative to the holes, and agents can choose from a larger variety of similar-length paths to get from one point to another. Here, a Voronoi decomposition of the space makes sense, since the length of the shortest path between any two points in the cell will be approximately equal to the Euclidean distance between them. On the other hand, the Street environment consists of narrow corridors, and the lengths of the shortest path between two points will be relatively long compared to the straight line path. Moving always to the closest active task (Greedy) in this case will

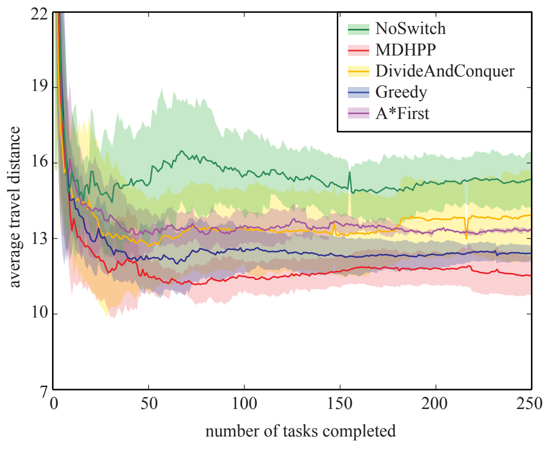


(a)

	# switches per task	computation time per switch
NoSwitch	0 (0)	–
MDHPP	1.43 (13.10)	8.75 (44.02)
DivideAndConquer	2.93 (28.29)	0.92 (3.57)
Greedy	1.13 (1.63)	1.48 (5.19)
A*First	1.49 (2.07)	5.91 (13.03)

(b)

Fig. 5. Results for 10 simulation runs/policy in the Plant environment. (a) Average distance/task for the system over time. Mean cost over the 10 runs are shown as a solid line, and the shaded areas indicate the standard deviation. (b) Switching frequency and computation time per switch. Entries take the form 'mean (std. dev.)'.



(a)

	# switches per task	computation time per switch
NoSwitch	0 (0)	–
MDHPP	2.00 (8.95)	5.65 (37.97)
DivideAndConquer	2.88 (11.96)	0.98 (5.17)
Greedy	1.67 (7.77)	1.61 (4.48)
A*First	0.81 (1.29)	2.28 (43.73)

(b)

Fig. 6. Results for 10 simulation runs/policy in the Street environment. (a) Average distance/task for the system over time. Mean cost over the 10 runs are shown as a solid line, and the shaded areas indicate the standard deviation. (b) Switching frequency and computation time per switch. Entries take the form 'mean (std. dev.)'.

perform better than repeatedly traversing the same corridors to reach tasks.

In addition to total cost, we investigated the switching frequency for each policy and the computation time required for a switch. Similarly to cost, the switching frequency for all policies is high at the beginning of the simulation but quickly stabilizes around some constant value. Tables 5(b) and 6(b) show the number of switches per task in the steady state, as well as the mean computation time required every time a new group forms. For all policies, the number of switches per task is approximately the same (2-3 switches/task). However, MDHPP and DivideAndConquer show higher standard deviations. This seems to indicate that these two policies are more sensitive to the actual task locations in the environment.

Group computation time is "wasted time": no progress towards tasks can be made since agents do not know which tasks they will shortly be assigned. Despite the fact that the agents do not travel during group computations and therefore no additional cost is incurred, practically we would like to minimize this downtime. Based on the results, we can see that MDHPP requires the greatest amount of time to determine if a switch should be made, as expected, although because of high variance, the average travel distance obtained from this policy is not actually significantly different from any of the other policies.

Finally, the results show evidence that progress on some queues was being made. As discussed in Section IV, we would also like to ensure that no queues are stalled. We can check this condition by consulting the total aging function C_t over all currently active tasks. If all tasks are completed in finite time (no queue stalls), then the aging function will remain bounded. As shown in Fig. 7, the switching policies that do not allow redistribution of queues are not able to keep the aging function from steadily increasing towards infinity. On the other hand, the policies that allow redistribution and reordering of task queues keep the aging function relatively small and thus prevent stalling. The MDHPP policy yields the smallest total aging for the system.

B. Experiment

We implemented the system for two KUKA youBots, using a VICON motion capture for tracking the vehicles and real-time MATLAB computation of the control inputs. The youBots collectively served four queues, each consisting of tasks alternating between two locations. Figure 8(a) shows an overhead shot of the experimental setup with the robots' trajectories during part of a test using the DivideAndConquer policy overlaid. The effects of the aging function can clearly be seen in the trajectory of robot 2. Without aging, robot 2 would serve only the green tasks in the upper part of the environment; with aging in effect, the turquoise task on the right becomes sufficiently critical that it is also performed.

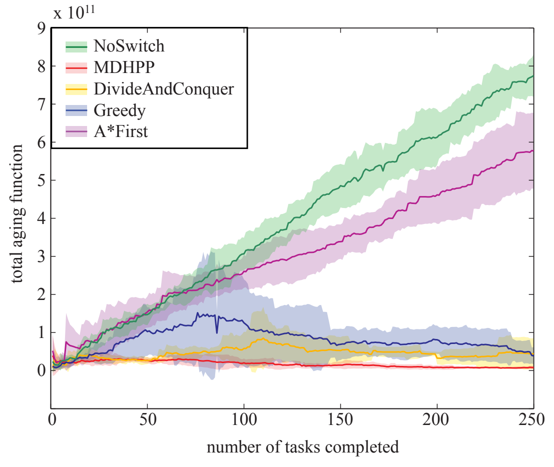
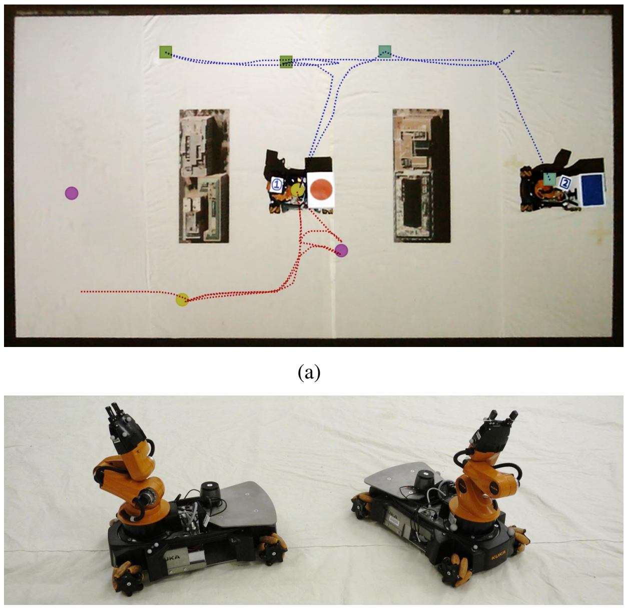


Fig. 7. Aging function summed over active tasks over time for the Plant environment. Means over 10 simulation runs are shown as solid lines, while the shading indicates the standard deviation.



(a)

(b)

Fig. 8. (a) Overhead view of experimental setup using KUKA youBots (b). Two youBots serve four queues consisting of tasks in alternating locations. Task locations are colored according to queue. Partial trajectories of the robots are overlaid in red (robot 1) and blue (robot 2). The effect of the aging function can be seen where robot 2 moves to perform the waiting turquoise task rather than serve the green queue exclusively.

VII. Discussion

In this paper, we present algorithms for decentralized task assignment for a team of agents that perform task queues in a perfectly known environment. By allowing agents in communication to share and exchange tasks, we are able to reduce the total cost, as a measure of distance traveled, to satisfy all tasks. We show not only that task switching can result in a stable system, but that queues are guaranteed to make progress if the local cost can be lowered by the switch.

For infinite-length queues, we introduce the idea of an aging function, borrowed from the literature in job scheduling, which enables guarantees of progress in *all* task queues. Finally, we present and discuss simulation and experimental results for multiple task switching policies in the case where there are more task queues than agents; this can occur when robots fail and other robots must complete the failed robots' queues. Our work has potential applications for any system where agents must collectively perform series of tasks in order, such as manufacturing or pickup-delivery systems.

Future work includes further optimizing performance by considering patterns in the task queue. For example, in manufacturing, we can expect task queues, which may correspond to assembly instructions, to follow some structure. By learning these patterns, agents can not only exchange task queues but also predict whether it will be beneficial to pass queues to other agents in the future. In this way, encounters between robots will not be restricted to those that happen by chance while agents are moving towards tasks, but can occur at preplanned locations arranged by the agents themselves. In this case, we expect that the prediction power of agents will allow them to further decrease task execution costs.

References

[1] F. Bullo, E. Frazzoli, M. Pavone, K. Savla, and S. L. Smith, "Dynamic vehicle routing for robotic systems," *Proceedings of the IEEE*, no. 99, pp. 1–23, 2011.

[2] S. L. Smith, M. Pavone, F. Bullo, and E. Frazzoli, "Dynamic vehicle routing with priority classes of stochastic demands," *SIAM Journal on Control and Optimization*, vol. 48, no. 5, pp. 3224–3245, 2010.

[3] D. J. Bertsimas and G. Van Ryzin, "Stochastic and dynamic vehicle routing in the euclidean plane with multiple capacitated vehicles," *Operations Research*, pp. 60–76, 1993.

[4] "Matternet," 2012. [Online]. Available: http://matternet.us/

[5] M. B. Dias, R. Zlot, N. Kalra, and A. Stentz, "Market-based multirobot coordination: A survey and analysis," *Proceedings of the IEEE*, vol. 94, no. 7, pp. 1257–1270, 2006.

[6] M. Golfarelli, D. Maio, and S. Rizzi, "A task-swap negotiation protocol based on the contract net paradigm," Research Center for Informatics and Telecommunication Systems, Univ. Bologna, Tech. Rep. 005–97, 1997.

[7] N. Ayanian, D. Rus, and V. Kumar, "Decentralized multirobot control in partially known environments with dynamic task reassignment," in *3rd IFAC Workshop on Distributed Estimation and Control in Networked Systems*, 2012.

[8] V. Kumar, K. Ramesh, and V. N. Rao, "Parallel best-first search of state-space graphs: A summary of results," in *1988 National Conference on Artificial Intelligence*, 1988, pp. 122–127.

[9] Z. Cvetanovic and C. Nofsinger, "Parallel Astar search on message-passing architectures," in *23rd Annual Hawaii International Conference on System Sciences*, vol. 1, 1990, pp. 82–90.

[10] N. Ayanian, V. Kallem, and V. Kumar, "Synthesis of feedback controllers for multiple aerial robots with geometric constraints," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011, pp. 3126–3131.

[11] H. W. Kuhn, "The Hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[12] K. Aida, "Effect of job size characteristics on job scheduling performance," in *Job Scheduling Strategies for Parallel Processing*, 2000, pp. 1–17.

[13] D. H. J. Epema, "Decay-usage scheduling in multiprocessors," *ACM Transactions on Computer Systems*, vol. 16, no. 4, pp. 367–415, 1998.

[14] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY: W. H. Freeman & Co., 1979.

[15] M. Skutella and W. Welz, "Route planning for robot systems," in *2010 Annual International Conference of the German Operations Research Society*, 2011, p. 307.

[16] M. Kvasnica, P. Grieder, and M. Baotić, "Multi-Parametric Toolbox (MPT)," 2004. [Online]. Available: http://control.ee.ethz.ch/˜mpt/