Task Matching and Scheduling for Multiple Workers in Spatial Crowdsourcing

Dingxiong Deng
Dept. of Computer Science
Univ. of Southern California
dingxiod@usc.edu

Cyrus Shahabi
Dept. of Computer Science
Univ. of Southern California
shahabi@usc.edu

Linhong Zhu
Information Science Institute
Univ. of Southern California
linhong@isi.edu

ABSTRACT

A new platform, termed spatial crowdsourcing, is emerging which 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). The current approach is to formulate spatial crowdsourcing as a matching problem between tasks and workers; hence the primary objective of the existing solutions is to maximize the number of matched tasks. Our goal is to solve the spatial crowdsourcing problem in the presence of multiple workers where we optimize for both travel cost and the number of completed tasks, while taking the tasks' expiration times into consideration. The challenge is that the solution should be a mixture of task-matching and task-scheduling, which are fundamentally different. In this paper, we show that a baseline approach that performs a task-matching first, and subsequently schedules the tasks assigned per worker in a following phase, does not perform well. Hence, we add a third phase in which we iterate back to the matching phase to improve the assignment per the output of the scheduling phase, and thus further improves the quality of matching and scheduling. Even though this 3-phase approach generates high quality results, it is very slow and does not scale. Hence, to scale our algorithm to large number of workers and tasks, we propose a *Bisection-based* framework which recursively divides all the workers and tasks into different partitions such that assignment and scheduling can be performed locally in a much smaller and promising space. Our experiments show that this approach is three orders of magnitude faster than the 3-phase approach while it only sacrifices 4% of the results' quality.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*

Keywords

spatial crowdsourcing, task matching and scheduling, scalability

1. INTRODUCTION

As the number of smartphones increases and their technology advances, the opportunities to leverage these devices to positively affect human behavior expand. For example, to improve the food delivery experience, a user could simply tap into a crowdsourcing application, "FAVOR" [10] and request food delivery from any restaurant. Once receiving this request, the server could assign it to any available nearby "runner" to pick up the food at the required store. Thus, we are witnessing the emergence of a new platform, termed spatial crowdsourcing, for anyone to submit requests for real-world tasks, tagged with time and location, and then distribute these tasks among people with smartphones in the vicinity of the tasks, who are willing to accept the task and may receive monetary reward for completing it. Typically, a spatial crowdsourcing platform consists of workers, spatial tasks (i.e., tasks related to a location), and users who request tasks. Its goal is to crowdsource a set of requested spatial tasks to available workers, which requires the workers to physically go to those locations in order to perform the tasks. For example, in the "FAVOR" platform, each online runner is an available worker, and each food delivery request is a spatial task related to a restaurant location. Spatial crowdsourcing has application in numerous domains such as transportation (e.g., Uber), journalism [18, 19], and business intelligence (e.g., Gigwalk [13] and TaskRabbit [23]).

The challenge is that whether these spatial crowdsourcing platforms can provide both an efficient and effective mechanisms to support users' demands at scale [12]. Several existing approaches [1, 8, 15, 16, 22, 24] have focused on the optimization of spatial task assignment. For instance, Kazemi and Shahabi [15] formulate the spatial crowdsourcing as a matching problem between tasks and workers with the primary objective of maximizing the number of matched tasks. However, a shortcoming of these approaches is that they ignored the fact that maximizing the number of matched tasks is not necessarily equivalent to maximizing the number of completed tasks once the additional travel costs associated with moving to tasks locations, and the expiration time of tasks are taken into account. On the contrary, Deng et al. [9] formulated spatial crowdsourcing as a scheduling problem in which the goal is to maximize the number of completed tasks per worker. They made the simplifying assumption that each worker has a set of pre-assigned tasks in order to focus on the scheduling optimization for a single worker.

To address both of the above shortcomings, we propose a new formulation of spatial crowdsourcing as a combination of two optimization problems: task-matching and task-scheduling. Our objective is to solve the spatial crowdsourcing problem in the presence of multiple workers where we optimize for both total travel cost and the number of completed tasks, while taking the tasks expiration times into consideration. One challenge is that the solution is a mixture of task-matching and task-scheduling, while existing approaches in spatial crowdsourcing work on either task-matching [15,16] or task-scheduling [9], but not both. One straight-

SIGSPATIAL'15, November 03-06, 2015. Bellevue, WA, USA
 ISBN 978-1-4503-3967-4/15/11 ...$15.00.
DOI: http://dx.doi.org/10.1145/2820783.2820831

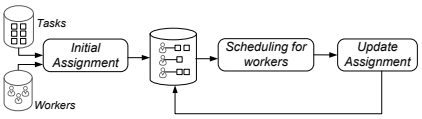


Figure 1: The 3-phase framework

forward idea is to take a good matching approach from [15] to perform the global task-matching, and then perform the task-scheduling for the tasks assigned to each worker utilizing a scheduling heuristic from [9]. However, this baseline method does not consider the possibility of re-matching and re-scheduling for the current/remaining workers and tasks, thus produces poor scheduling results.

To overcome this issue, we propose a 3-phase framework, termed Global Assignment and Local Scheduling (GALS), that incorporates the idea of the baseline approach, which contains matching and scheduling as the first two phases, but with an additional third phase in which we iterate back to the matching phase to improve the assignment per the output of the scheduling phase (See Figure 1). We also provide an insertion heuristic such that the newly assigned tasks can be incrementally inserted into a partial schedule, which speeds up the computation as compared to a re-scheduling approach.

Even though GALS could achieve high quality results in terms of matching and scheduling, it does not scale to large number of tasks and workers due to the bottleneck of global assignment. Note that due to the online nature of large-scale crowdsourcing system, where requests (i.e., tasks) need to be matched and scheduled in real-time with short response time, the online response-time of the proposed solution should scale as the number of on-line workers and tasks grows.

To speed-up GALS, we need to break the global assignment phase (the bottleneck) to a set of local assignments. Intuitively, on average a worker is scheduled with only a small number of tasks, and each task has a relatively small number of candidate workers. Therefore, for each worker, we only need her/his nearby workers and tasks to obtain a good scheduling. This motivates us to propose a *Task-oriented partitioning* algorithm, which segments the large flow network into smaller partitions and thus the Local Assignment and Local Scheduling (LALS) can be completed in a much smaller but promising search space.

Nevertheless, the conventional LALS framework still encounters efficiency bottleneck arising from two problems: the *Stragglers problem* across the partitions and the *Residual problem*. The Stragglers problem refers to the phenomenon where a small number of partitions have much heavier workloads and take significantly longer than the others to complete. On the other hand, after scheduling for each partition, the aggregated number of remaining workers and tasks (i.e., the residual) might be large, which leads to high computation cost. We thus propose a *Bisection-based framework* that iteratively performs a top-down recursive bisection procedure to achieve the balanced partitioning and a bottom-up merge to avoid the residual problem. Through this process, our algorithm not only achieves near-optimal results as GALS, but also scales to a massive number of workers and tasks. We also develop an analytical method that automatically determines the parameter (i.e., the size of one partition) of the bisection framework.

Since to the best of our knowledge, this is the first study of the combination of task assignment and scheduling in spatial crowdsourcing, we did not have competitive approaches to which compare LALS (besides the baseline and GALS approaches). However, recently other partitioning approaches have been proposed with a different focus of improving the efficiency of task assignment in spatial crowdsourcing [1, 8]. Thus, these approaches do not consider the balanced workloads between different partitions and the connectivity information between tasks and workers, which are critical for efficiency and the quality of the scheduling phase (see more detailed discussion in Section 6). Nevertheless, in Section 5, we adapt these partitioning approaches to our problem setting and show that our partitioning technique is at least one order of magnitude faster. Our experiments also show that while GALS outperforms the baseline by up to 30% in terms of the number of completed tasks, our BisectionLALS algorithm is three orders of magnitude faster than GALS by sacrificing only 4% of the matching+scheduling quality.

The remainder of this paper is organized as follows. In Section 2, we formally define our Maximum Task Scheduling problem with Multiple Workers (MTSMW). In Section 3, we present the Global Assignment and Local Scheduling (GALS) framework to solve the problem. We propose the Bisection-based Local Assignment and Local Scheduling work in Section 4. Experiment results are reported in Section 5. In Section 6, we review the related work and Section 7 concludes the paper.

2. PROBLEM DEFINITION

We use s to denote a spatial task with location l_s and deadline d_s. A worker, denoted by w, is a person who volunteers to perform spatial tasks. Each online worker w is associated with location l_w, current time instance t_w, capacity q_w, and a spatial region g_w.

With spatial crowdsourcing, a spatial task s can be completed by a worker only if the worker is physically located at the location l_s. Considering the expiration time, a spatial task s can be completed only if the worker arrives at l_s before its deadline d_s. In addition, we use a task sequence $R = (s_1, s_2, \cdots, s_m)$ to denote a task scheduling for a worker. The size of a task sequence, denoted by $|R|$, is the number of tasks in R. We say a task sequence R satisfies a worker's capacity constraint if $|R| < q_w$. Moreover, we use $\texttt{set}(\mathtt{R})$ to denote the set interpretation of the sequence R, and R is a finite sequence over $\texttt{set}(\mathtt{R})$.

With the above notation, given a worker w and his/her task scheduling $R = (s_1, s_2, \cdots, s_m)$, the arrival time of w at task s_i in R is defined as follows:

$$a(s_i) = \begin{cases} a(s_{i-1}) + c(s_{i-1}, s_i) & \text{if } i \neq 1 \\ t_w + c(w, s_1) & \text{if } i = 1 \end{cases}$$

where $c(a, b)$ is the travel cost from the location of a to the location of b.

In addition, for a worker w, the travel cost of a task sequence R, denoted by $\delta(R)$, is the travel cost of visiting all the tasks. That is,

$$\delta(R) = c(w, s_1) + c(s_1, s_2) + \cdots + c(s_{m-1}, s_m)$$

Definition 1. Given an instance set $I_t = \{W_t, S_t\}$, where W_t is a set of workers and S_t is a set of tasks in time t, a **planning** $\mathcal{P}$ is a set of task sequences $\{R_{w_1}, R_{w_2}, \cdots, R_{w_n}\}$ for all the workers, where $n = |W_t|$, visiting each of the task at most once, i.e.,

$$\begin{cases} \texttt{set}(R_{w_i}) \cap \texttt{set}(R_{w_j}) = \emptyset \quad (1 \leq i < j \leq |W_t|) \\ \bigcup_{i=1}^{p} \texttt{set}(R_{w_i}) \in S_t \end{cases}$$

Figure 2 shows an example with 5 spatial workers and 8 spatial tasks. One potential planning is $R_{w_1} = (s_1)$, $R_{w_2} = (s_3, s_2)$, $R_{w_3} = (s_4)$, $R_{w_4} = (s_6, s_7)$ and $R_{w_5} = (s_5, s_8)$.

Based on the above definitions, we now formally define the problem of Maximum Task Scheduling with Multiple Workers (MTSMW):

PROBLEM 1. *Given the instance set $I_t = \{W_t, S_t\}$, the solution of MTSMW is to find a planning $\mathcal{P} = \{R_{w_1}, R_{w_2}, \cdots, R_{w_n}\}$ for all the workers that satisfies each worker's capacity, spatial constraint and the tasks' expiration time constraint; i.e.,*

$$\begin{cases} |R_{w_i}| \leq q_{w_i} \quad (1 \leq i \leq |W_t|) \\ \texttt{set}(\texttt{R}_{\texttt{w}_\texttt{i}}) \in \texttt{g}_{\texttt{w}_\texttt{i}} \quad (\texttt{1} \leq \texttt{i} \leq |\texttt{W}_\texttt{t}|) \\ a(s) \leq d_s \quad (\forall s \in \bigcup_{i=1}^{p} \texttt{set}(\texttt{R}_{\texttt{w}_\texttt{i}})) \end{cases}$$

The MTSMW problem aims to find a solution P which (1) maximizes the number of completed tasks (i.e., $\sum_i |R_{w_i}|$), and (2) minimizes the sum of the average travel cost per task over all workers (i.e., $\sum_i \frac{\delta(R_{w_i})}{|R_{w_i}|}$). Note that the first goal is the primary goal, and we use the travel cost as tiebreaker, i.e., in case of ties in the number of completed tasks, we choose the solution with minimal travel cost.

In this work, we consider the model of Server-Assigned-Task (SAT) [15], in which the spatial crowdsourcing server is responsible for calculating the mathcing and scheduling the tasks. For ease of presentation, we summarize our notations used throughout this paper in Table 1.

Table 1: Notations and explanations

Notations	Explanations
w, s	a spatial worker and spatial task
l_w, l_s	the spatial location of worker w and task s
q_w, g_w	the capacity, spatial region of worker w
t_w	the start time instance of worker w
d_s	the deadline of task s
$a(s)$	the arrive time at task s
$c(a, b)$	the travel cost between location a and b
R	a route sequence
$\delta(R)$	the travel cost of the route R
$\texttt{set}(\texttt{R})$	the spatial tasks contained in R
S_w	tasks that are assigned to worker w at one iteration
W, S	a worker set and task set
$\|W\|, \|S\|$	number of workers and tasks in W and S
$\mathcal{P}$	a planning for all the workers

Proof of NP-hard. It has been proven in [9] that the maximum task scheduling (MTS) problem with single worker and single objective is NP-hard, thus the MTSMW problem with multiple workers and two objectives is also NP-hard.

Remark: In this paper, we use matching and assignment interchangeably. However, in general they are not equivalent to each other.

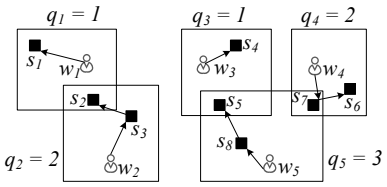


Figure 2: Example

3. THE GALS ALGORITHM

In this section, we discuss our heuristic algorithm, namely, Global Assignment and Local Scheduling algorithm (**GALS**), to solve Problem 1. We first discuss the framework of GALS as follows:

1. Initial task assignment: Find a matching for the set of workers W and the set of tasks S. With the matching, a task s is assigned to a worker w if they are matched with each other.
2. Local scheduling: For each worker w and her/his assigned tasks S_w, if there is no schedule for this worker, we make a new schedule; Otherwise, we update her/his current schedule.
3. Update assignment: Update the flow network with the updated scheduling and consequently refine the matching between workers and tasks.
4. Repeat 2–4 until it meets the termination condition.

In the rest of this section, we provide the design details of each component.

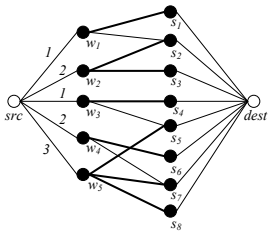


Figure 3: Flow network with 5 workers and 8 tasks, a source and destination vertices are added. We connect one edge between a (w, s) pair only if s is in the spatial region of w. The capacity for the edges from src to w is q_w, and the capacity for the remaining edges is 1.

3.1 Initial Assignment module

In the assignment module, we adopt the similar techniques proposed by Kazemi et al. [15] to solve the maximum task assignment problem. That is, we formulate the matching problem as a maximum flow problem, and find the matching of worker and task pairs from the flow. We could also associate each edge with a cost (e.g., travel time or distance) in the flow network, thus min-cost-maximum-flow algorithm could be used. We outline the assignment module in Algorithm 1.

Algorithm 1 Matching(W, S)

Input: Worker set W and task set S.
Output: An assignment between W and S.
1: Construction a flow network wrt. (W, S)
2: Calculate the maximum flow or min-cost-max-flow assignment based on different cost functions
3: Find the assignment between workers and tasks from the flow

Figure 3 shows the example of the corresponding flow network from Figure 2. One solution to the max flow problem in this flow network is shown as the bolded edges. With respect to this max flow, an initial assignment is retrieved as $S_{w_1} = \{s_1\}, S_{w_2} = \{s_2, s_3\}, S_{w_3} = \{s_4\}, S_{w_4} = \{s_6\}$ and $S_{w_5} = \{s_5, s_7, s_8\}$. The reason we use the max-flow algorithm for assignment is that we observed the assignment phase is the most determining factor in terms of the number of completed tasks through our experiment. As an example, in Figure 2, if initially s_2 is assigned and scheduled to w_1, then s_1 cannot be completed because w_1 is the only candidate worker but he does not have any more capacity. Another reason is due to the property stated in the following Lemma.

LEMMA 1. *The maximum-flow value f_v is the upper bound of the number of completed tasks.*

Proof (Sketch): Suppose f^* tasks are completed in the optimal scheduling, then the corresponding scheduling could form a valid assignment in the flow network. From the definition of maximum flow, we know that $f^* \leq f_v$, therefore, f_v is the upper bound of the number of completed tasks. ■

3.2 Scheduling module

The basic idea of our scheduling module is to construct a new scheduling based on the existing scheduling, rather than re-scheduling from scratch. This is because one worker would possibly already have a partial schedule when he/she is in the process of GALS algorithm. Specifically, given a worker w with the assigned tasks S_w and existing scheduling R_w (R_w could be empty), the purpose of the scheduling module is to insert as many tasks in S_w as possible into R_w with *minimum travel cost*. In the following we first introduce the insertion feasibilities and then present the details of the insertion heuristic which is used for creating a new scheduling as well as updating an existing scheduling.

Insertion Feasibilities. To simplify our problem, we assume that when we are inserting a task into an existing scheduling $R_w = (s_1, s_2, \cdots, s_m)$, the order of the existing tasks in R_w does not

change. Unfortunately, even with this assumption, not every task in S_w is feasible to be inserted. This is because inserting a new task between s_{j-1} and s_j could potentially change the arrival times of the subsequent tasks $(s_{j+1}, \cdots, s_m)$, which causes some scheduled tasks after s_j become impossible to complete. Therefore, the insertion feasibility check evaluates whether a task insertion increases the number of completed tasks (feasible), or in the opposite does not increase the number of completed task (infeasible). Specifically, our insertion feasibility check is built upon the following Lemma:

LEMMA 2. *The necessary conditions for inserting a new task s between s_{j-1} and s_j on a partial feasible route $R = (s_1, s_2, \cdots, s_m)$ are (assume each task takes zero time to be completed):*

$$\texttt{td} + a(s_i) \leq d_{s_i}, \text{ when } j \leq i \leq m$$

where $\texttt{td}$ denotes the travel delay caused by inserting task s, that is, $\texttt{td} = c(s, s_{j-1}) + c(s_j, s) - c(s_{j-1}, s_j)$.

Insertion scheduling. As outlined in Algorithm 2, in our incremental scheduling, at each iteration, for each unscheduled task $s \in S_w$, we scan all the possible $|R_w|$+1 positions in the current partial task sequence R_w, to find the spot with the *minimum travel delay* of inserting s while keeping its feasibility. The algorithm for finding the best insertion place in a partial route is listed in Algorithm 3.

Time complexity of insertion heuristic. In order to insert tasks from S_w into a partial scheduling R_w, the function FindBestInsertionPlace is called at most $|S_w|^2$ times and each with a cost of $|R_w|$, therefore the total time complexity is $O(|S_w|^2|R_w|)$. Although the time complexity is quadratic, in practice it is still efficient because the number of assigned tasks for each worker is a relatively small value, and is basically decreasing over each iteration.

Algorithm 2 InsertionScheduling(w, R_w, S_w)

Input: A worker w, a partial route R_w and a task set S_w.
Output: An updated task sequence R_w for w.

```
1: repeat
2:    for all unscheduled tasks s ∈ S_w do
3:       (C_s, pos_s) ← FindBestInsertionPlace(w, R_w, s)
4:    (s_ε, pos_ε) ← arg min C_s, ∀s ∈ S_w
5:    Insert s_ε into the pos_ε of R_w
6: until no more tasks could be inserted into R_w
```

Algorithm 3 FindBestInsertionPlace(w, R_w, s)

Input: A worker w, a partial task sequence R_w and a task s.
Output: The insertion place `bestPos` and the travel delay `minCost`.

```
1: /* Assume R_w = (s_1, s_2, ··· , s_m) */
2: for i ← 0 to m do
3:    if i == 0 then
4:       cost ← c(w, s_1)
5:    else if i == p then
6:       cost ← c(s_m, s)
7:    else
8:       cost ← c(s_i, s) + c(s, s_{i+1}) − c(s_i, s_{i+1})
9:    Check the feasibility according to Lemma 2
10:   if cost < minCost then
11:      minCost ← cost
12:      bestPos ← i
13: return minCost and bestPos
```

3.3 Assignment update module

Once we complete the scheduling phase for those workers with the assigned tasks, we iterate back to the assignment phase: We build the corresponding flow network between the remaining workers and tasks, and calculate the new assignment. However, we also need to maintain a set of forbidden worker-task pairs to avoid duplicate task assignment that has occurred in the previous iterations. The details of our update algorithm are outlined in Algorithm 4. We use RWS as the remaining worker set and RTS as the remaining task set (Lines 1–2). For each worker w with newly updated scheduling, we check the capacity of w, and insert it into RWS if w still has available capacity (Lines 4–5). In addition, for those tasks that are assigned to w but could not be scheduled via our insertion heuristic, we add them into RTS and mark the matching (w, s) as the forbidden matching pair (Lines 7–9). Once we build the flow network with respect to RWS, RTS and the forbidden matching pairs, we calculate the maximum flow and find the new assignment for the next iteration (Lines 10–12).

Example of GALS: *From Section 3.1, an initial max-flow assignment of Figure 2 is $S_{w_1} = \{s_1\}, S_{w_2} = \{s_2, s_3\}, S_{w_3} = \{s_4\}, S_{w_4} = \{s_6\}$ and $S_{w_5} = \{s_5, s_7, s_8\}$, subsequently GALS makes a scheduling for each worker. Suppose only task s_7 cannot be completed by worker w_5, then the worker-task pair(w_5, s_7) is marked as forbidden for the next iteration. Consequently a new flow network between w_4 and s_7 is built because s_7 is in w_4's spatial region and w_4 has one more capacity, finally s_7 is scheduled into w_4's existing scheduling as shown in Figure 2.*

Algorithm 4 UpdateFlowNetworkAndAssignment()

Output: An assignment A between RWS and RTS.

```
1: RWS ← the unassigned workers in the last assignment
2: RTS ← the unassigned tasks in the last assignment
3: for each w with newly assigned tasks do
4:    if q_w > |R_w| then
5:       RWS ← RWS ∪ w
6:    /* task s has not been scheduled into R_w */
7:    if s ∈ S_w AND s ∉ R_w then
8:       RTS ← RTS ∪ s
9:       Add (w, s) into the forbidden matching pairs
10: Build flow network wrt. (RWS, RTS) and the forbidden pairs
11: Calculate the maximum flow
12: Find the task assignment for RWS
```

Table 2: Time complexity analysis for the GALS Algorithm, $|W^r|$ ($|S^r|$) is number of workers (tasks) at iteration r, $|E^r|$ is number of edges in the flow network, and $|S^r_w|$ is number of tasks assigned to a worker w.

Operation	Complexity
Initial task assignment	$O((\|W\| + \|S\|)^2 \cdot \|E\|)$
Schedule for one worker	$O(\|S^r_w\|^2 \|R^r_w\|)$
Update flow network and assignment	$O((\|W^r\| + \|S^r\|)^2 \|E^r\|)$

3.4 Complexity analysis of GALS

Termination condition: The iteration terminates when the number of edges in the remaining flow network is zero (i.e., there exists no mapping between RWS and RTS). The flow network starts with at most $|W| \cdot |S|$ edges, at each iteration the number of edges is decreasing because we are updating the scheduling of workers and/or marking some worker-task pairs as forbidden, therefore eventually the algorithm would terminate.

Complexity analysis: Table 2 summarizes the cost of each operation in one iteration of our GALS algorithm. Given a flow network with $|V|$ vertices and $|E|$ edges, the cost of finding a maximum flow is $O(|V|^2|E|)$ by $IBFS$ algorithm [14]. Therefore, the time complexity of our GALS algorithm is $O\Big(I + \sum_r \big((|W^r| + |S^r|)^2|E^r| + \sum_{k=1}^{|W_r|} |S^r_k|^2 |R^r_k|\big)\Big)$, where I is the initial task assignment cost. Clearly the cost is dominated by the initial global assignment phase and assignment update phase.

4. THE LALS ALGORITHM

In Section 4.1, we first propose a Task-oriented partitioning and describe how it is used in a Naïve Local Assignment Local Scheduling (LALS) framework to reduce the bottleneck of global assignment in GALS. We then present the Bisection-based LALS framework to further improve its efficiency.

Algorithm 5 TaskOrientedPartitionGeneration($curW$, $curS$, θ)

Input: the current worker set ($curW$), the current task set ($curS$) and a threshold value θ.
Output: A single partition p with PWS and PTS.

```
1: PWS ← ∅, PTS ← ∅
2: WS ← ∅, TS ← ∅
3: s ← randomly choose a seed task, TS ← TS ∪ s
4: workload ← 0
5: while TRUE do
6:    for all s ∈ TS do
7:       for all w ∈ W_s (workers could complete task s) do
8:          if w ∈ curW AND w ∉ PWS then
9:             add w into WS and PWS
10:            workload ← workload + 1
11:   TS ← ∅
12:   for all w ∈ WS do
13:      for all s ∈ T_w (tasks in the region w) do
14:         if s ∈ curT AND s ∉ PTS then
15:            Add s into TS and PTS
16:   WS ← ∅
17:   if workload ≥ θ or No worker(task) is left then
18:      return (PWS, PTS) as a partition
19:   if TS is empty then
20:      s ← findNextCloseTask()
21:      TS ← TS ∪ s
```

4.1 Task-oriented partitioning

Various partitioning techniques [3, 4, 26] have been proposed to find a balanced partitioning that minimizes the number of edge cuts for arbitrary graph; however, our flow network is essentially a bipartite graph and our objective is to segment the flow network into different partitions with balanced workloads (i.e., number of edges) as well as preserve the spatial properties. Thus, we aim to divide tasks and their relevant workers (i.e., close in location and also connected in the flow network) into the same partition.

The intuition for our Task-oriented partitioning to generate a single partition is similar to "blow balloons" as shown in Figure 4: Starting from an empty partition (empty balloon), we randomly select a task, add it into the partition, and grow the partition with its nearby workers and tasks until the workload of this partition (the number of edges in this sub network) reaches a threshold (a full balloon). Given the current available worker and task set, Algorithm 5 presents the details of generating a partition with the workload (edge number) constraint. The partition is initialized with an empty set of workers PWS and set of tasks PTS (Line 1). We also initialize TS with a randomly chosen task (Lines 2–4). Subsequently from each iteration, we expand from each task $t \in TS$ to its nearby workers and increase the workload (Line 6–10). Similarly, we grow the partition from the newly added workers in WS (Lines 12–15). This process continues until the workload passes the threshold. The expansion strategy outlined in Lines 6–10 and Lines 12–15 guarantees that tasks and workers in a partition are centered on the seed task of this partition.

NaïveLALS algorithm and its complexity. With a list of generated partitions through Algorithm 5, the idea of the NaïveLALS algorithm is to use GALS to make scheduling for each partition as well as the remaining workers and tasks.

Suppose the partitioning cost is J, the time complexity of using GALS for one partition p is H_p and the cost of scheduling for the remaining workers and tasks is H_r, then the total cost of Naïve-LALS is $O(J + \sum_p H_p + H_r)$. In terms of the partitioning cost, since it only requires one single pass of the entire network and additional costs to call the function `findNextClosestTask`, then the computational cost of Task-oriented partitioning is $O(|W| + |S| + |E| + kF)$, where F is the computational cost of `findNextClosestTask`, and k is the number of function calls. Through this way, the large cost of global assignment I in GALS is divided into a set of local assignments, thus improves the efficiency.

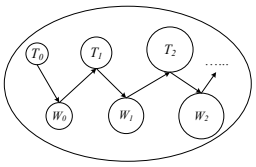


Figure 4: Task-oriented partition: The partition extends from T_0 to its one-hop relevant worker set W_0, then W_0 extends to T_1, and so on. The size of circle represents the relative size of worker/task set.

4.2 Bisection-based LALS framework

Even though NaïveLALS improves the overall efficiency, it still encounters the bottleneck due to two problems: i.e., the Stragglers problem among partitions and the Residual problem. The Stragglers problem refers to the case where a small number of partitions have much heavier workloads and thus take significantly longer running time than the others to complete. In the Task-oriented partitioning, it is highly possible to generate gigantic partitions because of the one-hop expanding strategy where both the worker and task set simultaneously grow. The residual problem occurs when, after running GALS for each partition, the aggregated number of remaining workers and tasks is large, which leads to high computation cost for assigning remaining workers and tasks.

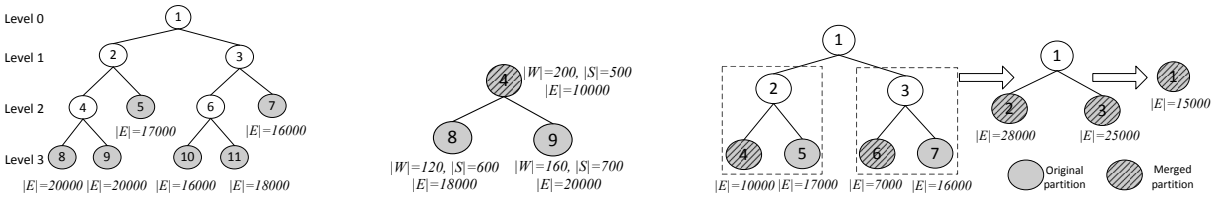
To address the Stragglers and the Residual problems, we propose a Bisection-based LALS mechanism. Our algorithm recursively partitions the workers and tasks into two equal-sized parts until the workload is less than a predefined threshold, and then merges the remaining workers and tasks in a bottom-up fashion after assignment and scheduling. Through this process, our algorithm not only achieves similar matching-scheduling results as GALS, but also scales to large number of workers and tasks. We first outline the **BisectionLALS** mechanism in Algorithm 6, and then present the details of recursive bisection and bottom-up merging.

Algorithm 6 BisectionLALS (W, S, θ)

Input: Worker set W, task set S and the threshold value θ.
Output: A planning $\mathcal{P}$ for W.

```
1: curW ← W, curS ← S
2: curWorkload ← |E|
3: while curWorkload > θ do
4:    partitions ← RecursiveBisection (curW, curT, curWorkload, θ)
5:    BottomUpMerging (partitions, θ)
6:    Update curW, curT, curWorkload
7: if curWorkload > 0 then
8:    Schedule for curW and curT with GALS
```

Recursive top-down bisection partitioning. The recursive top-down bisection partitioning is built upon the intuition that it is easy to control and balance the partition size if we only generate two partitions. The idea is outlined in Algorithm 7. We initially divide the current flow network into two equal-sized parts through the Task-oriented partitioning, then they are recursively divided into two until the workload is less than a predefined threshold. With the recursive bisection, we could generate a list of balanced partitions, and the workload of each partition is less than the threshold value to guarantee that local assignment and scheduling in each partition can be efficiently computed.



(a) Top-down bisection (b) Merging two partitions (c) Bottom-up merging

Figure 5: Bisection-based mechanism with threshold value $\theta = 30,000$

Figure 5(a) depicts an example of a partitioning tree corresponding to the recursive bisection algorithm. Given two thousand workers, ten thousand tasks and four hundred thousand edges, and the threshold value 30,000, the recursive bisection returns 6 partitions in which workload is ranging from $16,000 - 20,000$.

Algorithm 7 RecursiveBisection($curW$, $curS$, $curWorkload$, θ)

Input: Current worker set $curW$, current task set $curS$, current workload and the threshold value θ.
Output: partitions

1: $\theta \leftarrow curWorkload/2$
2: **if** curWorkload $>$ threshold **then**
3: $p_L, p_R \leftarrow$ TaskOrientedPartitionGeneration(curW, curS, θ)
4: **if** p_L.workload $> \theta$ **then**
5: RecursiveBisection(p_L.W, p_L.S, p_L.workload, θ)
6: **else**
7: Add p_L into partitions
8: **if** p_R.workload $> \theta$ **then**
9: RecursiveBisection(p_R.W, p_R.S, p_R.workload, θ)
10: **else**
11: Add p_R into partitions
12: **return** partitions

Algorithm 8 BottomUpMerging(partitions, θ)

Input: partitions and threshold θ
Output: A merged partition

1: **while** there exists more than one partition **do**
2: Combine the last two sibling partitions p_L and p_R at the bottom level as p_{merge}
3: **if** p_{merge}.workload $> \theta$ **then**
4: Schedule for p_L and p_R using GALS
5: $p_{merge} \leftarrow$ remaining workers and tasks from p_L and p_R
6: Insert p_{merge} into the corresponding positions of the partition list

Bottom-up merging. To address the residual problem, we propose a bottom-up merging approach, which fully utilizes the spatial properties of the binary tree generated by the above bisection partitioning algorithm. Our intuition is based on the fact that the sibling partitions of the binary tree are spatially close to each other, and thus merging remaining workers and tasks from sibling partitions guarantees the quality of local assignment and scheduling. Therefore, the basic operation of our bottom-up merging procedure is to merge and/or schedule for two sibling partitions. Specifically, if the combined workload of two sibling partitions is larger than the predefined threshold, we do a local assignment and local scheduling for each partition, and merge the remaining workers and tasks as a new partition; otherwise, we only merge these two sibling partitions as a new partition. In both cases the newly generated partition is inserted into the existing partitioning tree. Figure 5(b) illustrates the process of scheduling and merging. Because the combined workload of sibling partitions 8 and 9 is larger than the threshold $30,000$, thus local assignment and scheduling is performed for both 8 and 9. We then merge the remaining workers and tasks into a new partition 4 with fewer number of workers and tasks.

The pseudo-code of BottomUpMerging is outlined in Algorithm 8. We keep merging and running local assignment and scheduling for partitions located at the bottom level (Lines 2-5) until only one root partition is left. Figure 5(c) explains this procedure, in which the grayed patterned nodes represent the merged partitions. Initially, as illustrated in Figure 5(b), a new partition 4 is generated by merging the remaining workers and tasks of its children partitions 8 and 9; while similarly a new partition 6 is generated by merging the remaining workloads of partitions 10 and 11. Subsequently, as a different case, because the combined workloads of 4 and 5 is less than the threshold, a new partition 2 is generated by simply merging 4 and 5 (assignment and scheduling is not performed). Finally, the procedure stops at the root node after merging the remaining workers and tasks from partitions 2 and 3. This root partition will be scheduled in the next iteration of Algorithm 6.

The bottom-up merging is efficient because it prohibits assignment and scheduling for a partition with workload larger than the threshold. In addition, the size of the output partition at root node reduces dramatically, which ensures the Bisection-based LALS terminates quickly in a few iterations.

4.2.1 Time complexity analysis

Given the worker set $|W_i|$ and task set $|S_i|$ with the number of edges $|E_i|$ at iteration i, the total cost of Algorithm 6 could be expressed as:

$$Cost = \sum_i [H_{partition}(i) + H_{merge}(i)] + H_r$$

where H_r is the cost of scheduling for the remaining worker set and task set for Lines 7–8 in Algorithm 6.

We now discuss the complexity of each part separately. Note that at the i^{th} iteration , the height of the partitioning tree $h_i = \log \frac{|E_i|}{\theta}$. Thus, we have the following:

LEMMA 3. *At the i^{th} iteration, the cost of recursive bisection $H_{partition}$ is $O((|E_i| + |W_i| + |S_i|) \cdot h_i)$.*

PROOF. At each level, we at most linearly scan the entire worker and task set with cost $|E_i|+|W_i|+|S_i|$, and the height of the partitioning tree is h_i, thus the partitioning cost $H_{partition}(i) = O((|E_i|+|W_i|+|S_i|)\cdot h_i)$. ■

LEMMA 4. *At the i^{th} iteration, the cost of bottom-up merging H_{merge} is $O((|W_p| + |S_p|)^2 \cdot |E_i|)$.*

PROOF. In the worst case, local assignment and local scheduling is performed at each node in a complete partitioning tree. That is, it requires running GALS for $O(2^{h_i})$ number of partitions. For each GALS, the cost is dominated by the assignment, which is denoted as $O(|W_p| + |S_p|)^2 \cdot \theta)$, assume that $|W_p|$ ($|S_p|$) is the number of workers (tasks) located at the partition. Therefore, we have $H_{merge} = 2^{h_i} \cdot 2O((|W_p| + |S_p|)^2 \cdot \theta) = O(2^{h_i}(|W_p| + |S_p|)^2 \cdot \theta) = O((|W_p| + |S_p|)^2 \cdot |E_i|)$. ■

Total number of iterations. We notice that the Bisection-based algorithm converges quickly: the size of flow-network reduces significantly per iteration. However, the total number of iterations is also related to the threshold value θ. and it is unclear how the workload reduces per iteration with θ. Thus, we conducted a set of empirical studies, in which we vary the threshold value θ and report how the workload varies per iteration on different datasets. Figure 6(a)

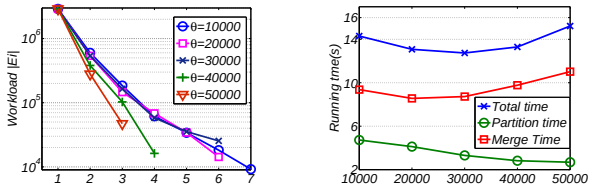
shows the experiment result on a flow network with five thousand workers, twenty-five thousand tasks and three million edges. We observe that the workload $|E_i|$ reduces exponentially as i increases, i.e.,

$$\log |E_i| = k \cdot i + \log |E| \tag{1}$$

where $|E|$ is the number of edges in the original flow network, and $k (k < 0)$ is a variable that is related to θ.

From the experiment results on different datasets which varying θ , we fit the slope k into the polynomial equation of θ, i.e., $k = \sum_m b_m \theta^m (m \geq 0, b_m$ is constant we learned from the fitting algorithm). Thus, given the initial workload $|E|$ and threshold value θ, we could predict how the workload changes per iteration.

From Algorithm 6, BisectionLALS terminates when E_i is smaller than the threshold value θ. Therefore, the total number of iterations is bounded by $O(\frac{\log \theta - \log |E|}{k}) = O(\frac{\log \theta - \log |E|}{\sum_m b_m \theta^m})$.



(a) Workload v.s. iteration i (b) Running time v.s. θ

Figure 6: Effect of θ for BisectionLALS

4.2.2 *Finding the threshold value θ*

In the following we develop an analytical method that automatically determines the threshold value θ. In general, the partitioning cost reduces as the value of θ increases because the bisection algorithm terminates earlier at the lower level of the partitioning tree and thus less cost is incurred; whereas the merging cost increases when the value of θ increases because higher value of θ leads to a higher computation cost of GALS at each larger-size partition. Figure 6(b) clearly depicts this trade-off.

Therefore, to achieve the best efficiency, we propose a cost model to derive the value of θ. In particular, we quantify the total cost $H = H_{partition} + H_{merge}$ (H_r is ignored because it is dominated by these two costs) as a expression of θ. Note that $H_{partition}(i)$ and $H_{merge}(i)$ are dependent on the workload $|E_i|$. From Equation 1, we know E_i could be expressed as:

$$|E_i| = e^{(\sum_m b_m \theta^m \cdot i + \log |E|)} \tag{2}$$

With this equation, both $H_{partition}(i)$ and $H_{merge}(i)$ could be quantified as a function of θ. Substitute Equation 2 into Lemma 3 and 4, we obtain the following:

$$H_{partition}(i) = (\sum_m i \cdot b_m \theta^m + \log |E| - \log \theta) e^{(\sum_m i \cdot b_m \theta^m + \log |E|)} \tag{3}$$

$$\begin{aligned} H_{merge}(i) &= \sum_i (w_p + s_p)^2 \cdot e^{\sum_m i \cdot b_m \theta^m + \log |E|} \\ &= \sum_i (\theta/d_t + \theta/d_w)^2 \cdot e^{\sum_m i \cdot b_m \theta^m + \log |E|} \end{aligned} \tag{4}$$

where d_t/d_w is a constant which represents the average degree of task/worker in one node of the partitioning tree.

Substitute $H_{merge}(i)$ and $H_{partition}(i)$ in the expression $\sum_i (H_{partition}(i) + H_{merge}(i))$ with Equation 3 and Equation 4, we then use a numerical method to find the stationary point of θ. (Note that it is non-trivial to derive the closed form solution for the exponential sum expression.) Hence, given the size of a flow network, we could automatically determine the threshold value to achieve the best efficiency. For example, we derive $\theta = 32,000$ for $|E| =$ 3 million, which conforms to the empirical study shown in Figure 6(b).

5. EXPERIMENT

5.1 Experiment setup

Dataset. It is challenging to find real datasets to reflect the scheduling applications from the real-life systems such as Uber, Gigtalk and TaskRabbit because their data are not publicly available due to their commercial value. Therefore, we generated synthetic (SYN) dataset as well as adopted the Gowalla and Yelp check-in dataset for simulation by following the approaches of previous work [9, 15, 16]. In the following, we discuss our dataset in more details.

Synthetic dataset. For the synthetic dataset, we varied the number of tasks from $5K$ to $100K$. Initially the workers and tasks are located in a 500×500 grid, and the grid size increases when the number of tasks increases. The travel cost between two locations is proportional to their Euclidean distances. In our experiment, we generated both uniform (SYN-UNI) and skew (SYN-SKEW) distributions. For SYN-UNI, locations of spatial tasks and workers are chosen uniformly from the grids. For SYN-SKEW, 80% of workers and tasks are co-located in 6 random Gaussian clusters (with $\delta = 0.5$ and random centers) while locations of the remaining 20% workers and tasks are generated from a uniform distribution. Proceeding in this way, at the area with dense tasks, the workers are densely distributed.

Table 3: Statistics of the real datasets Gowalla and Yelp per time instance.

	Gowalla	Yelp
Avg. # of tasks $\|S\|$	12421	15320
Avg. # of workers $\|W\|$	1800	3200
Avg. # of W/T	42	64
Avg. # of q_w	15	22

Real dataset. Gowalla[1] was a location-based social network, where users are able to check in at different locations in their vicinity. For our experiments, we assumed that the Gowalla users are spatial workers, and checking in a spot is equivalent to completing a spatial task at that location. We picked the granularity of a time instance as one day. Consequently, we assumed that all the users who checked in during a day as our available workers for that day. Because users may have various check-ins during a day, for every user w, we set q_w as the number of check-ins of the user in that day, and we calculated g_w as the minimum bounding rectangle of those checked-in locations. Moreover, the check-in time was used as the deadline of one spatial task. The travel cost was calculated by the Euclidean distance divided by the average travel speed (i.e., 40 miles/hour).

The Yelp dataset[2] was captured in the greater Phoenix, Arizona and Las Vegas. For our experiments, we assumed that the businesses are the spatial tasks, Yelp users are the workers, and reviewing a business is equivalent to accepting a spatial task at its location. For each worker, we set the spatial region g_w as the minimum bounding rectangle of the locations of his review list per day. Moreover, we set the capacity of one worker as the size of his review list. We define the location of one spatial task as the location of the reviewed restaurant and randomly generated the deadline of one task at the range of 8am to 9pm. The travel cost was calculated the same way as the Gowalla data.We select 30 instances from one month review data at October 2012. The statistics of Gowalla and Yelp data are summarized in Table 3.

[1] snap.stanford.edu/data/loc-gowalla.html
[2] http://www.yelp.com/dataset_challenge/

Algorithms. To the best of our knowledge, we are the first to study the assignment and scheduling problem in spatial crowdsourcing. Therefore, we first evaluated the proposed GALS and LALS algorithms via different implementations. For GALS, in the assignment phase, we tested both the max-flow assignment and min-travel-cost max-flow assignment from [15]. We found that they achieved similar results, thus we only report the results of GALS using max-flow assignment. For both the NaïveLALS (NLALS) and the Bisection-LALS (BLALS) frameworks, we adapted two competitive partitioning techniques (i.e., **L**ocation-based and **K**Means-based) and compared them with our proposed **T**ask-oriented partitioning. In particular, Location-based partitioning [1] divides the entire region into equal-sized grid area, without considering the workload of each partition. With KMeans based partitioning [8], tasks are first divide into two even subsets via KMeans clustering, and for each subset of tasks it adds the workers that could complete these tasks into the corresponding partition. To save space we only show the algorithms with better performance under each framework, i.e., we compare NLALS-L (Location-based) with NLALS-T, and BLALS-K (KMeans-based) with BLALS-T. In terms of the threshold value, we determine the best threshold value via the proposed cost model. Note that the value is independent of the partitioning techniques. We compared our algorithms with the baseline algorithm (A&S) which uses the global matching approach for task-matching and insertion heuristic for task-scheduling, and ignores the third assignment update phase in GALS. We use insertion heuristic through our experiments for fair comparison.

In summary, we compared BLALS-T and NLALS-T with BLALS-K, NLALS-L, GALS and A&S, to demonstrate the advantage of Task-oriented partitioning. In addition, we compared BLALS-T with NLALS-T, GALS and A&S to evaluate the performance of Bisection-based LALS framework.

Table 4: Experiment parameters

Parameters	Value range
number of task $\|S\|$	$5K, 10K, \mathbf{25K}, 50K, 100K$
W/T	$40, \mathbf{80}, 120$
maximum region g_w	$3\%, 6\%, \mathbf{9\%}, 12\%, 15\%$
maximum capacity of q_w	$10, 15, \mathbf{20}, 25, 30$

Configuration and measures. We evaluated the scalability of the algorithms by varying the number of tasks from $5K$ to $100K$. In the default setting, the spatial workers were with the maximal capacity 20 and the spatial region sizes were at most $[9\%, 9\%]$ of the entire region, the tasks deadlines were randomly generated. We then varied the average number of workers whose spatial region contains a given task, namely workers per task(W/T). Intuitively, high value of W/T represents a worker-dense area. We also evaluated the effects of worker's region constraint g (from $[3\%, 3\%]$ to $[15\%, 15\%]$) and capacity constraint q (from 10 to 30). However, we do not report the results of varying region and capacity constraint because of the space limitation and the effects are similar of varying $|S|$ and W/T. To simulate the real application scenario, we tested the overall performance of our algorithms for a continuous 10 time instances: each instance represents a small time interval (e.g, one hour or one day). At each instance, the server receives a set of new tasks and workers, but also considers the available workers [3] and tasks (unscheduled and unexpired) from the previous instances. Because the server did not have the global knowledge in this online setting, the server only optimized the task scheduling at each time instance. The default settings (in bold font) of the experiment parameter are listed in Table 4.

[3]If one worker has non-empty scheduling but still has capacities, we also keep the worker for the next time instance.

For each of these experiments, the results were reported as averaged over 50 runs. We recorded the total number of scheduled tasks, the response time and the average travel cost per task. All algorithms were implemented in Java and all experiments were conducted on the Linux OS with Intel Core i5-2400 CPU @ 3.10G HZ and 16 GB memory.

5.2 Scalability with size of data set

In this set of experiments, we evaluated the scalability of our approaches by varying the number of tasks from $5K$ to $100K$. Table 5 shows the number of scheduled tasks on SYN-UNI. GALS performs best and improves the baseline algorithm A&S by up to 30%, and the gain increases as the number of tasks increases. This indicates that both the global assignment and the assignment update phases in GALS are the crucial steps to improve the final scheduling quality. In addition, NLALS-T outperforms NLALS-L, and is comparable with GALS in terms of quality, which demonstrates the advantage of our Task-oriented partitioning technique. We also notice BLALS-T performs better than BLALS-K. The reason is that BLALS-K allows overlapping workers between different partitions, the resolution of conflicts would reduce the advantage of global assignment. Finally, BLALS-T achieves similar results as NLALS-T, which indicates that our bisection framework does not lose much accuracy under our Task-oriented partitioning technique, and the bottom up merge process preserves the spatial properties. Therefore, BLALS-T is able to achieve near-optimal results as GALS. For example, for the 50 thousand dataset, compared with GALS, the number of completed tasks by BLALS-T is reduced from 49000 to 47000 (only 4%).

Table 5: No. of scheduled tasks on SYN-UNI when varying $|S|$

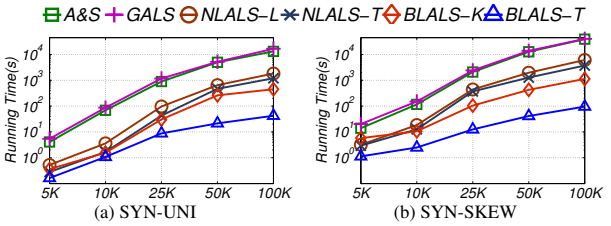
Data size	A&S	GALS	NLALS-L	NLALS-T	BLALS-K	BLALS-T
$5K$	3501	**4656**	4525	4642	4458	4561
$10K$	7072	**9708**	9602	9684	9594	9653
$25K$	17743	**24725**	23588	24699	23539	24576
$50K$	36365	**49050**	46706	48833	46006	47404
$100K$	55966	**76871**	70855	72414	69408	72003

Table 6: No. of scheduled tasks on SYN-SKEW when varying $|S|$

Data size	A&S	GALS	NLALS-L	NLALS-T	BLALS-K	BLALS-T
$5K$	3379	**3986**	3804	3911	3829	3896
$10K$	7075	**8263**	7908	8201	7706	8093
$25K$	19049	**21849**	20010	21747	19682	21473
$50K$	35614	**43653**	40333	43368	39286	42095
$100K$	56511	**68505**	60595	66275	59172	63937

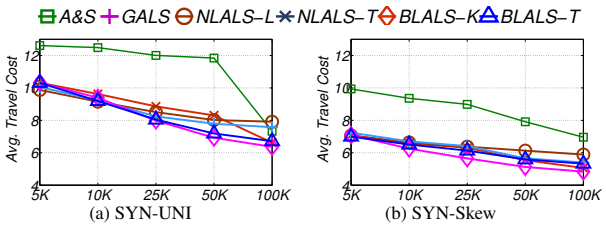
Table 6 depicts the performance on SYN-SKEW. The results are similar with those on SYN-UNI: GALS performs best and BLALS-T is able to achieve close results as GALS. On average, the number of scheduled tasks on the skewed data is lower than that on the uniform data. The reason is that in the skewed dataset, a large number of tasks are covered by zero or fewer number of workers; while a small number of tasks are covered by a concentrated clusters of workers. On the other hand, the performance gain of GALS on SYN-SKEW over A&S is lower than on SYN-UNI, which indicates that the assignment phase plays a more important role on SYN-SKEW than on SYN-UNI. We notice that on SYN-SKEW the advantage of Task-oriented partitioning over Location-based partitioning is larger than on SYN-UNI. This is because on SYN-UNI, workers and tasks are uniformly distributed, Location-based partitioning might yield as balanced partitions as Task-oriented partitioning. In contrast, on SYN-SKEW, Task-oriented partitioning achieves much better results than Location-based partitioning because it generates more balanced partitions via utilizing task connectivity information.

Figures 7(a) and (b) show the running time of different approaches on SYN-UNI and SYN-SKEW respectively. As expected, the running time increases when the number of tasks increases. A&S and



(a) SYN-UNI (b) SYN-SKEW

Figure 7: Running time of varying number of tasks $|S|$



(a) SYN-UNI (b) SYN-Skew

Figure 8: Avg. travel cost of varying number of tasks $|S|$

GALS are the least efficient approach due to its high computational cost in the global assignment phase. GALS performs little worse than A&S because it requires an additional assignment update phase. Even for the 25K dataset, both GALS and A&S take more than 1000 seconds, which indicates that both of them are not applicable for large-scale data in real applications. With the Naïve-LALS framework, the running time of NLALS-L and NLALS-T marginally improves over GALS. This is because the straggler and residual problems exist in the conventienal LALS framework. Finally, we notice that BLALS-T is the most efficient method, which is one order of magnitude faster than BLALS-K, and several orders of magnitude faster than another approaches. The reasons are twofolds: first, the bisection framework successfully addresses the problems incurred in the LALS framework; second, unlike K-means based partitioning, our Task-oriented partitioning does not allow duplicate workers to exist in different partitions, and generates more balanced partitions by considering the connectivity information.

Table 7: Experiment results on Gowalla

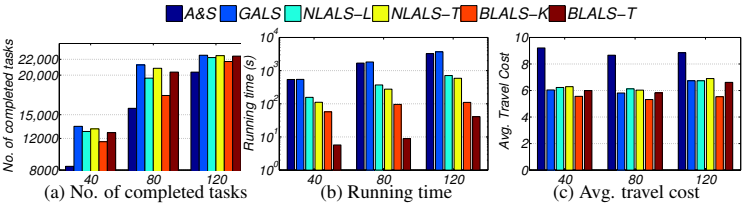
	No. of Tasks	Running time(s)	Travel cost/task (mile)
A&S	8574	136.80	8.30
GALS	**10031**	147.02	5.87
NLALS-L	9219	28.47	6.02
NLALS-T	9867	15.49	**5.80**
BLALS-K	9094	24.11	5.86
BLALS-T	9547	**3.80**	5.86

Table 8: Experiment results on YELP

	No. of Tasks	Running time(s)	Travel cost/task (mile)
A&S	10785	432.07	8.98
GALS	**13125**	739.80	6.10
NLALS-L	12654	130.74	6.20
NLALS-T	13074	76.30	6.37
BLALS-K	12501	39.09	**5.64**
BLALS-T	12898	**5.923**	5.88

In conclusion, BLALS-T well addresses all the efficiency bottlenecks for task assignment and scheduling in spatial crowdsourcing and is scalable for real-time applications.

Figures 8 (a) and (b) show the average travel cost per task of different approaches on SYN-UNI and SYN-SKEW respectively. The results for our proposed approaches are close to each other. We observe that when the number of tasks increases, the average travel cost decreases. A possible reason is that when the number of tasks increases, the nearby tasks of each worker are increasing accordingly. Also note A&S has the highest travel cost on SYN-SKEW, which indicates that the assignment update module in GALS is important for high scheduling quality. Table 7 and 8 depict the experiment results on the Gowalla and YELP dataset. The results are similar to those on synthetic data. GALS algorithm achieves the best scheduling quality but the worst efficiency. NLALS-T (NLALS-L) improves the efficiency of GALS marginally but still does not scale to large number of workers and tasks. Finally, BLALS-T achieves near-optimum results as GALS and is more efficient than all the other alternative approaches.



(a) No. of completed tasks (b) Running time (c) Avg. travel cost

Figure 9: Experiment results of varying W/T on SYN-UNI

5.3 Effect of W/T

In this set of experiments, we evaluated our approaches by varying the average number of candidate workers per task, i.e., W/T. We only report the experiment results on SYN-SKEW here due to the space limitation. Figure 9(a) shows the number of completed tasks by different approaches. Clearly, the number of completed tasks increases as W/T grows. This is because larger value of W/T means more candidate workers per task. When the number of workers per task is 120, the number of completed tasks by different approaches are close to each other, which means that most tasks have been completed. In addition, Figure 9 indicates that GALS $\gg$ {NLALS-T, BLALS-T} $\gg$ {NLALS-L, BLALS-L} $\gg$ A&S, where $A \gg B$ denotes on average A performs better than B in terms of number of completed tasks. Figure 9(b) and (c) show the running time and the average travel cost respectively. BLALS-T is consistently more efficient than all the other approaches. The travel cost of our approaches is similar to each other while A&S obtains a much higher average travel cost.

Table 9: Total No. of completed tasks of batch processing

Data size	AS	GALS	NLALS-L	NLALS-T	BLALS-K	BLALS-T
$10K$-UNI	83698	98266	98100	**104920**	97848	98136
$10K$-SKEW	69590	83148	78888	**95132**	78454	81890
Gowalla	81030	96674	90874	**102232**	85536	91692
Yelp	108300	127492	111752	**134656**	110540	125538

Table 10: Total running time (seconds) of batch processing

Data size	AS	GALS	NLALS-L	NLALS-T	BLALS-K	BLALS-T
$10K$-UNI	861.64	1049.18	140.590	105.74	85.26	**26.74**
$10K$-SKEW	1175.78	1254.29	216.88	99.86	77.268	**24.34**
Gowalla	1551.91	1636.86	227.38	160.26	102.52	**24.55**
Yelp	3961.76	4176.65	911.12	516.2	189.63	**41.59**

Table 11: Avg. Travel cost of batch processing

Data size	AS	GALS	NLALS-L	NLALS-T	BLALS-K	BLALS-T
$10K$-UNI	12.32	9.54	**8.84**	9.14	9.18	9.22
$10K$-SKEW	9.11	6.29	6.632	6.21	**5.77**	6.05
Gowalla	8.85	5.76	6.16	6.26	**5.57**	5.84
Yelp	8.47	6.06	6.20	6.30	**5.37**	5.98

5.4 Dynamic scenario

In the last set of experiments, we evaluated our approaches in the continuous 10 instances on SYN-UNI, SYN-SKEW and real dataset, where at each instance we have new workers and tasks. The number of tasks at each instance for SYN-UNI and SYN-SKEW is $10K$. Tables 9, 10 and 11 report the experimental results of number of completed tasks, total running time and average travel cost respectively. We find that NLALS-T performs better than NLALS-L and A&S in terms of the number of completed tasks. In addition, BLALS-T achieves the lowest computational cost and similar number of completed tasks as NLALS-T, which demonstrates that BLALS-T is practical for real applications.

6. RELATED WORK

Crowdsourcing has recently attracted a lot of attentions from researchers in different application domains such as database system [11,25], team formulation [17] and crowd recommendation [21].

Recently spatial crowdsourcing [1, 2, 8, 12, 15] is emerging, and the interests of spatial croudsourcing research focus on the task assignment [1, 5, 8, 15, 22], trust [16] and privacy issues [24]. For example, Kazemi and Shahabi [15] formulated task assignment in spatial crowdsourcing as a matching problem with the primary objective of maximizing the number of matched tasks, and Alfarrarjeh et al. [1] scaled up the assignment algorithm in a distributed setting. Reliable task assignment addressing trust issues in spatial crowdsourcing have been studied in [8, 16]. In [8] Cheng et.al. proposed a partitioning heuristic which recursively divides tasks and workers via KMeans clustering to improve the assignment efficiency. Compared with [1, 8], our bisection-based partitioning technique generates disjoint worker and task sets per partition, thus we do not need to address the issue of conflicts assignment. In addition, our partitioning technique utilizes both the spatial and the connectivity information between workers and tasks to achieve a balanced workload, thus successfully avoids the straggler problem for the partitioning techniques. We also develop an analytical method to automatically determine the size of one partition. Therefore, when adapting their partitioning approaches to our problem setting, our proposed algorithm is at least one order of magnitude faster than their approaches (see more details in Section 5). On the other hand, Deng et. al. [9] studied the scheduling problem from the worker's perspective and developed both exact and heuristic algorithms to help the worker to formulate the best scheduling. However, their paper assumes that each worker has been pre-assigned with some tasks, and thus only deals with single worker case. In this paper, we are able to assign tasks and suggest scheduling for multiple workers simultaneously in a unified framework.

Our work is also related to some combinatorial optimization problem such as Vehicle Routing Problem(VRP) [6, 20]. 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, each worker has a spatial constraint and only accepts the tasks in her/his nearby region; the spatial tasks are also not guaranteed to be completed by the workers. Finally, in a spatial crowdsourcing platform, we need to provide efficient solutions for potentially millions of tasks and workers. This is different from the solutions in VRP which take hundreds of seconds even for one hundred delivery points. There also exists partitioning approaches [7] for Dynamic VRP where the typical solution is to divide the customers into different partitions, then assign one vehicle to each partition and run a TSP routing strategy separately. However, in our problem setting many workers reside in one partition, and TSP cannot be used to schedule for multiple workers.

7. CONCLUSION

In this paper, we proposed a unified framework for spatial crowdsourcing platform, in which for each worker, we not only assign tasks, but also provide an effective scheduling with minimum travel cost. Our first algorithm, GALS, which iteratively improves the assignment and scheduling, performs the best in the number of completed tasks. However, due to a computational bottleneck, it is too slow to be used in online spatial crowdsourcing systems. Hence, we devised a Bisection-based LALS framework which performs a top-down recursive bisection and bottom-up merge procedure iteratively. The recursive bisection is performed with Task-oriented partitioning that maintains both the spatial properties and the connectivity of tasks. The experiment results verified that our bisection-based LALS algorithm is orders of magnitude faster than GALS while only sacrificing little quality, and thus is suitable for real world systems.

There are several promising directions for future work. First, we plan to develop algorithms to support trust-able task assignment and scheduling in which multiple workers are required to complete a single task. Second, we intent to extend the current framework to a dynamic scenario where both workers and tasks vary over time.

Acknowledgments. Dingxiong Deng and Cyrus Shahabi were supported by NSF grants IIS-1320149 and CNS-1461963, the USC Integrated Media Systems Center (IMSC), and unrestricted cash gifts from Google, Northrop Grumman, Microsoft, and Oracle. Linhong Zhu was supported by DARPA grant No. W911NF-12-1-0034. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of any of the sponsors such as the National Science Foundation.

8. REFERENCES

[1] A. Alfarrarjeh, T. Emrich, and C. Shahabi. Scalable spatial crowdsourcing: A study of distributed algorithms. To appear in MDM '15, 2014.

[2] F. Alt, A. S. Shirazi, A. Schmidt, U. Kramer, and Z. Nawaz. Location-based crowdsourcing: Extending crowdsourcing to the real world. In *CHI*. ACM, 2010.

[3] S. Arora, S. Rao, and U. Vazirani. Expander flows, geometric embeddings and graph partitioning. *Journal of ACM*, 56:5:1–5:37, April 2009.

[4] F. Bourse, M. Lelarge, and M. Vojnovic. Balanced graph edge partition. KDD '14, pages 1456–1465, New York, NY, USA, 2014. ACM.

[5] I. Boutsis and V. Kalogeraki. On task assignment for real-time reliable crowdsourcing. In *ICDCS*, pages 1–10, 2014.

[6] O. Bräysy and M. Gendreau. Vehicle routing problem with time windows, part i: Route construction and local search algorithms. *Transportation Science*, 39(1):104–118, Feb. 2005.

[7] F. Bullo, E. Frazzoli, M. Pavone, K. Savla, and S. L. Smith. Dynamic vehicle routing for robotic systems. *Proceedings of the IEEE*, 99(9):1482–1504, 2011.

[8] P. Cheng, X. Lian, Z. Chen, R. Fu, L. Chen, J. Han, and J. Zhao. Reliable diversity-based spatial crowdsourcing by moving workers. *Proc. VLDB Endow.*, 8(10):1022–1033, June 2015.

[9] D. Deng, C. Shahabi, and U. Demiryurek. Maximizing the number of worker's self-selected tasks in spatial crowdsourcing. In *GIS*. ACM, 2013.

[10] F avor. `https://favordelivery.com/`.

[11] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. SIGMOD '11, 2011.

[12] R. K. Ganti, F. Ye, and H. Lei. Mobile crowdsensing: current state and future challenges. *IEEE Communications Magazine*, 49(11):32–39, 2011.

[13] Gigwalk. `http://gigwalk.com/`.

[14] A. Goldberg, S. Hed, H. Kaplan, R. Tarjan, and R. Werneck. Maximum flows by incremental breadth-first search. In *Algorithms-ESA 2011*. Springer Berlin Heidelberg, 2011.

[15] L. Kazemi and C. Shahabi. Geocrowd: Enabling query answering with spatial crowdsourcing. In *GIS*, pages 189–198. ACM, 2012.

[16] L. Kazemi, C. Shahabi, and L. Chen. Geotrucrowd: Trustworthy query answering with spatial crowdsourcing. In *GIS*, pages 314–323. ACM, 2013.

[17] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. KDD '09, pages 467–476, 2009.

[18] J. Lehmann, C. Castillo, M. Lalmas, and E. Zuckerman. Finding news curators in twitter. In *WWW*, Republic and Canton of Geneva, Switzerland, 2013.

[19] J. Lehmann, C. Castillo, M. Lalmas, and E. Zuckerman. Transient news crowds in social media. In *In ICWSM*, 2013.

[20] Y. Li, D. Deng, U. Demiryurek, C. Shahabi, and S. Ravada. Towards fast and accurate solutions to vehicle routing in a large-scale and dynamic environment. In *SSTD*, volume 9239, pages 119–136. 2015.

[21] Y. Liu, T. Alexandrova, and T. Nakajima. Using stranger as sensors: temporal and geo-sensitive question answering via social media. In *WWW*, 2013.

[22] L. Pournajaf, L. Xiong, and V. Sunderam. Dynamic data driven crowd sensing task assignment. *Procedia Computer Science*, 29(0):1314–1323, 2014.

[23] TaskRabbit. `https://www.taskrabbit.com/`.

[24] H. To, G. Ghinita, and C. Shahabi. A framework for protecting worker location privacy in spatial crowdsourcing. *Proceedings of VLDB*, 7(10):919–930, 2014.

[25] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. SIGMOD '13, NY, USA, 2013. ACM.

[26] L. Zhu, W. K. Ng, and J. Cheng. Structure and attribute index for approximate graph matching in large graphs. *Information Systems*, 36(6):958–972, 2011.