On On-line Task Assignment in Spatial Crowdsourcing

Mohammad Asghari
Department of Computer Science
University of Southern California
masghari@usc.edu

Cyrus Shahabi
Department of Computer Science
University of Southern California
shahabi@usc.edu

Abstract—A new platform, termed spatial crowdsourcing (SC), is emerging that enables a requester to commission workers to physically travel to some specified locations to perform a set of spatial tasks (i.e., tasks related to a geographical location and time). For spatial crowdsourcing to scale to millions of workers and tasks, it should be able to efficiently assign tasks to workers, which in turn consists of both matching tasks to workers and computing a schedule for each worker. The current approaches for task assignment in spatial crowdsourcing cannot scale as either task matching or task scheduling will become a bottleneck. Instead, we propose an on-line assignment approach utilizing an auction-based framework where workers bid on every arriving task and the server determines the highest bidder, resulting in splitting the assignment responsibility between workers (for scheduling) and the server (for matching) and thus eliminating all bottlenecks. Through several experiments on both real-world and synthetic datasets, we compare the accuracy and efficiency of our real-time algorithm with state of the art algorithms proposed for similar problems. We show how other algorithms cannot generate as good of an assignment because they fail to manage the dynamism and/or take advantage of the spatiotemporal characteristics of SC.

I. INTRODUCTION

Smartphones are ubiquitous: we are witnessing an astonishing growth in mobile phone subscriptions. The International Telecommunication Union estimates there are nearly 7 billion mobile subscriptions worldwide. Meanwhile, the mobile phones’ sensors (e.g., cameras) are advancing and the network bandwidth is constantly increasing. Consequently, every person with a mobile phone can now act as a multi-modal sensor, collecting and sharing various types of high-fidelity spatiotemporal data instantaneously (e.g., picture, video, audio, location, time, speed, direction, and acceleration).

Exploiting this large crowd of potential workers and their mobility, a new mechanism for efficient and scalable data collection has emerged: Spatial Crowdsourcing (SC) [1]. Spatial crowdsourcing requires workers (e.g., willing individuals) to perform a set of tasks by physically traveling to certain locations at particular times. Spatial crowdsourcing is exploited in numerous industries, e.g., Uber, TaskRabbit, Waze, Gigwalk, etc., and has applications in numerous domains such as citizen-journalism, tourism, intelligence, disaster response and urban planning. With spatial crowdsourcing, a requester submits a set of spatiotemporal tasks to a spatial crowdsourcing server (SC-Server). Subsequently, the SC-Server has to select a worker to perform each task.

Different studies on spatial crowdsourcing [1], [2], [3], [4], [5], [6], [7] can be classified based on two basic characteristics of the problem; (1) whether the problem matches a task with a worker and (2) whether the problem schedules matched tasks for the workers. Early studies in SC [1], [6], [5] used a scheduling-oblivious-matching (SOM) approach where tasks are matched with workers without considering the workers’ schedule. Assuming the tasks have already been matched with workers, other studies. [8], [2] study the problem of scheduling the tasks that have been assigned to a worker. They show that there is no guarantee that the worker could schedule all of its matched tasks. We argue that with Spatial Crowdsourcing, it is not enough to only match a task with a worker. An SC-Server must consider the schedule of every worker when matching a task to workers and only consider those workers who are able to fit the task in their schedule. In this paper, we define the task assignment problem in SC consisting of two phases, a matching phase and a scheduling phase, which need to happen in tandem. Neither of these phases should be ignored, otherwise, the resulting solution is rendered infeasible for real-world applications.

More recent studies consider both matching and scheduling in spatial crowdsourcing [3], [4], [7]. These studies utilize a batched assignment scheme, where the assignment is delayed for a period of time (i.e., batching time interval) during which all the arrived tasks are batched to be matched and scheduled during the next interval. Once the tasks are batched and processed together, suddenly the matching phase becomes complex because many tasks need to be matched to many workers. This in turn adds to the running time and increases the batching time interval.

A long batching time interval (e.g., 10 minutes) has two main disadvantages. First, the duration of the batching time interval should be subtracted from the tasks’ deadline, leaving each task with less available time to be scheduled. Second, the batch scheme can no longer generate real-time assignments. While there might be cases where a real-time assignment is not required, there are many other real-world applications where real-time assignments are a necessity; For example, an Uber user requesting a ride, does not want to wait for 10 minutes to find out if a driver is available or not. Contrary to
batched assignment, in on-line assignment, a task is assigned to a worker as soon as it arrives at the SC-Server. This requires the server to perform matching and scheduling in real-time. With on-line assignment, at each point of time the SC-Server is processing only one task and hence, the matching phase becomes a one-to-many matching where there are multiple workers and only one task. Consequently, the complex many-to-many matching phase of batched assignment is reduced to only selecting the best worker that can fit the task in its schedule. Even though matching is fast with on-line assignment, the server must still perform scheduling for multiple workers. Therefore, the scheduling phase becomes the bottleneck in on-line assignment. As shown in [2], scheduling for a single worker can be performed fast. However, an on-line monolithic[9] SC-Server (monolithic-SC), where the server has to schedule only a single task for all workers, is not capable of processing tasks in real-time. For example, in New York City, during rush hours, there are as many as 10+ ride requests per second [10]. Through experiments, we show that a monolithic-SC server is not able to support such throughput in real-time.

In this paper, we introduce Auction-SC as an extension to the auction-based framework of [11] to be applied in a generic SC environment. In [11], we proposed an auction-based ride-sharing platform, named APART, with a passenger-to-driver assignment algorithm. The main objective of the assignment algorithm in APART is to maximize the platform provider’s profits, without compromising the passengers’ and drivers’ monetary incentives (i.e., either charging the passengers more or paying the drivers less). With Auction-SC, we introduce a new bidding rule where the goal is to maximize the number of completed tasks. Similar to monolithic-SC, Auction-SC is an on-line framework where tasks get matched with workers as soon as they become available and hence, the matching phase is fast. Moreover, we overcome the scheduling bottleneck by distributing it among the workers. Furthermore, with ride-sharing platforms, the capacity of the car bounds the number of requests that have to be scheduled at any point in time. Consequently, even the exhaustive search algorithms of [11] performed scheduling for each driver in real-time. However, in a general SC environment, workers can accept as many tasks as they wish as long as they are able to finish them before the tasks’ deadlines. Consequently, we propose a dynamic programming based approach for scheduling tasks that overcomes the complexity of the branch-and-bound scheduling algorithm of [11].

We conduct many experiments on both real-world and synthetically generated workloads to evaluate different aspects of Auction-SC compared to state-of-the-art approaches proposed for task assignment in SC. For our real world data we use two geo-social datasets (Gowalla & Foursquare). First, we show that compared to Auction-SC, both the SOM and batched approach result in a much lower assignment rate as a result of ignoring scheduling while matching and long batching intervals, respectively. Subsequently, we show that when matching and scheduling are performed in tandem, neither the batched scheme nor the on-line monolithic SC-Server can process more than 5 tasks per second. However, with the auction based framework, the throughput of the system increases by orders of magnitude.

In sum, the contributions of this paper are as follows:

- We formally define the on-line task assignment problem in SC and analyze its complexity.
- We extend APART[11] and introduce Auction-SC to be applied in generic SC environments and utilize it with a new bidding rule to maximize the number of completed tasks.
- We use the unique characteristics of the task assignment problem in SC and propose a novel algorithm for scheduling tasks for each worker.
- Through experiments on real-world and synthetic data, we show Auction-SC outperforms alternative approaches in the number of completed tasks and scalability.

The remainder of this paper is organized as follows. In Section II we formally define the on-line task assignment problem in SC and discuss its complexity. We review the related work in Section III. We briefly overview the auction-based framework of [11] in Section IV and discuss how it can be extended to be applied in a generic SC environment. In Section V we propose our dynamic programming based algorithm for scheduling. We show the results of our experiments on both real-world and synthetic data in Section VI and conclude the paper with guidelines for future work (Section VII).

II. Preliminaries

In this section, we define the terminologies used in the paper and provide a formal definition of the problem under consideration and analyze the complexity of the problem.

We start by defining some terminologies in order to formally define the task assignment problem in spatial crowdsourcing.

Definition 1 (Spatial Task). A spatial task $t$ shown as $\langle l, [r, d]\rangle$ is a task to be performed at location $l$ with geographical coordinates, i.e., latitude and longitude. The task becomes available at $r$ (release time) and expires at $d$ (deadline).

It should be pointed out that in a spatial crowdsourcing environment, a spatial task $t$ can be executed only if a worker is at location $t.l$. For example, if the query is to report the traffic situation at a specific location, someone has to actually be present at the location to be able to report the traffic. Hereafter, whenever we use task we are referring to a spatial task. Now, we formally define a worker.

Definition 2 (Worker). A worker $w$ shown as $\langle l, T, [s, e]\rangle$ is any entity, e.g., a person, willing to perform spatial tasks. We show the current location of the worker by $w.l$. Each worker has a list of tasks assigned to it, $w.T$. Also $w.s$ and $w.e$ show the availability of the worker such that the worker is available during the time interval $\{w.s, w.e\}$.

Throughout the paper, we assume every worker moves one unit of length per unit of time. Therefore, we assume that
\( \phi(a,b) \) shows both the distance and the travel time for moving from point \( a \) to point \( b \).

**Definition 3 (Schedule).** A schedule \( \pi \) is an ordered list of tasks shown as \( \langle t_1, ..., t_n \rangle \) where \( n \) is the number of tasks in \( \pi \). We show the \( i^{th} \) task in \( \pi \) with \( \pi^i \).

**Definition 4 (Valid Schedule).** Schedule \( \pi \) is a valid schedule for worker \( w \), if and only if:

\[
\forall i, 1 \leq i \leq n \sum_{j=1}^{i} \phi(\pi^{j-1}, \pi^j) \leq \pi^i.d - t_0
\]

where \( \pi^0 \) and \( t_0 \) represent the current location of \( w \) and current time, respectively.

At each point in time, a worker \( w \) is associated with a valid schedule \( (\pi_w) \) and completes the tasks based on their order in \( \pi_w \).

**Definition 5 (Matching).** Assuming we have a set of workers \( W \) and a set of tasks \( T \), we call \( M \subseteq W \times T \) a matching if for each \( t \in T \) there is at most one \( w \in W \) such that \( (w,t) \in M \). We call \( (w,t) \in M \) a match and say \( t \) has been matched to \( w \).

For each matching \( M \), we define the value (benefit) of \( M \) as:

\[
Value(M) = |M|
\]

**Definition 6 (Valid Matching).** A matching \( M \) is valid if and only if, for every worker \( w \), there exists a valid schedule \( \pi_w \), such that \( (w,t) \in M \implies t_i \in \pi_w \).

Now we can formally define the Task Assignment in Spatial Crowdsourcing (TASC) as follows:

**Definition 7 (Task Assignment in SC).** Given a set of workers \( W \), a set of spatial tasks \( T \) and a cost function \( \phi : (W \cup T) \times T \rightarrow \mathbb{R} \) where \( d(a,b) \) is the distance between \( a \) and \( b \), the goal of the TASC \((W,T,d)\) problem is to find a valid matching \( M \) with maximum value.

It is important to note that with task assignment in SC, the goal is to find a valid matching. This means that in addition to finding a matching between tasks and workers, the SC-Server has to also find a schedule for each worker to perform the tasks. Throughout this paper we use the terms matching phase and scheduling phase to refer to the two different aspects of task assignment in SC.

In a real life scenario, the SC-Server only finds out about the exact properties of tasks once they are submitted. Similarly, the server does not know when future workers will become available. Consequently, a real-world SC-Server can either process every single task as soon as it becomes available (online) or periodically wait for a specific duration and process all the tasks that have been submitted during that time (Batched). In this paper, we study the OnlineTASC problem, where the server processes an incoming task as soon as it is submitted by the requester.

**A. Complexity Analysis**

Previous studies have shown the TASC problem is NP-Hard [3]. However, the focus of this study is the OnlineTASC problem and thus, in this section we briefly discuss the complexity of OnlineTASC.

In order to analyze on-line algorithms, where each request is processed without knowing the future, we use a method named competitive analysis [12]. With this method, the performance of an on-line algorithm is compared to the performance of an optimal off-line algorithm that has knowledge of future events (clairvoyant). For the TASC problem, we measure the performance of each algorithm based on the number of assigned task. Assuming we show the performance of algorithm \( A \) on input \( I \) as \( |A(I)| \) we can define competitive ratio of an algorithm as:

**Definition 8 (Competitive Ratio).** For an on-line algorithm \( A \), we say \( A \) is \( c \)-competitive for some \( c > 0 \), if and only if:

\[
c = \min_{I \in \mathcal{I}} \left\{ \frac{|A(I)|}{|A^*(I)|} \right\}
\]

where \( A^* \) is the optimal off-line algorithm and \( \mathcal{I} \) is the set of all possible inputs.

Now we can prove the following theorem regarding the complexity of OnlineTASC.

**Theorem 1.** There does not exist a deterministic on-line algorithm for the OnlineTASC problem that is \( c \)-competitive (\( c > 0 \)).

**Proof.** Suppose there exists an algorithm \( A \) that is \( c \)-competitive for some \( c \geq 0 \). To prove no such algorithm exists, all we need to do is to prove there is at least one possible input \( I \), for which \( \frac{|A(I)|}{|A^*(I)|} \) is unboundedly small. For analyzing the competitive ratio of a deterministic on-line algorithm \( A \), it is assumed that there exist an adversary which knows every decision \( A \) makes and creates an input knowing what decision \( A \) is going to make. Here we show, how an adversary can generate an input for which the competitive ratio of any algorithm is unboundedly small. For simplicity, we only consider points on the x-axis and assume there is only one worker at point \( x = 0 \) in the beginning.

The input starts with \( t_1 \) such that \( t_1 = (5, [0, 5]) \) (Figure 1(a)); a task at point 5 with release time 0 and deadline 5. The algorithm can make two choices for the worker: (1) move towards \( t_1 \) or (2) stay still (in theory it can also make the worker to move away from \( t_1 \) which in the context of this proof would be similar to case (2)). If choice 1 is selected, the adversary can generate an input such that at time \( t = 2 \), tasks \( t_2, ..., t_n \) are all submitted with the exact same properties as \( (4, [2, 7]) \) (Figure 1(b)). Considering that at the release time of \( t_2, ..., t_n \), the worker is at point \( x = 2 \), it does not have enough time to get to \( t_2, ..., t_n \) before their deadline. However, an optimal off-line algorithm would have known about \( t_2, ..., t_n \) in advance and would have ignored \( t_1 \) in order to be able to complete \( n - 1 \) tasks instead. In other words, \( |A| = 1 \) where \( |A^*| = n - 1 \) and the
ratio could be unboundedly small by increasing \( n \). Therefore, we contradicted the assumption that \( A \) is \( c \)-competitive. A similar argument can be made if choice 2 was selected by the algorithm by releasing tasks \( t_2, \ldots, t_n \) with properties as \( (7, [2,7]) \) (Figure 1(c)).

![Fig. 1: Adversary Generated Input](image)

III. RELATED WORK

Many studies in spatial crowdsourcing research focus on the task assignment problem\([1, 8, 2, 4, 3, 5, 6, 7]\). Kazemi and Shahabi\([1]\), Cheng et. al.\([5]\) and Fonteles et. al.\([6]\) formulate task assignment in spatial crowdsourcing as a matching problem. In \([1, 6]\) the primary objective is to maximize the number of matched tasks while in \([5]\) the minimize the distance between the tasks and their matched workers. Furthermore, neither of these studies consider the schedule of a worker when matching tasks and workers. Deng et. al.\([8]\) and Li et. al.\([2]\) study the problem of scheduling tasks that have already been assigned to a worker. While in \([8]\) all matched tasks are available at the time scheduling is performed, the authors look at the on-line version of the same problem in \([2]\) where the scheduling algorithm is performed immediately after a new task gets matched with the worker. More recent studies have considered both matching and scheduling using the batched scheme \([4, 3, 7]\). Chen et. al.\([4]\) formulate the problem as in Integer Linear Program where the objective is to minimize the total traveled distance of the workers. In \([3]\), to process each batch, the algorithm first performs matching and then tries to schedule the matched tasks. For those tasks that did not get scheduled another round of matching and scheduling is performed and this continues until either all tasks are scheduled or there is no eligible worker remained for an unscheduled task. In \([7]\), Guo et. al. propose greedy-enhanced genetic algorithms for matching and scheduling time-sensitive and time-insensitive tasks separately.

Our work is also related to some combinatorial optimization problems such as Vehicle Routing Problem (VRP) \([13]\). The general setting of VRP is to serve a number of customers with a fleet of vehicles and the objective is to minimize the total travel cost of those vehicles. Compared with VRP, with spatial crowdsourcing our objective is to maximize the number of completed tasks, whereas VRP aims to minimize the total travel time. In addition, the spatial workers in our problem setting are not located at one or several fixed depots, and each worker can show up at any unique location. In our setting the spatial tasks are also not guaranteed to be completed by the workers.

IV. AUCTION-SC FRAMEWORK

With a real-world SC system, the SC-Server will find out about a task and its properties only when it is released. The complexity of the many-to-many matching in addition to the need for immediacy (e.g., Uber) render the batch scheme impractical for real-world scenarios. Furthermore, in on-line monolithic-SC, scheduling multiple workers becomes the bottleneck and hence, real-time assignment is not guaranteed. For example, in New York City, during rush hours, there are as many as 10+ ride requests per second \([10]\). Through experiments, we show neither the batched scheme nor the on-line monolithic-SC are able to process such throughput in real-time.

In this section we first briefly overview how tasks are dispatched in Auction-SC. More details regarding the task dispatching process can be found in \([11]\). Following, we introduce the bidding rule in Auction-SC which maximizes the number of completed tasks and explain how each worker computes and submits its bid in Auction-SC.

A. Task Dispatching in Auction-SC

Auction-SC considers workers as bidders and tasks as goods. The server plays the role of a central auctioneer. Figure 2 explains how tasks are dispatched to nearby workers at a very high level: (1) Everything starts with a requester submitting a new task to the server. (2) Once the server receives a new task, it notifies the available workers in the vicinity of the pick-up location about the new task. (3) Each worker independently computes his bid by finding the optimal schedule that can fit the new task into his current schedule. The bidding process is performed as a sealed-bid auction where workers simultaneously submit bids and no other worker knows how much the other workers have bid. (4) Once all the bids are received, the server assigns the task to the worker with the optimal bid.

Algorithm 1 outlines the process of assigning an incoming task \( t \), where \( W_t \) is the set of eligible workers for task \( t \) (i.e., the workers that can reach the location of \( t \) before its deadline\([11]\)) (line 4). For each candidate worker \( w \), the ComputeBid method (line 5) is executed to perform scheduling and compute \( w 's bid. A worker’s bid is relative to the extra cost for the worker to perform the task (Section IV-B). Consequently, the platform chooses the worker with the minimum bid. In case of a tie in line 8, the algorithm randomly selects one worker among the ones with the lowest bid. Notice that in practice all the iterations of the for loop in Algorithm 1 (lines 4-7) run in parallel.

We end this section with a brief discussion on the communication cost in Auction-SC. As a result of task dispatching
and bid submission in Auction-SC, it seems that the communication cost in this framework will limit its advantages over existing approaches: i.e., batched assignment and monolithic-SC. However, regardless of the approach, there is always going to be some communication cost. In both the batched and online monolithic-SC approaches, the SC-Server is responsible for performing scheduling. To perform scheduling, the server requires the exact location of the workers, which means the workers have to constantly communicate with the server to update their locations. At the very least, each time a new task arrives, the server needs to communicate with the eligible workers to get their exact locations. On the other hand, the server in Auction-SC does not require the exact location of the workers as it is not responsible for performing scheduling for the workers. Instead, the server in Auction-SC does not require the exact location of the workers to get their exact locations. On the other hand, the arrival of a new task, the server needs to communicate with the eligible workers to find the most optimal schedule consisting of the tasks in \( \pi_w \) in addition to the new incoming task \( t \). The \textit{FindBestSchedule()} method in line 2 does this using a dynamic programming approach explained in Section V.

**Algorithm 2 ComputeBid\((w, t)\)**

| Input: \( w \) is the worker computing the bid and \( t \) is the incoming task. |
| Output: \( bid \) as the value of worker \( w \)'s bid for task \( t \). If \( w \) cannot complete task \( t \), \( bid \) is set to \(-\infty\) |
| 1: \( finish = \text{GetFinishTime}(\pi_w) \) |
| 2: \( \pi^* = \text{FindBestSchedule}(w, t) \) |
| 3: \( finish^* = \text{GetFinishTime}(\pi^*) \) |
| 4: \( bid = finish^* - finish \) |
| 5: \text{return} \( bid \) |

**B. Worker’s Bid Computation**

With Auction-SC, every worker computes his bid using a predefined bidding rule. A worker’s bid represents how good it is for him to be matched with the task. When computing a bid for a task, the workers have no knowledge about other tasks that might arrive in the future. Consequently, they have to make a greedy decision based on their current status. Existing studies have used various heuristics in order to match tasks with workers; e.g., spatial region [5], nearest neighbor [6], [1], [7] and earliest expiring task [6]. All these heuristics are applied to an SOM approach where the schedule of the worker is ignored. In this section we introduce a new heuristic where a task is assigned to a worker who can insert it into its schedule better than other worker and hence, it is called \textit{Best Insertion (BI)}.

Intuitively, if a worker spends less time to complete a task, it will likely have more time for performing other tasks. In BI, the server gives priority to workers that can better insert the incoming task into their schedules. Auction-SC considers the \textit{extra time} each worker will need to complete the new task in addition to its current schedule. Algorithm 2 shows how each worker computes its bid. The \textit{GetFinishTime()} method on lines 1 and 3 return the time the input schedule gets completed. The key step in computing the bid is for the workers to find the most optimal schedule consisting of the tasks in \( \pi_w \) in addition to the new incoming task \( t \). The \textit{FindBestSchedule()} method in line 2 does this using a dynamic programming approach explained in Section V.

**Algorithm 1 Dispatch\((W, t, \tau)\)**

| Input: \( W \) is the set of currently available workers, \( t \) is a new task and \( \tau \) is the current time |
| Output: \( w_{opt} \in W \) as the worker that task \( t \) gets assigned to |
| 1: \( w_{opt} \leftarrow \text{null} \) |
| 2: \( Bids_t \leftarrow \text{null} \) |
| 3: \( W_t \leftarrow \text{EligibleWorkers}(t) \) |
| 4: \textbf{for} \( w \in W_t \) \textbf{do} |
| 5: \( \text{bid}_w^t \leftarrow \text{ComputeBid}(w, \pi_w, t, \tau) \) |
| 6: \( Bids_t \leftarrow Bids_t \cup \{\text{bid}_w^t\} \) |
| 7: \textbf{end for} |
| 8: \( w_{opt} \leftarrow \arg \min_w \{\text{bid}_w^t \mid \text{bid}_w^t \in Bids\} \) |
| 9: \text{return} \( w_{opt} \) |

**V. Task Scheduling in SC**

For worker \( w \), to compute his bid on task \( t \), he has to find the most optimal valid schedule that can add \( t \) to \( \pi_w \). This requires checking all permutations of the set of tasks \( \pi_w \cup \{t\} \). The most optimal schedule is a permutation of the tasks such that (1) is a valid schedule and (2) has the earliest completion time among all permutations that yield a valid schedule. Scheduling a large set of tasks through an exhaustive search can take a long time. In this section, we introduce a dynamic programming approach for scheduling (DPS) that in practice, does not require checking all permutations of the set of tasks every time.

The key idea of DPS comes from Lemma 1 and Corollary 1:

**Lemma 1.** If any task is removed from a valid schedule, the remaining schedule is still valid.
Corollary 1. When adding a new task \( t \) to schedule \( \pi_w \), worker \( w \) only has to only consider those permutations of tasks in \( \pi_w \) that result in a valid schedule.

To better explain Lemma 1 and Corollary 1 consider the following example. A set of 3 tasks \( T = \{t_1, t_2, t_3\} \) have 3! = 6 different permutations. Let us assume that only 3 out of the 6 permutations yield a valid schedule for \( T \) and are shown as \( \pi_1, \pi_2 \) and \( \pi_3 \). To schedule a new task \( t_4 \), we only need to check if \( t_4 \) can be inserted in \( \{\pi_i\}_{i=1}^{3} \) without reordering the tasks already in \( \pi_i \).

To achieve this, in addition to \( \pi_w \), every worker also keeps track of other valid schedules that are not necessarily optimal. For each worker, we utilize a Valid Schedule Tree (VST) data structure to keep track of all valid schedules. A path from the root to any leaf in a VST corresponds to a valid schedule. Figure 3 shows an example of a VST and how a new task can be added to it.

**Figure 3: Valid Schedule Tree (VST) Example**

![VST Example](image)

Figure 3(a) shows a VST with 2 tasks. The root of the tree is always the current location of the worker. In Figure 3(a) we assume both \( \pi_{a_1} = \{t_1, t_2\} \) and \( \pi_{a_2} = \{t_2, t_1\} \) are valid schedules. Once \( t_3 \) arrives, it can be added to 3 different positions in both \( \pi_{a_1} \) and \( \pi_{a_2} \) and hence, 6 permutations have to be checked (Figure 3(b)). Assuming the dark nodes in Figure 3(b) result in invalid schedules, they are not added to the VST. Figure 3(c) shows the updated VST after adding \( t_3 \). In general, a new task can either be added on an existing edge or after a leaf of a VST. Consequently, if a new task \( t_4 \) arrives, based on the VST in Figure 3(c), only 11 options have to be checked (out of potentially 4! = 24 permutations with 4 tasks).

**Algorithm 3 FindBestSchedule(\( w, t \))**

**Input:** \( w \) is the worker and \( t \) is the incoming task.

**Output:** \( \pi^* \) as the optimal schedule for worker \( w \) after inserting \( t \). If \( w \) cannot add \( t \), the return value is null.

1: \( vst' = \text{InsertTask}(vst, t) \)
2: if \( vst' == \text{null} \) then
3: return null
4: end if
5: \( \pi^* = \text{ShortestSchedule}(vst') \)
6: return \( \pi^* \)

Algorithm 3 shows the process of finding the best schedule for inserting a new task \( t \). The \( \text{InsertTask}() \) method in line 1, generates a new VST for \( w \) (\( vst' \)) by inserting a new task into \( w \)’s current VST (\( vst \)). Once \( vst' \) is generated, the \( \text{ShortestSchedule}() \) method in line 5 runs DFS on \( vst' \) starting at the root and returns the schedule with the earliest finish time (\( \pi^* \)).

To further improve the performance of the scheduling process, for every node \( n \) in a VST we assign a cutoff time (\( n_c \)), which gives the latest time worker \( w \) can arrive at node \( n \). If worker \( w \) arrives at \( n \) after \( n_c \), then every schedule \( \pi \) where \( n \in \pi \), will become invalid. For example in Figure 4(a), if the worker arrives at node \( n_1 \) at time \( n_1c + \epsilon (\epsilon > 0) \), then both schedules \( \pi_1 = \{t_1, t_2, t_3\} \) and \( \pi_2 = \{t_1, t_3, t_2\} \) would become invalid. Earlier we explained that new tasks can either be added on an existing edge of a VST or after a leaf node. Figure 4(b) shows the process of adding \( t_4 \) on edge \( e \) in Figure 4(a). Task \( t_4 \) can be added on edge \( e \) only if:

\[
t_0 + \phi(w, t_4) + \phi(t_4, t_1) \leq n_1c
\]

where \( t_0 \) is the current time and \( \phi(a, b) \) gives the time it takes to go from point \( a \) to point \( b \).

**Figure 4: Example of Cutoff Times in a VST**

![Cutoff Times](image)

For a leaf node \( n \), the cutoff time (\( n_c \)), is equal to the deadline of the task corresponding to \( n \). For every other node, the cutoff time is equal to the smaller value between the deadline of the task corresponding to that node and the maximum cutoff time of its children:

\[
n_c = \begin{cases} 
    d(n) & \text{if } n \text{ is leaf} \\
    \min\{d(n), \max\{n'c \mid n' \in \text{next}(n)\}\} & \text{otherwise}
\end{cases}
\]

where \( d(n) \) is the deadline of the task corresponding to node \( n \).

Algorithm 4 outlines the process of recursively inserting a new node \( n \) in a sub-tree rooted at \( r \). The algorithm first attempts to insert \( n \) directly after \( r \) (lines 5-11). The \( \text{TrimInvalidSchedules}() \) method on line 7, removes all schedules rooted at node \( n \) that become invalid by inserting the new task before \( n \) (Algorithm 5). In the next step (lines 15-20), the algorithm recursively inserts node \( n \) after \( r \)’s children.

We end this section with brief discussion on the complexity of the scheduling algorithm. Even though in theory, the number of valid schedules can grow exponentially as we add more tasks, in our experiments on both real world and synthetic data, we realized that in practice this does not
Algorithm 4 Insert($r, n, \tau$)

Input: $r$ as the root of the sub-tree, $n$ as the node to be inserted and $\tau$ as the time the worker arrives at $r$

Output: Updated $r$ (if $n$ could not be inserted, returns $null$)

1: if $\tau + \phi(r, n) > n_c$ then
2: return $null$
3: end if
4: $r' = r$
5: $n' = n$
6: for $c$ in $r$.Children() do
7: $c' = \text{TrimInvalidSchedules}(c, \tau + \phi(r, n) + \phi(n, c))$
8: if $c' \neq null$ then
9: $n'.\text{AddChild}(c')$
10: end if
11: end for
12: if $n'.\text{HasChildren()}$ then
13: $r'.\text{AddChild}(n')$
14: end if
15: for $c$ in $r$.Children() do
16: $c' = \text{Insert}(c, n, \tau + \phi(r, c))$
17: if $c' \neq null$ then
18: $r'.\text{AddChild}(c')$
19: end if
20: end for
21: if $n'.\text{Children()}$ is not empty then
22: $r.\text{AddChild}(n')$
23: end if
24: if $r'.\text{HasChildren()}$ then
25: return $r'$
26: else
27: return $null$
28: end if

Algorithm 5 TrimInvalidSchedules($n, \tau$)

Input: $r$ as the root of the sub-tree and $\tau$ as the time the worker arrives at $r$

Output: Updated $r$ (if no schedule is valid, returns $null$)

1: if $\tau > n_c$ then
2: return $null$
3: end if
4: if $r.\text{IsLeaf()}$ then
5: return $r$
6: end if
7: $r' = r$
8: for $c$ in $r$.Children() do
9: $c' = \text{TrimInvalidSchedules}(c, \tau + \phi(r, c))$
10: if $c' \neq null$ then
11: $r'.\text{AddChild}(c')$
12: end if
13: end for
14: if $r'.\text{HasChildren()}$ then
15: return $r'$
16: else
17: return $null$
18: end if

VI. EXPERIMENTS

A. Dataset

We evaluate our algorithms using real check-in data in Foursquare and Gowalla and convert them to spatial tasks and workers in our system. We consider check-ins as a spatial task performed at the location the check-in happened. For each location, we consider all check-ins within a two hours duration. For each task, we set the release time and deadline to the first and last check-in time within the two hours duration. We consider each user as a spatial worker with start and end times equal to the user’s first and last check-in during a day. We select the initial location of a worker as a random point within the bounding box of all checked-in locations of the corresponding user. We also measure the travel time with the Euclidean distance between two points divided by an average speed of $60\text{km/h}$. We use the data from 5 metropolitan areas: New York, Los Angeles, Paris, London & Beijing. Table I shows the total number of tasks (and workers) for each city.

<table>
<thead>
<tr>
<th>City</th>
<th>Gowalla # Tasks</th>
<th>Gowalla # Workers</th>
<th>Foursquare # Tasks</th>
<th>Foursquare # Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Los Angeles</td>
<td>197,353</td>
<td>4,126</td>
<td>185,061</td>
<td>9,136</td>
</tr>
<tr>
<td>New York</td>
<td>118,406</td>
<td>3,987</td>
<td>577,124</td>
<td>17,367</td>
</tr>
<tr>
<td>London</td>
<td>60,180</td>
<td>2,293</td>
<td>186,755</td>
<td>9,711</td>
</tr>
<tr>
<td>Paris</td>
<td>18,932</td>
<td>1,829</td>
<td>105,098</td>
<td>6,095</td>
</tr>
<tr>
<td>Beijing</td>
<td>3,638</td>
<td>699</td>
<td>21,013</td>
<td>1,075</td>
</tr>
</tbody>
</table>

TABLE I: Number of tasks/worker for each city in real dataset

We also generate a synthetic datasets with realistic streaming workload using [15]. To generate a workload suitable for SC systems, we modeled three different sets of parameters:

Temporal Parameters: We assume workers and tasks arrive following Poisson processes. In our experiments, the default Poisson arrival rates for tasks and workers are $\mu_t = 20/min$ and $\mu_w = 3/min$, respectively. Subsequently, the duration of the tasks and workers were randomly sampled from closed range of $[1, 4]$ hours and $[1, 8]$ hours, respectively.

Spatial Parameters: Figure 5 shows the spatial distribution of tasks from our real-world dataset in Los Angeles. As depicted, the tasks are not uniformly distributed in space. The spatial distribution is rather skewed, meaning that the density of the tasks at certain areas is higher. To model the same behavior with our synthetic workloads, we created 6 two dimensional Gaussian clusters with randomly selected means and standard deviations. Eighty percent of the tasks are sampled within the clusters and the rest are uniformly distributed.

Static Parameters: In addition to the spatiotemporal parameters, we consider two other parameters. The default workload...
size of each experiment is 50K tasks. The task arrival rate and the number of tasks determine the duration of the simulation. Based on the duration of the simulation and the workers’ arrival rate, the total number of workers may vary. The maximum number of tasks a worker can perform, i.e., $w_{max}$, is a uniformly random number from the closed interval $[8, 12]$.

![Fig. 5: Spatial Distribution of Tasks in Gowalla](image)

**B. Experimental Methodology**

1) **Algorithms:** We compared the results of our framework (AUC) with three other approaches: NN (i.e., nearest neighbor) as a scheduling-oblivious-matching (SOM) approach, BCHD (i.e., batched) and MONO (i.e., on-line monolithic-SC).

We explained that the SOM approach is when the tasks are assigned to workers without considering the workers’ schedules. In other words, the server assigns tasks to workers based on some heuristic and once the task is assigned to a worker, the worker attempts to add the task to its schedule. If it succeeds then the task gets completed and otherwise the task is dropped and will not get completed. We tried various heuristics and among those the nearest neighbor generated the best results and hence, we only include the NN algorithm in our comparisons with other approaches.

Our implementation of BCHD is based on the algorithms in [3]. In our implementation of BCHD, we set an initial batching interval of 1 second. The first batch will consist of the tasks that arrived in the first second. All tasks that arrive while the first batch is being processed are queued. Once the first batch is processed those tasks that have been queued from the second batch will be processed by the server. This process repeats itself as new tasks arrive.

The MONO algorithm is implemented based on the on-line monolithic-SC scheme. MONO is similar to AUC as both process a task as soon as it arrives at the server. However, unlike AUC, in MONO the server is responsible for both the scheduling and matching phases of the process.

2) **Configurations and Metrics:** In our experiments, we evaluate two different aspects of our framework. First, we compare the assignment rate of the proposed algorithms, i.e., the percentage of tasks that are completed. We compare the assignment rate of AUC with those of BCHD and NN. The reason we do not include the results of MONO is that this algorithm uses the same heuristics as AUC. Consequently with regard to the assignment rate, the results of MONO is similar to those of AUC. Next, we focus on the scalability of Auction-SC. We compare the average processing time of a single task in AUC to those of BCHD, NN and MONO.

Since AUC is a distributed algorithm, to account for its communication cost, for each incoming task, we compute the size of the messages transferred between the SC-Server and workers. We assume Auction-SC is running over a 3G cellular network with a transmission speed of $100Kb/s$. We divide the transmission speed by the size of the message and get the transmission time of each message. All experiments were conducted on an Intel(R) Core(TM)2 Duo 3.16GHz PC with 4GB of memory and 500GB of disk running Windows 10. Methods were implemented in Java.

**C. Assignment Rate**

In this section, we evaluate the assignment rate of AUC and compare it with those of BCHD and NN. First, we compare the algorithms using our real-world and synthetic dataset. Subsequently, using the synthetic datasets, we show how varying spatial and temporal parameters of the problem can affect the assignment rate.

![Fig. 6: Assignment Rate of Real-Time Approaches](image)

First we compare the assignment rate of different algorithms. Figure 6 depicts that AUC outperform NN by almost 25%. The main reason is that AUC performs the scheduling and matching phases in tandem. However, NN is an SOM approach where the schedule of the worker is not considered when a task is matched with him. Consequently, it is likely that a task gets assigned to a worker while the worker is not able to schedule it and thus, the task does not get completed. Furthermore, AUC outperform BCHD by almost a similar margin. This is because BCHD performs the matching phase, the schedule of the worker is not considered and hence, a task might end up getting matched to and scheduled for a worker that was not the best worker. This in turn can lower the chances of that worker to get assigned to a new task in the future. The second reason is that while a task is waiting at the server to get processed with the next batch, depending on the length of the batching interval, it will lose some portion of its available time before its deadline, which in turn, can lower the...
chances of the task fitting a worker’s schedule (more details in Section VI-D).

![Fig. 7: Assignment Profile-Varying Worker/Task Arrival Rates](image)

In order to study the effect of temporal parameters of SC, we ran several experiments using different pairs of task arrival rates ($t_{\text{rate}}$) and worker arrival rates ($w_{\text{rate}}$). Figure 7 illustrates the effect of increasing $t_{\text{rate}}$ and $w_{\text{rate}}$ on the assignment rate. The level of grayness corresponds to the percentage of completed tasks with black and white representing 100% and 0%, respectively. With small number of workers, $ast_{\text{rate}}$ increases, the percentage of completed tasks decreases where at the top left corner of each plot we get close to 0%. On the other hand with small number of incoming tasks, as we increase $w_{\text{rate}}$, eventually all tasks will be completed.

To better evaluate the difference between alternative algorithms, in Figures 8(a) and 8(b) we performed a pair-wise comparison of different algorithms by taking their task completion rates. For example, Figure 8(a) shows the difference between AUC and NN. We observe that all approaches perform similarly at the two extreme cases discussed in Figure 7, i.e., high task-low worker and low task-high worker. AUC outperforms NN and BCHD up to 30% when the problem is more complex, i.e., outside the extreme cases. An interesting observation in Figure 8(a) is that AUC outperforms NN by a larger margin at scale (higher $t_{\text{rate}}$ and $w_{\text{rate}}$). The reason is that with higher $t_{\text{rate}}$ and $w_{\text{rate}}$ more workers are moving around and more tasks arrive and leave so in general the spatiotemporal dynamism of the system increases. AUC takes advantage of the dynamism by guaranteeing a task gets assigned to a worker that can complete it. On the contrary, NN ignores the schedule of a worker during matching and this becomes more important as there is more dynamism in the system.

In Section V we mentioned that in cases where the VST tree grows exponentially, we can replace Algorithm 3 with a polynomial time approximate algorithm. In the next experiments we replaced Algorithm 3 with the insertion algorithm from [14] that runs in $O(n^2)$ (ApproxBI). Figure 8(c) depicts the change affects the assignment rate by less than only 5%.

### D. Scalability

The last set of experiments focus on comparing the scalability of AUC with BCHD and MONO. We can measure the scalability of a SC systems by the average time required to process a task which in turn indicates the throughput of the system. Figure 9 illustrates the average processing time of a single task in different algorithms for different worker arrival rates. As shown in Figure 9, as the arrival rate of workers increases, the average processing time of a single task in AUC does not change since AUC utilizes the workers for scheduling an incoming task. On the contrary, with MONO and BCHD, in addition to performing matching, the server also performs scheduling for all the workers. Consequently, with these algorithms, the average processing time of a single task grows as we increase the number of workers and is several orders of magnitude higher than that of AUC.

To provide a more practical perspective, in Figure 10 we compare the scalability of different approaches given the current minimum requirements of a ride sharing application in New York City [10]. As shown, while MONO and BCHD cannot satisfy the current requirements, AUC can scale much higher than what currently is needed.

To summarize the results of our experiments, we showed that SOM approaches cannot generate high assignment rates since they do not consider the schedule of workers when assigning tasks. With batched approaches, tasks are not necessarily assigned to the best workers and long batching intervals reduce the chance of new tasks fitting in a workers schedule. On the other hand, the immediate assignment of tasks along with consideration of scheduling at the time a task is being
while scaling orders of magnitude higher than both batched and on-line monolithic-SC methods.

In this paper, we assumed each task can be performed instantaneously, e.g., taking a picture. Once that assumption is relaxed, we will face new challenges with scheduling the tasks. We also assumed each task requires the worker to travel to a single location. However, in other applications the worker may need to visit multiple locations for a single task, e.g., in the Uber application the worker has to pick up a passenger at one location and drop him off at a second location. We plan to extend our Auction-SC framework to incorporate these two features.

VII. Conclusion and Future Work

We studied the problem of on-line task assignment in spatial crowdsourcing. We introduced an auction-based framework where tasks get processed one at a time, the server has to perform scheduling for large number of workers is time consuming and prevents on-line monolithic-SC approaches from scaling. However, with Auction-SC, tasks are processed one at a time (i.e., fast matching) and the server utilizes the workers for performing scheduling (i.e., fast scheduling).

REFERENCES