Turbo-GTS: A Fast Framework of Optimizing Task Throughput for Large-Scale Mobile Crowdsourcing

WEI LI, Auburn University
HAIQUAN CHEN, California State University, Sacramento
WEI-SHINN KU and XIAO QIN, Auburn University

In mobile crowdsourcing, workers are financially motivated to perform as many self-selected tasks as possible to maximize their revenue. Unfortunately, the existing task scheduling approaches in mobile crowdsourcing fail to consider task execution duration and do not scale for massive tasks and large geographic areas. In this article, we propose a novel framework, Turbo-GTS, in support of large-scale geo-task scheduling, with the objective of identifying an optimal task assignment for each worker to maximize the total number of tasks that can be completed for an entire worker group, given the geographic locations of each task and each worker. Since the exact solution to the geo-task scheduling problem is computationally intractable, we first propose two sub-optimal approaches (least cost neighbor with particle filtering and non-urgency degree particle filtering with iterative clustering) based on particle filtering and DBSCAN for the single-worker geo-task scheduling problem. We then extend our work to solve the multi-worker geo-task scheduling problem by proposing two space partitioning-based methods (QT-NNH and QT-NUD), which leverage point-region quadtree to ensure workload balancing. The effectiveness and efficiency of the four proposed approximate solutions are verified by our extensive experiments using both real and synthetic data. Compared to state-of-the-art approaches, our proposed solutions are able to return a higher number of completed tasks for the worker group while reducing the computation cost by up to three orders of magnitude when coping with massive tasks distributed in large geographic areas.

CCS Concepts: • **Information systems → Location based services**;

Additional Key Words and Phrases: Mobile crowdsourcing, particle filtering, quadtree

ACM Reference format:
Wei Li, Haiquan Chen, Wei-Shinn Ku, and Xiao Qin. 2020. Turbo-GTS: A Fast Framework of Optimizing Task Throughput for Large-Scale Mobile Crowdsourcing. *ACM Trans. Spatial Algorithms Syst.* 6, 1, Article 1 (January 2020), 29 pages.
https://doi.org/10.1145/3363450

1 INTRODUCTION

The ubiquity of mobile platforms and smartphones breeds a large number of mobile crowdsourcing applications like TaskRabbit (a mobile marketplace allowing users to outsource small tasks to

This research was funded in part by National Science Foundation grants IIS-1618669 (III) and ACI-1642133 (CICI).
Authors' addresses: W. Li, W.-S. Ku, and X. Qin, Department of Computer Science and Software Engineering, Auburn University, 1301 Shelby Center, Auburn, AL 36849; emails: {wei.li, weishinn, xqin}@auburn.edu; H. Chen, Department of Computer Science, California State University, Sacramento, Riverside Hall 5018, MS 6021, 6000 J Street, Sacramento, CA 95819-6021; email: haiquan.chen@csus.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
2374-0353/2020/01-ART1 $15.00
https://doi.org/10.1145/3363450

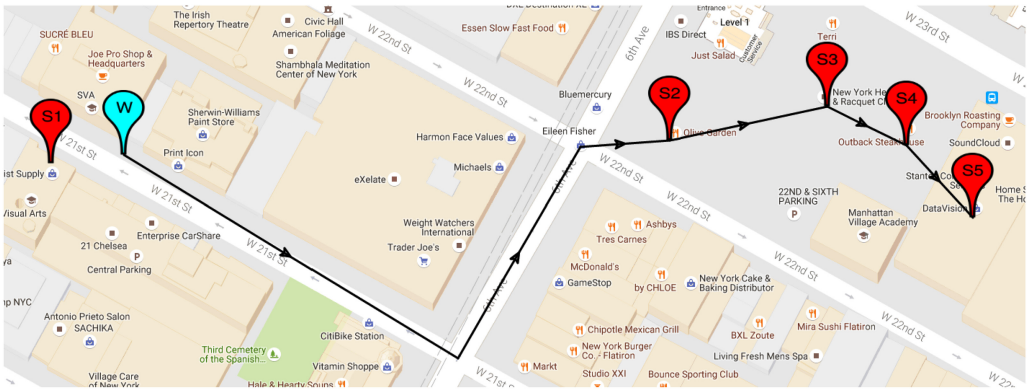


Fig. 1. An example of spatial task scheduling in Manhattan Island, New York City. w represents the current location of a worker; s_1 represents a task near w but with longer execution duration, whereas s_2, s_3, s_4, and s_5 represent four other tasks located farther from w but with shorter execution duration. The arrow represents the optimal task execution route for w.

workers in their neighborhood [5]), Uber (a mobile application allowing passengers with smartphones to submit trip requests that are then routed to willing drivers [20]), Gigwalk (a crowdsourcing service that helps businesses appraise their performance [18]), and MediaQ (an online media management framework allowing workers to collect, organize, share, search, and trade user-generated mobile images and videos [24]). A common feature of all of these applications is that workers are required to go to the exact spot of each task in person and perform the task by means of a mobile device.

In mobile crowdsourcing, obtaining an optimal or near-optimal task schedule for a worker or a group of workers to accomplish as many tasks as possible is a crucial yet quite challenging problem. Take the scenario shown in Figure 1 as an example. A worker at location w has five tasks to complete, s_1, s_2, s_3, s_4 and s_5, each with a preset expiration time. According to the existing scheduling solutions [10, 23], s_1 will be selected as the first option to execute because it is geographically closest to w. Considering that s_1 needs 1 hour to finish while all of the other four tasks only need 1 minute to complete, it is very likely that s_2, s_3, s_4, and s_5 will all have expired after finishing s_1, leading to the fact that the worker can only accomplish one task. Therefore, the optimal solution in the case should be the task sequence $s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5$ so that the worker can have a higher probability of getting more tasks completed.

Limitations of existing approaches. We make the following troubling observations on the existing research for mobile crowdsourcing:

- *Lack of capability to scale up to identify optimal/sub-optimal task assignment for mobile crowdsourcing with massive tasks in large geographic areas.* As shown in our experiments (to be detailed in Sections 6 and 7), existing scheduling solutions [10, 11, 23] fail to find optimal/sub-optimal task assignment when coping with massive tasks distributed in large geographic areas. Tables 1 and 2 show the performance of BSH [10] (single-worker task scheduling) and BLALS-T [11] (multi-worker task scheduling), using tasks randomly chosen from the Foursquare mobile user check-ins in New York City [36–38]. As depicted in Tables 1 and 2, novel task scheduling solutions that are able to accomplish more tasks given massive tasks distributed in large geographic areas are particularly desirable in practice.
- *Absence of efficient task scheduling frameworks to support real-time mobile crowdsourcing applications.* Existing scheduling solutions [10, 11, 23] suffer from inefficiency in terms of

Table 1. Performance of the State-of-the-Art Single-Worker Task Scheduling Solution (BSH [10]) in Mobile Crowdsourcing

Total Number of Tasks	3,000	4,000	5,000	6,000	7,000
CPU Cost (hours)	0.8	1.3	5.5	7	7.5
Number of Accomplished Tasks	84	94	104	108	113

Table 2. Performance of the State-of-the-Art Multi-Worker Task Scheduling Solution (BLALS-T [11]) in Mobile Crowdsourcing on 300 Workers

Total Number of Tasks	10,000	15,000	20,000	25,000	30,000
CPU Cost (minutes)	<1	2	4	8	11
Number of Accomplished Tasks	7,483	8,879	9,861	10,926	12,048

generating the task assignment sequence for each worker when handling massive tasks distributed in large geographic areas. As depicted in Tables 1 and 2, the computation cost of BSH [10] and BLALS-T [11] increased exponentially when the total number of tasks was increased. For example, BSH [10] cost 7.5 hours to schedule 113 tasks for one worker, whereas BLALS-T [11] needed 11 minutes to assign and schedule 12,048 tasks among 300 workers.

- *Failure to take into account important spatial and temporal constraints.* Existing scheduling solutions [10, 11, 23] ignore the task execution duration and treat the task expiration time as the only temporal factor in task scheduling. However, in reality, it always requires a certain amount of time for a worker to complete a task. For example, in Gigwalk, tasks may require a few minutes or a few hours [15]. For example, a hotel gig may take a Gigwalker 1 to 2 hours [13]. A preferred scheduling solution should be able to consider important spatiotemporal characteristics of real-world mobile crowdsourcing applications.

Our goal and contributions. In this article, we propose a novel framework, Turbo-GTS, in support of large-scale geo-task scheduling (GTS), with the objective of identifying an optimal task assignment for each worker to maximize the total number of tasks that can be completed for an entire worker group, given the geographic locations of each task and each worker. Turbo-GTS is able to perform scheduling by taking into account various spatial and temporal constraints, including task execution duration, task expiration time, and task geographic locations. The contributions of this work are summarized as follows:

- We formulate the GTS problem that maximizes the number of completed tasks for a worker group by taking into account important spatial and temporal constraints, including task execution duration and task expiration time and propose a novel framework, Turbo-GTS, to solve the GTS problem with massive tasks distributed in large geographic areas.
- We define the non-urgency degree (NUD) for task assignment and present two approximate solutions, least cost neighbor with particle filtering (LCPF) and non-urgency degree particle filtering with iterative clustering (NUD-IC), for the single-worker geo-task scheduling (SGTS) problem. LCPF solves the problem by incorporating particle filtering, whereas NUD-IC generates the sub-optimal schedule by integrating particle filtering with DBSCAN clustering based on the NUD.
- Inspired by the proposed LCPF and NUD-IC for the SGTS problem, we solve the multi-worker geo-task scheduling (MGTS) problem by proposing two space partitioning–based

task assignment methods, QT-NNH and QT-NUD, which leverage point-region (PR) quadtree to ensure workload balancing among multiple workers.
- The effectiveness and efficiency of the four proposed approximate solutions were verified by our extensive experiments using the synthetic data and Foursquare [36–38] mobile user check-in data in three highly populated cities (i.e., New York City; Austin, Texas; Tokyo). Compared to the state-of-the-art approaches, all of our proposed solutions were able to return a higher number of completed tasks for a single worker or the entire worker group while reducing the computation cost by up to three orders of magnitude when coping with massive tasks distributed in large geographic areas.

The remainder of this article is organized as follows. Section 2 describes the related work on task scheduling in mobile crowdsourcing. Section 3 formulates our studied problem and presents the system architecture of Turbo-GTS. In Section 4, we elaborate on LCPF and NUD-IC algorithms for single-worker task scheduling. QT-NNH and QT-NUD are introduced to solve multi-worker task scheduling in Section 5. Experimental evaluation is presented in Sections 6 and 7. Section 8 concludes the article and presents our future work.

2 RELATED WORK

2.1 Mobile Crowdsourcing

Crowdsourcing enables businesses to utilize the spare time of a crowd of people to do jobs that would have been done by their employees, contractors, or outsourcing suppliers in the past [19]. For example, Amazon's Mechanical Turk is a micro-task market that helps individuals and businesses engage a large group of people to do things for monetary payment [25]. Although currently crowdsourcing is mainly used to perform small and easy jobs, models in support of more sophisticated tasks such as searching queries, decision making, and scientific problem solving have already been investigated [2, 4, 26, 30]. The commercial prospects of crowdsourcing have also been discussed in Vukovic [33] and Whitla [35]. Beyond the benefits from the perspective of enterprises, crowdsourcing even has the potential to be used for governments to deal with disasters [14, 16, 39].

Mobile crowdsourcing means to outsource a set of spatial tasks to a group of workers who are physically located at a certain place to perform a certain task at a certain time. There are two types of mobile crowdsourcing: reward-based and self-driven–based crowdsourcing. In reward-based mobile crowdsourcing, workers have reward benefits for correctly performed tasks. For example, EasyShift [28]—a smartphone application similar to Gigwalk—pays a user $3 to $8 for checking price and availability of several varieties of baby food products in a local retailer [3], where the user takes photos of the store's front entrance and the baby food shelves inside before answering several questions according to the actuality. Workers in self-driven–based crowdsourcing volunteer to perform tasks. In this case, workers are driven by incentives rather than being induced by any physical rewards; sample incentives include praise, recognition, job enrichment, and promotion opportunities. For example, a community data campaign was initiated during the spring of 2010 in the East Los Angeles neighborhood of Boyle Heights [1], where local residents with smartphones were recruited and answered survey questions regarding their neighborhood, home environment, transportation, school conditions, and work conditions. All volunteers participating in this campaign are self-motivated users.

2.2 Geo-Task Scheduling

Considering that traditional task scheduling fails to take spatial data into account, solutions from other works [7, 21, 22, 34] cannot be directly used in mobile crowdsourcing. We note that (1) location awareness, (2) workers' path selection, and (3) spatial indexing structures are three

most unique characteristics of mobile crowdsourcing that distinguishes itself from traditional task scheduling. First, mobile crowdsourcing requires workers to physically be at specific locations to complete the tasks. Every time a worker is assigned to a new task, she has to physically move to that task. As a result, the relative cost of completing any specific task to each worker is always changing dynamically, which traditional task scheduling models are unable to handle well. Second, since workers must travel to event places and perform the tasks, it is critical for the workers to select the best route and wisely schedule the task sequence. Traditional task scheduling or routing problems like the traveling salesman problem (TSP) are commonly used to formulate the problem of finding the optimal solution given fixed tasks. However, in mobile crowdsourcing, finding the right paths is much more challenging because (1) multiple workers start from different locations and (2) both the number of workers and tasks may vary over time due to the unavailability of the workers or the expiration of tasks. Last but not least, a desirable solution to mobile crowdsourcing relies on the efficient and effective design and integration of different spatial indexing mechanisms, such as R-tree, quadtree, or KD-tree, which is beyond the scope of the traditional task scheduling problems.

Task scheduling with spatial data has also been studied in the past. Kazemi and Shahabi [23] proposed three strategies to assign tasks from the crowdsourcing server's perspective (i.e., a server determines which task will be assigned to which worker). Their strategies were designed to address the maximum task assignment (MTA) problem by turning MTA into a maximum flow minimum cost problem. Dang and Cao [6] further improved the model of Kazemi and Shahabi to dedicatedly address the assignment problem of special tasks that can only be performed by workers with certain expertise in reward-based mobile crowdsourcing. The problem with their solutions is that the travel cost calculated based on their model can be very inaccurate, as the order chosen to perform tasks from a selected task set really matters in travel cost computation. Heuristic algorithms and exact algorithms based on dynamic programming and branch-and-bound strategies were proposed in Deng et al. [10]. However, their approaches suffer from scalability and inefficiency. In our prior work [27], we presented two approximate solutions, LCPF and NUD-IC, for the SGTS problem. The most closely related work to this article is Deng et al. [11], which attempted to solve the MGTS problem by maximizing the total number of tasks that can be accomplished by all workers. However, their work has two limitations. First, they assume that each worker has a maximum number of tasks that can be assigned to a worker. Second, they assume that each worker is only able to select tasks within a predetermined neighborhood. Therefore, their approach suffers from poor performance in practice when coping with massive tasks distributed in large geographic areas.

3 PROBLEM FORMULATION AND SYSTEM ARCHITECTURE

Definition 3.1. A spatial task s is a crowdsourced task located in a spatial region Z, with three innate attributes: its *locations* in Z (in this article, we use the same symbol s to represent both the location and the task itself for convenience, as the task itself is nothing but the three attributes in our model), its *execution duration* u_s and *expiration time* e_s.

In mobile crowdsourcing, each task will be located at a certain position in a spatial region and can only be accomplished there. Moreover, each task also has its own execution duration, which indicates how long it takes to complete the task, and expiration time, which indicates when the task is going to expire if it is still left unfinished. In this article, we use an imaginary task s_0 to represent the spot where w is located in Z originally. Similarly, t_0 represents the time when w starts from s_0, or the completion time of the imaginary task s_0.

Definition 3.2. A task set $S = \{s_1, s_2, \ldots, s_n\}$ is a collection of spatial tasks in Z waiting to be accomplished, in which each task can be accomplished once and only once.

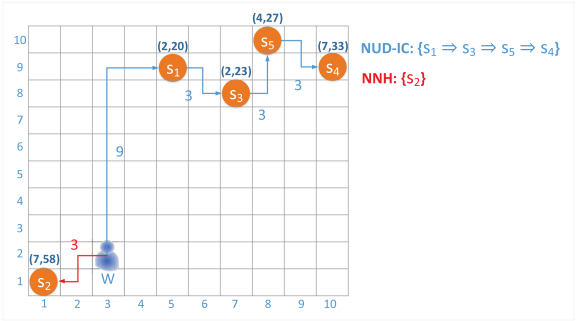


Fig. 2. Here, w represents the location where the worker starts at time 0. s_{1-5} represent all tasks at their respective locations. Parenthesized number pairs on top of each task represent this task's execution duration and expiration time with the same time unit as the scales in the grid, where distances are calculated in taxicab geometry.

In our model, we assume that no task can be executed by more than one worker (i.e., collaboration of workers is not taken into account in this article). We also assume that a worker will lose no time to head for the next one once the current task has been finished.

Definition 3.3. A task s_j is a *potential successor* of task s_i at time t_{s_i} if and only if it satisfies $t_{s_i} + c_{s_i,s_j} + u_{s_j} \leq e_{s_j}$, where c_{s_i,s_j} denotes the time required for traveling from s_i to s_j and t_{s_i} stands for the time when s_i has been accomplished.

Checking whether a task is another task's potential successor is very useful in pruning those tasks that have already expired or are about to expire so that the whole task set can be reduced in the future steps no matter what algorithm is being used. In addition, we assume that all tasks in S are the potential successors of s_0 at time t_0 when the worker is still in her original position. Otherwise, it makes no sense to include those unaccomplishable tasks from the very beginning into the candidate task set S.

Definition 3.4. A *task sequence* $R_{final} = \{s_{r_1}, s_{r_2}, \ldots, s_{r_m}\}$ $(1 \leq r_k \leq n, 1 \leq k \leq m, 1 \leq m \leq n)$ is a succession of tasks assigned to or chosen by a worker from the whole task set S such that all tasks in R_{final} are distinct from each other (if $i \neq j$, then $r_i \neq r_j, 1 \leq i, j \leq m$) and are able to be accomplished before their respective expiration time. The number of elements in R_{final} (m in this case) is defined as the *length of task sequence*.

Definition 3.5. Single-worker geo-task scheduling (SGTS) is designed to find the longest task sequence obtained from a task set S in a spatial region Z, given the time when the worker starts from her original location.

Figure 2 shows an example of the SGTS problem. In this figure, w indicates the worker's starting position at time 0. If NNH [10] is used, w will end up with only s_2 accomplished because the expiration time of all other tasks does not leave her enough time to perform any of them at the completion time of s_2; however, if w chooses NUD-IC (to be introduced in Section 4) as her scheduling method instead, w is going to end up with the longest task sequence $s_1 \rightarrow s_3 \rightarrow s_5 \rightarrow s_4$.

Definition 3.6. Multi-worker geo-task scheduling (MGTS) is designed to find a task scheduling plan for all workers from a worker set W so as to achieve the maximum number of accomplished tasks from a task set S in a spatial region Z, given the time when those workers start from their respective locations.

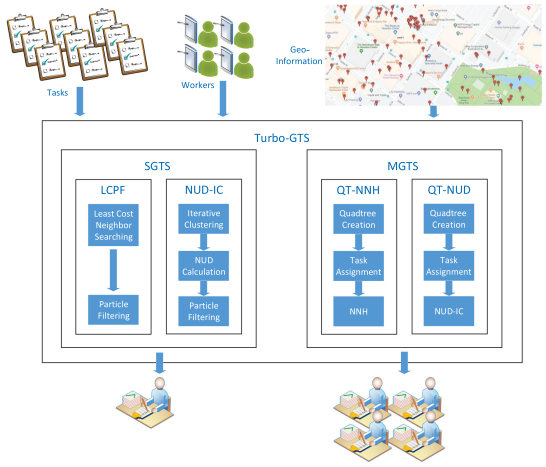


Fig. 3. System architecture of Turbo-GTS.

The MGTS problem is not simply a set of SGTS problems, and it cannot be solved by solving each worker's SGTS problem one by one, for workers compete for tasks and the solution to each SGTS problem conflicts with others. Thus, the MGTS problem has to be solved as a whole to achieve the maximum number of accomplished tasks by all workers.

Figure 3 shows the system architecture of Turbo-GTS. We proposed two algorithms, LCPF and NUD-IC (we will compare them in detail later in Section 4), for the SGTS problem and two algorithms, QT-NNH and QT-NUD (we will compare them in detail later in Section 5), for the MGTS problem. Turbo-GTS takes as input the workers' current locations and the related information (e.g., location, execution duration, and expiration time) of each task and returns the task assignment for each worker to maximize the total number of tasks that can be completed by all workers.

Table 3 summarizes the notations used in the problem formulation and the following algorithms.

4 SINGLE-WORKER GEO-TASK SCHEDULING

In this section, we introduce two approximate solutions, namely LCPF and NUD-IC, for the SGTS problem.

4.1 The Least Cost Neighbor with Particle Filtering

Particle filtering is a Monte Carlo method used to perform inference in a state-space model where the state of systems changes in every instant of time [29]. It was first proposed to deal with recursive Bayesian estimation [17] and then it was used as a solution to the general nonlinear filtering problems [8, 9]. Assume we want to estimate a Markov chain $\{X_n\}(n \geq 1)$, then we have

$$x_k = f_k(x_{k-1}, v_{k-1}). \tag{1}$$

In Equation (1), x_k is the state vector at instant k, f_k a possibly nonlinear function, and v_{k-1} an independent and identically distributed (i.i.d.) process noise sequence. However, what can be measured by us directly is just the corresponding observations $\{Z_n\}(n \geq 1)$, and we have

$$z_k = h_k(x_k, n_k). \tag{2}$$

In Equation (2), h_k is a nonlinear function and n_k a measurement noise sequence. Given the preceding settings, now we are able to estimate the probability density function (PDF) $p(x_k|z_{1:k})$ for the state vector x_k at instant k via particle filtering. First, a certain number ($ptcl_no$) of random

Table 3. Symbolic Notations

Symbol	Meaning
S	Whole task set
W	Worker set
w	Worker in W
Z	Spatial region where W and S are located
s	Task in S, representing its location in Z
e_s	Expiration time of s
u_s	Execution duration of s
s_0	Imaginary task representing where w starts in Z
t_0	Time when w starts from s_0
c_{s_i,s_j}	Time cost in traveling between s_i and s_j
t_s	Completion time of s
$ptcl_no$	Number of particles in particle filtering
Q	Priority queue of $ptcl_no$ task sequence samples
R_i	Task sequence sample in Q
G_i	Task group generated by classifying each task's duration
n_{div}	Number of G_i to divide S into
ddr	Given decimal used to calculate n_{div}
R_{final}	Task sequence returned by an algorithm

samples or particles are chosen from $\{X_n\}$ and each sample will be assigned an importance weight through importance sampling [32]. This procedure is called the *prediction stage*. After that, each sample's importance weight will be normalized and, in proportion to each sample's normalized weight, $ptcl_no$ new samples will be generated again. This procedure is called the *update stage*, which will be followed by a new prediction stage and so forth. For the resampling step, if given the set of particles paired with the corresponding importance weights $\{x_{k-1}^i, w_{k-1}^i\}_{i=1}^{ptcl_no}$ at instant $k-1$, the new sample set at the next instant k can be obtained by the following formulas:

$$x_k^i \sim p\left(x_k | x_{k-1}^i\right), \tag{3}$$

$$w_k^i \propto w_{k-1}^i p\left(z_k | x_k^i\right). \tag{4}$$

LCPF is a particle filtering algorithm based on the heuristic of nearest neighbor. For the sake of clarity and precision in describing our algorithm, we present the definition of *least cost neighbor* based on nearest neighbor as follows.

Definition 4.1. Given a task $s \in S$ at time t_s, let $S_{sub} = \{s_i | t_s + c_{s,s_i} + u_{s_i} \leq e_{s_i}, s_i \in S, s \neq s_i\}$; if there exists a task $s^* \in S_{sub}$ such that $c_{s,s^*} + u_{s^*} \leq c_{s,s_i} + u_{s_i}$ for any $s_i \in S_{sub}$, then s^* is the *least cost neighbor* of s in S at time t_s.

It is easy to see that searching for the least cost neighbors of a task s from the whole task set S is actually to seek the nearest neighbors (considering the execution duration of tasks into the calculation of nearness) of s from all of its potential successors in S.

LCPF is detailed in Algorithm 1. Before the particle filtering process begins, an empty priority queue Q for task sequence samples with the reciprocal of the completion time as each sample's priority is created. This means that the less the completion time of the task sequence, the larger the priority of this sample in the queue (line 1 in Algorithm 1), and the first sample with only one task s_0 (representing w's start location) is inserted into Q (line 2 in Algorithm 1). Lines 3 through 21 in

ALGORITHM 1: LCPF(S, s_0, t_0, $ptcl_no$)

1: Create an empty priority queue Q for at most $ptcl_no$ task sequence samples $\langle R_1, R_2 \ldots, R_{ptcl_no}\rangle$, where R_i with smaller completion time has a higher priority.
2: $Q.insertWithPriority(\{s_0\}, t_0^{-1})$ ▷ Reciprocal of completion time as priority.
3: **while** $Q.isNotEmpty()$ **do**
4: $R_{final} \leftarrow Q.getHighestPriority()$ ▷ Element with highest priority is returned but not removed.
5: Create an empty priority queue Q^*.
6: Copy all elements of Q into Q^*.
7: $Q.removeAllElements()$
8: **while** $Q^*.isNotEmpty()$ **do**
9: $R \leftarrow Q^*.pullHighestPriority()$ ▷ Element with highest priority is returned and removed.
10: $s_{end} \leftarrow$ the last task in R
11: $t_{end} \leftarrow$ the completion time of R
12: $S_{nb} \leftarrow$ at most $ptcl_no$ least cost neighbors of s_{end} at t_{end} from S.
13: **for each** $s_j \in S_{nb}$ **do**
14: $R_j^* \leftarrow R \cup \{s_j\}$
15: $Q.insertWithPriority(R_j^*, t_{s_j}^{-1})$ ▷ Reciprocal of completion time as priority.
16: **if** $|Q| > ptcl_no$ **then**
17: $Q.removeLowestPriority()$ ▷ Element with lowest priority is removed.
18: **end if**
19: **end for**
20: **end while**
21: **end while**
22: **return** R_{final}

Algorithm 1 show the whole process of particle filtering. Initially, the task sequence with the least completion time is kept into R_{final} as a candidate of the final result (line 4 in Algorithm 1), and Q^* is created as a duplicate of Q (lines 5 and 6 in Algorithm 1) before clearing out all elements in Q for the upcoming new samples (line 7 in Algorithm 1). Lines 8 through 20 in Algorithm 1 show the process of re-sampling. Each task sequence sample finds at most $pctl_no$ least cost neighbors of the last task in the sequence (lines 9 through 12 in Algorithm 1), generates at most $pctl_no$ new samples by appending each least cost neighbor to the sequence (line 14 in Algorithm 1), and inserts every newly generated sample into Q (line 15 in Algorithm 1). Lines 16 through 18 in Algorithm 1 make sure only the top $ptcl_no$ task sequences with the least completion time are kept in Q. The process of re-sampling continues until no more new samples can be found in Q (line 3 in Algorithm 1) and the current value kept in R_{final} is returned as the final result (line 22 in Algorithm 1).

4.2 Non-Urgency Degree Particle Filtering with Iterative Clustering

In this section, we elaborate on the NUD-IC algorithm, which generates the sub-optimal schedule based on particle filtering, DBSCAN clustering, and the concept of NUD to be introduced here. The basic idea behind the NUD-IC algorithm is that if the completion of a task can leave more time for the accomplishment of other tasks before they expire, this task should be selected as the worker's next stop with a higher probability.

Given a task s and its completion time t_s, the NUD of s for each task $s_i \in S$ at time t_s can be defined as follows:

$$NUD(s_i, s, t_s) = \frac{e_{s_i} - u_{s_i} - c_{s,s_i} - t_s}{u_{s_i}}, \tag{5}$$

where the numerator means how early s_i could be finished before its expiration time when w starts from s at time t_s. Given a task sequence sample R_i, its importance weight can be in proportion to

ALGORITHM 2: *NUD_SUM*(S, R)

```
sum ← 0
s_end ← the last task in R
t_end ← the completion time of R
for each s_i ∈ S \ R do
    if s_i is a potential successor of s_end at t_end then
        sum ← sum + NUD(s_i, s_end, t_end)        ▷ Definition of NUD is shown in Equation (5).
    end if
end for
return sum
```

ALGORITHM 3: *NUDPF*$(S, s_0, t_0, ptcl_no)$

```
Create an empty priority queue Q for at most ptcl_no task sequence samples ⟨R_1, R_2 ..., R_ptcl_no⟩, where
R_i with larger NUD_SUM(S, R_i) value has a higher priority.
Q.insertWithPriority({s_0}, NUD_SUM(S, {s_0}))                    ▷ NUD_SUM value as priority
while Q.isNotEmpty() do
    R_final ← Q.getHighestPriority()    ▷ Element with highest priority is returned but not removed.
    Create an empty priority queue Q*.
    Copy all elements of Q into Q*.
    Q.removeAllElements()
    while Q*.isNotEmpty() do
        R ← Q*.pullHighestPriority()    ▷ Element with highest priority is returned and removed.
        s_end ← the end task of R
        t_end ← the end time of R
        for each s_j ∈ S \ R do
            if s_j is a potential successor of s_end at t_end then
                R*_j ← R ∪ {s_j}
                Q.insertWithPriority(R*_j, NUD_SUM(S, R*_j))          ▷ NUD_SUM value as priority
                if |Q| > ptcl_no then
                    Q.removeLowestPriority()                  ▷ Remove element with lowest priority.
                end if
            end if
        end for
    end while
end while
return R_final
```

the value of the function NUD_SUM(S, R_i), as shown in Algorithm 2. Thus, similar to LCPF, we propose a new particle filtering algorithm (NUDPF) by using NUD_SUM as the objective function. The details of NUDPF are shown as in Algorithm 3.

DBSCAN [12] is one of the most widely used data mining algorithms for clustering noisy data with outliers. However, it takes no temporal factor into account. We observe that tasks with long execution duration could be as unpromising as remote tasks. In our design, POIs are clustered via the DBSCAN algorithm not only based on their relative distances to each other but also based on the homogeneity of their execution duration. This process is called *iterative clustering*. Iterative clustering contains two steps: the first step is called *duration categorization*, which divides the whole task set into several groups according to each task's execution duration; the second step then applies the DBSCAN routine to each group thus generated in turn to create clusters. This design guarantees that tasks with diverse execution duration will never be assigned into the same

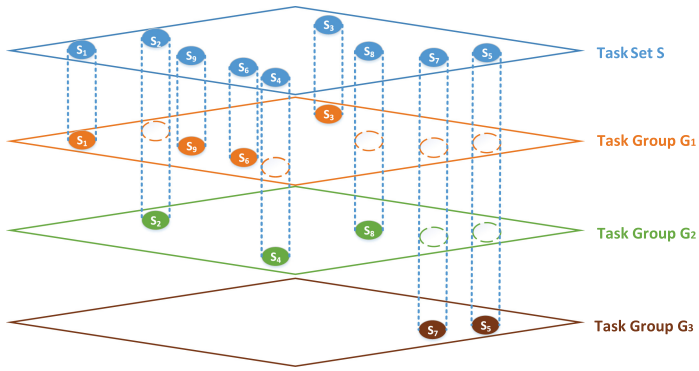


Fig. 4. An illustration of dividing a task set of nine tasks into three task groups through the duration categorization process.

cluster no matter how close they are located geographically. To achieve this goal, besides the two traditional DBSCAN parameters, ϵ and $MinPts$, we introduce a third parameter *duration division ratio* (ddr) into our iterative clustering process. Specifically, ddr determines how many task groups S should be divided into according to each task's execution duration (duration categorization).

In duration categorization, we first pick up the minimum u_{min} and maximum u_{max} out of the set $\{u_{s_i} \mid s_i \in S\}$ and create an interval $I = [u_{min}, u_{max}]$; second, we figure out the number of sub-intervals (n_{div}) that the interval I can be divided into via $n_{div} = \lceil 1/ddr \rceil, ddr \in (0, 1]$; third, letting $\Delta u = (u_{max} - u_{min}) * ddr$, we divide I into n_{div} sub-intervals: $I_1 = [u_{min}, u_{min} + \Delta u]$, $I_2 = (u_{min} + \Delta u, u_{min} + 2 * \Delta u], \ldots, I_{n_{div}-1} = (u_{min} + (n_{div} - 2) * \Delta u, u_{min} + (n_{div} - 1) * \Delta u]$, $I_{n_{div}} = (u_{min} + (n_{div} - 1) * \Delta u, u_{max}]$; last, we put tasks with their respective execution duration falling into the same sub-interval I_i into the same task group G_i ($1 \le i \le n_{div}$) and thus divide the whole task set S into n_{div} task groups $G_1, \ldots, G_{n_{div}}$. Figure 4 illustrates the process of dividing a set S of nine spatial tasks s_{1-9} into three task groups according to their respective execution duration.

After the whole task set S has been divided into n_{div} task groups, $G_1, \ldots, G_{n_{div}}$ by the duration categorization process aforementioned, the first group G_1 containing tasks with the shortest execution duration will be processed by the DBSCAN routine and the outliers thus extracted from G_1 will be added into G_2, which will be processed with the DBSCAN algorithm as well with its outliers being added into G_3 and so forth. When the last task group $G_{n_{div}}$ has been processed by the DBSCAN algorithm, the outliers generated in this step comprise tasks with long execution duration and/or located geographically in some remote areas. The detailed algorithm of iterative clustering is shown in Algorithm 4.

ALGORITHM 4: $IterativeCluster(S, \epsilon, MinPts, ddr)$

```
1: n_div = ⌈1/ddr⌉
2: Divide S into G_1, ..., G_{n_div} by duration categorization.
3: for each G_i ∈ {G_1, ..., G_{n_div}} do
4:     DBSCAN(G_i, ε, MinPts)
5:     if i < n_div then
6:         Put outliers of G_i into G_{i+1}.
7:     end if
8: end for
9: return all clusters and outliers.
```

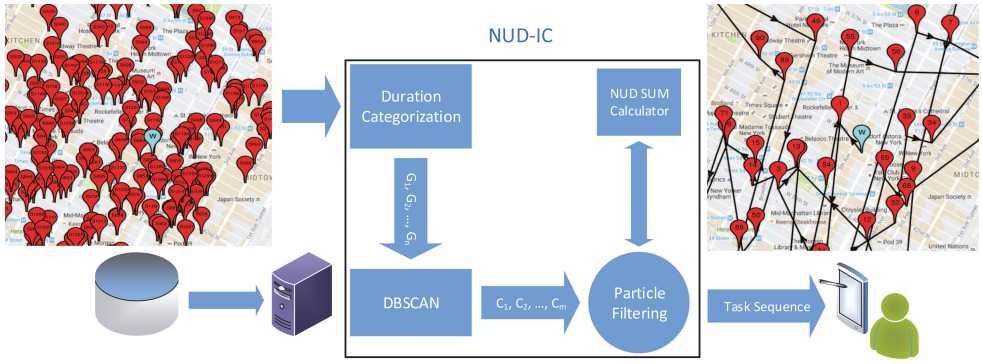


Fig. 5. Illustration of the NUD-IC algorithm.

ALGORITHM 5: $NUD-IC(S, s_0, t_0, ptcl_no, \epsilon, MinPts, ddr)$

1: $R_{final} \leftarrow \emptyset$
2: $(clusters, outliers) \leftarrow IterativeCluster(S, \epsilon, MinPts, ddr)$
3: $s_{cur} \leftarrow s_0$
4: $t_{cur} \leftarrow t_0$
5: **for each** $clst_i \in clusters$ **do**
6: $\quad R_{final} \leftarrow R_{final} \cup NUDPF(clst_i, s_{cur}, t_{cur}, ptcl_no)$
7: $\quad s_{cur} \leftarrow$ the last task in R_{final}
8: $\quad t_{cur} \leftarrow$ the completion time of R_{final}
9: **end for**
10: $R_{final} \leftarrow R_{final} \cup NUDPF(outliers, s_{cur}, t_{cur}, ptcl_no)$
11: **return** R_{final}

The algorithmic description and the block diagram of NUD-IC are shown in Algorithm 5 and Figure 5, respectively. NUD-IC first divides the whole task set into clusters and an outlier set through iterative clustering, and then applies the aforementioned particle filtering method NUDPF to each cluster and the outlier set following the same order as they are generated.

4.3 Algorithmic Comparison

In this section, we first compare our proposed methods to the baseline methods theoretically and then shed light on the performance improvement of the proposed algorithms against the baseline methods using an illustrative example.

Comparison of ICPF and NUD-IC with baselines. Suppose that we use n to represent the total number of tasks and k to represent the number of particles. Then the time complexity of LCPF can be estimated as

$$TSL * k * top_k_select() = O(n) * k * O(k * n) = O(k^2 * n^2), \tag{6}$$

where TSL is the task sequence length and $top_k_select()$ is the cost to select the top k particles in each step. NUD-IC comprises two sub-routines, *IterativeCluster* and *NUDPF* (which dominates

Table 4. Performance Comparison of ICPF and NUD-IC with Baselines

Methods	Optimizing Running Time	Optimizing Number of Accomplished Tasks	Detecting Outliers	Time Complexity
NNH [10]	✓✓✓	×	×	$O(n^2)$
BSH [10]	×	×	×	$O(n^4)$
LCPF	✓✓	✓	×	$O(k^2 * n^2)$
NUD-IC	✓	✓✓✓	✓	$O(k * n^3)$

the whole algorithm). Thus, the time complexity of NUD-IC can be represented as

$$\begin{aligned}\sum_i (TSL_{C_i} * k * NUD()) &= \sum_i (O(|C_i|) * k * O(|C_i|^2)) \\ &= O\left(k * \sum_i |C_i|^3\right) = O\left(k * \left(\sum_i |C_i|\right)^3\right) = O(k * n^3),\end{aligned} \tag{7}$$

where TSL_{C_i} is the task sequence length in cluster C_i, $NUD()$ denotes the cost to compute the NUD values, and $|C_i|$ represents the number of tasks within cluster C_i. On the contrary, the time complexity of NNH [10] is $O(n^2)$ and the time complexity of BSH [10] can be calculated as

$$Search_Tree_Height * N_{st} * NNH() = O(n) * O(n) * O(n^2) = O(n^4), \tag{8}$$

where N_{st} denotes the maximum number of tree nodes in each level in BSH [10], whereas $NNH()$ represents the cost of the NNH [10] approach that is used as lower bound computation in BSH [10]. Table 4 compares the characteristics of each algorithm under different application scenarios, where more check marks indicate better performance.

An illustrative example. Figure 2 compares NUD-IC to NNH [10] as an illustrative example. Since NNH [10] always chooses the task with the least total of traveling time plus execution duration, NNH [10] will return $w = \{s_2\}$ as the result. In this case, only s_2 can be accomplished by w since all other tasks, s_1, s_3, s_4, s_5, have expired when s_2 is completed. However, when applying NUD-IC, at first, s_1, s_3, and s_5 will be clustered together, whereas s_2 and s_4 will be detected as outliers by *IterativeCluster* (as shown in Algorithm 4). Next, the sub-routine *NUDPF* (as shown in Algorithm 3) runs on the cluster including s_1, s_3, and s_5 to determine which task should be selected first by comparing their respective resulting *NUD_SUM* values. Specifically, if w chooses s_1, the completion time of s_1 will be the traveling time between w and s_1 plus the execution duration of s_1, which is $9 + 2 = 11$. Therefore, s_1's *NUD_SUM* value $NUD_SUM(S, w \rightarrow s1) = NUD(s_3, s_1, 11) + NUD(s_5, s_1, 11) = \frac{23-2-3-11}{2} + \frac{27-4-4-11}{4} = 3.5 + 2 = 5$. Notice that here both s_3 and s_5 are potential successors of s_1 at time 11. Similarly, we can also derive s_3's *NUD_SUM* value, $NUD_SUM(S, w \rightarrow s_3) = NUD(s_1, s_3, 12) + NUD(s_5, s_3, 12) = \frac{20-2-3-12}{2} + \frac{27-4-3-12}{4} = 1.5 + 2 = 3.5$, s_5's *NUD_SUM* value, $NUD_SUM(S, w \rightarrow s_5) = NUD(s_3, s_5, 17) = \frac{23-2-3-17}{2} = 0.5$. According to Algorithm 3, since choosing s_1 will lead to the highest *NUD_SUM* value, s_1 will be selected and assigned to w as the worker's first task for completion. The aforementioned selection procedure repeats until all tasks in each cluster have been assigned first or already expired. Once all tasks in each cluster have been assigned, the algorithm continues to check on outlier tasks by comparing their respective *NUD_SUM* values to determine the next task to take. As shown in Figure 2, at last, NUD-IC yields $w = \{s_1 \rightarrow s_3 \rightarrow s_5 \rightarrow s_4\}$ as the final task assignment, with four tasks completed in total.

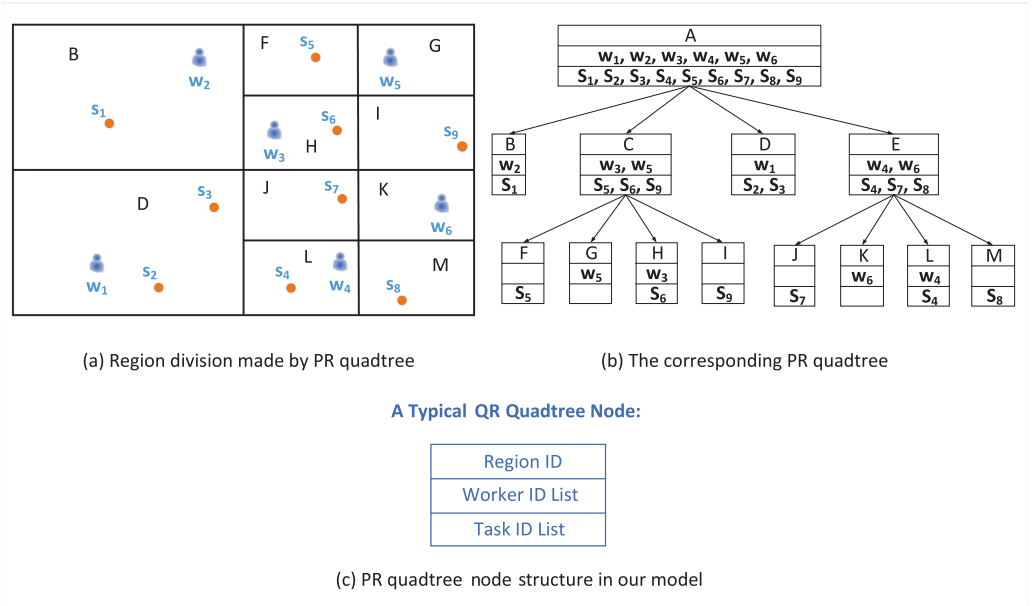


Fig. 6. An illustration of how a region A is partitioned using a PR quadtree. w_{1-6} represent workers and s_{1-9} represent tasks.

5 MULTI-WORKER GEO-TASK SCHEDULING

In this section, we elaborate on our solutions for the MGTS problem. The major challenge in MGTS is how to divide the whole task set into spatial sub-sets to ensure workload balancing among workers. Note that simply running any SGTS approaches for each worker sequentially does not work here. We propose two space partitioning-based methods, QT-NNH and QT-NUD, for MGTS, inspired by the proposed LCPF and NUD-IC for the SGTS problem. QT-NNH and QT-NUD leverage PR quadtree to ensure workload balancing among multiple workers.

5.1 Space Partitioning Using Point-Region Quadtree

Turbo-GTS employs PR quadtree [31] to perform space partitioning to ensure task workload balancing among multiple workers. PR quadtree is a quadtree organized in the same way as region quadtree, except its leaf nodes are either empty or contain a data point with its coordinates. In our design, if a node in PR quadtree contains more than one worker, then the node will generate four children; otherwise, the node will serve as a leaf node. Figure 6(a) illustrates how a spatial region with six workers and nine tasks can be partitioned using PR quadtree. The corresponding tree structure is shown in Figure 6(b). Figure 6(c) depicts the structure of a PR quadtree node. A PR quadtree node consists of three parts: the region of which the tree node represents, the workers who will be assigned to this region, and all tasks located in the region. As illustrated in Figure 6(a) and (b), at first, a PR quadtree root is created to represent the whole region A containing all workers and tasks. Next, because region A contains more than one worker, A is further divided into four sub-regions B, C, D, and E, each of which constitutes a child node of A, and each worker or task inside A is assigned to one of A's children based on their locations. Because each of region B and region D contains only one worker, they stop splitting and become leaf nodes. Region C and region E both contain more than one worker, so they will split into F, G, H, I and J, K, L, M, respectively. Such recursive splitting process stops when all leaf nodes contain no more than one worker.

ALGORITHM 6: $QT-NNH(Z, W, S)$

```
1: Create an empty PR quadtree qt.
2: qt.root ← {Z, W, S}.
3: TopDownDivideNNH(qt.root)
```

ALGORITHM 7: *TopDownDivideNNH*(qt_node)

```
1: if qt_node.NumberOfWorkers > 1 then
2:     {chld_1, ..., chld_4} ← qt_node.split()
3:     for each chld_i ∈ {chld_1, ..., chld_4} do
4:         TopDownDivideNNH(chld_i)
5:     end for
6:     for each chld_i ∈ {chld_1, ..., chld_4} do
7:         wk_set ← ∅
8:         for each chld_j ∈ {chld_1, ..., chld_4} and chld_j ≠ chld_i do
9:             wk_set ← wk_set ∪ chld_j.getWorkers()
10:        end for
11:        while wk_set ≠ ∅ do
12:            w* ← argmin{w_i.getCurrentTime() + travel_cost(w_i, chld_i) | w_i ∈ wk_set}
13:            NNH(chld_i.getTasks(), w*.getCurrentLocation(), w*.getCurrentTime())
14:            wk_set ← wk_set \ {w*}
15:        end while
16:    end for
17: else
18:    if qt_node.NumberOfWorkers = 1 then
19:        w ← qt_node.getWorkers()
20:        NNH(qt_node.getTasks(), w.getCurrentLocation(), w.getCurrentTime())
21:    end if
22: end if
```

5.2 QT-NNH and QT-NUD

The algorithmic design of QT-NNH and QT-NUD are shown in Algorithms 6 and 8, respectively. Both QT-NNH and QT-NUD are based on PR quadtree discussed in the preceding section. In QT-NNH, a PR quadtree root node is created first to represent the whole region Z including the worker set W and task set S (lines 1 and 2 in Algorithm 6). Such a root node is passed as a parameter to a recursive function *TopDownDivideNNH* (line 3 in Algorithm 6). *TopDownDivideNNH* then checks the received parameter qt_node to determine whether qt_node contains more than one worker (line 1 in Algorithm 7). If qt_node contains more than one worker, qt_node will then be split into four children and *TopDownDivideNNH* will be invoked on each child recursively (lines 2 through 5 in Algorithm 7). In this way, PR quadtree is constructed recursively. Once PR quadtree is created, every leaf node will be checked to determine whether there is one worker associated with the node qt_node (line 18 in Algorithm 7). If yes, NNH [10] is invoked for the current leaf node (lines 19 and 20 in Algorithm 7) to identify tasks for the associated worker—that is, we assign each worker to the tasks in the same node first. After all workers have been designated to the tasks in their own nodes individually, we continue to dispatch workers to the tasks that have not been assigned in the corresponding three sibling nodes (lines 6 through 16 in Algorithm 7). A worker with an earlier arrival time will have a higher priority to select a task (line 12 in Algorithm 7). This process repeats iteratively from the leaf nodes to the root node. QT-NUD follows the same process, except

QT-NUD employs NUDPF (as illustrated in Algorithm 9) in each tree node and its corresponding three sibling nodes to dispatch workers to tasks iteratively.

ALGORITHM 8: $QT-NUD(Z, W, S, pctl_no)$

```
Create an empty PR quadtree qt.
qt.root ← {Z, W, S}.
TopDownDivideNUD(qt.root, pctl_no)
```

ALGORITHM 9: $TopDownDivideNUD(qt_node, pctl_no)$

```
if qt_node.NumberOfWorkers > 1 then
    {chld_1, ..., chld_4} ← qt_node.split()
    for each chld_i ∈ {chld_1, ..., chld_4} do
        TopDownDivideNUD(chld_i, pctl_no)
    end for
    for each chld_i ∈ {chld_1, ..., chld_4} do
        wk_set ← ∅
        for each chld_j ∈ {chld_1, ..., chld_4} and chld_j ≠ chld_i do
            wk_set ← wk_set ∪ chld_j.getWorkers()
        end for
        while wk_set ≠ ∅ do
            w* ← argmin{w_i.getCurrentTime() + travel_cost(w_i, chld_i) | w_i ∈ wk_set}
            NUDPF(chld_i.getTasks(), w*.getCurrentLocation(), w*.getCurrentTime(), pctl_no)
            wk_set ← wk_set \ {w*}
        end while
    end for
else
    if qt_node.NumberOfWorkers = 1 then
        w ← qt_node.getWorkers()
        NUDPF(qt_node.getTasks(), w.getCurrentLocation(), w.getCurrentTime(), pctl_no)
    end if
end if
```

5.3 Algorithmic Comparison

In this section, we first compare our proposed methods to the baseline methods theoretically and then shed light on the performance improvement of the proposed algorithms against the baseline methods using an illustrative example.

Comparison of QT-NNH and QT-NUD with baselines. Suppose we use n_t to represent the total number of tasks, n_w to represent the total number of workers, and k to represent the number of particles. The time complexity of QT-NNH equals $QuadTree_Create_Cost + Scheduling_Cost$. In the worst case, $QuadTree_Create_Cost = O(n_t + n_w)$ and $Scheduling_Cost = O(n_t^2)$. Therefore, the time complexity of QT-NNH is $O(n_t + n_w) + O(n_t^2) = O(n_t^2)$. Similarly, the time complexity of QT-NUD can be represented as $O(n_t + n_w) + O(k * n_t^3) = O(k * n_t^3)$. On the contrary, BLALS-T [11] is composed of two major parts: recursive top-down bisection partitioning and bottom-up merging. Therefore, the time complexity of BLALS-T [11] equals $\sum_i [H_{partition}(i) + H_{merge}(i)] + H_r = \Omega(H_{merge}(1)) = \Omega((n_w + n_t)^2 * n_w * n_t)$, where i represents the i-th iteration of the outermost loop in BLALS-T [11]. $H_{partition}(i)$ and $H_{merge}(i)$ represent the cost of recursive top-down bisection partitioning and bottom-up merging in the i-th iteration. H_r is the cost of scheduling

Table 5. Performance Comparison of QT-NNH and QT-NUD with Baselines

Methods	Optimizing Running Time	Optimizing # of Accomplished Tasks	Detecting Outliers	Time Complexity
BLALS-T [11]	×	×	×	$\Omega((n_w + n_t)^2 * n_w * n_t)$
QT-NNH	✓✓	✓	×	$O(n_t^2)$
QT-NUD	✓	✓✓	✓	$O(k * n_t^3)$

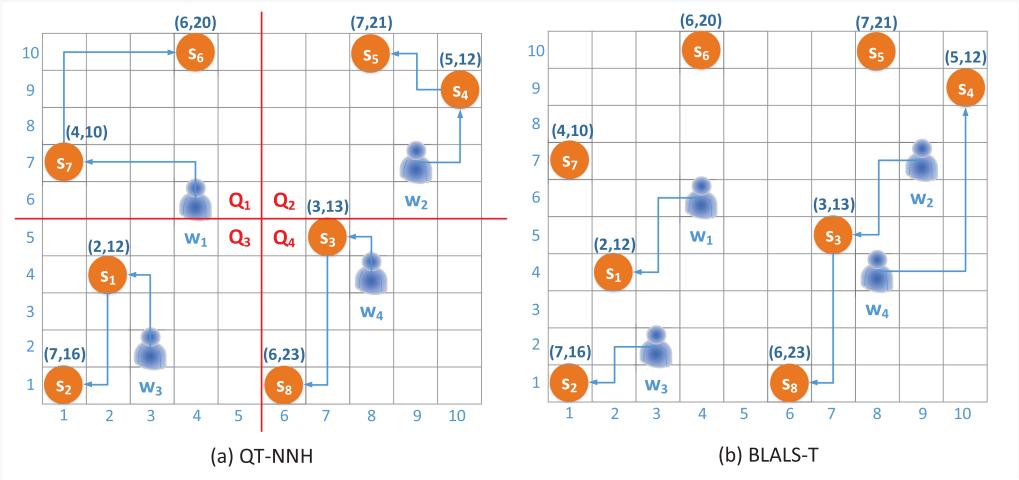
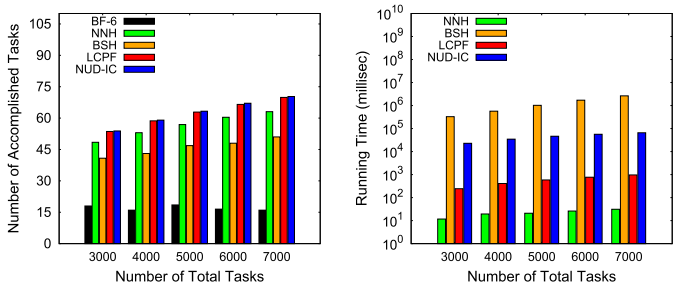


Fig. 7. A comparison of the results when QT-NNH and BLALS-T are applied respectively to the same scenario, where w_{1-4} represent workers and s_{1-8} represent tasks. Parenthesized number pairs on top of each task represent this task's execution duration and expiration time with the same time unit as the scales in the grid, where distances are calculated in taxicab geometry.

the remaining tasks when all recursive top-down bisection partitioning and bottom-up merging are finished. Table 5 compares the characteristics of each algorithm under different application scenarios, where more check marks indicate better performance.

An illustrative example. Figure 7 compares QT-NNH to BLALS-T [11] as an illustrative example. As shown in Figure 7(a), QT-NNH at first employs the proposed quadtree to divide the whole target spatial region into four sub-regions (i.e., four tree leaf nodes), let us say, Q_1, Q_2, Q_3, and Q_4 (represented using solid red lines). Next, the tasks in each leaf node will be assigned to its associated worker one by one according to NNH [10] (for QT-NNH) or NUD-IC (for QT-NUD) proposed for the SGTS scenario. If all tasks in a tree node have been assigned, the algorithm continues to check on the tasks in the sibling tree nodes and then the parent nodes iteratively in a bottom-up manner. For the scenario as shown in Figure 7(a), QT-NNH will return the following task assignment for each worker: $w_1 = \{s_7 \rightarrow s_6\}$, $w_2 = \{s_4 \rightarrow s_5\}$, $w_3 = \{s_1 \rightarrow s_2\}$,$w_4 = \{s_3 \rightarrow s_8\}$, with all eight tasks completed.

On the contrary, BLALS-T [11] assigns each worker the task that incurs the least cost to her current task sequence according to its insertion scheduling. This could lead to conflicts where multiple workers compete for the same tasks and some far-away tasks will be ignored since workers' spatial regions may overlap. As shown in Figure 7(b), BLALS-T [11] will generate the following task assignments for workers: $w_1 = \{s_1\}$, $w_2 = \{s_3 \rightarrow s_8\}$, $w_3 = \{s_2\}$, $w_4 = \{s_4\}$. Consequently, there are only five tasks, s_1, s_2, s_3, s_4, s_8, that can be completed, with no worker being able to complete s_5, s_6 and s_7.



(a) Number of Accomplished Tasks (Syn-uniform)

(b) CPU Cost (Syn-uniform)

Fig. 8. Effect of task number on the synthetic data of uniform distribution with expiration time range [40%, 60%].

6 EXPERIMENTAL EVALUATION FOR SGTS ALGORITHMS

In this section, we investigate the performance of LCPF and NUD-IC in terms of the number of accomplished tasks and CPU cost in support of SGTS using both real and synthetic datasets. All experiments were run on a machine with POWER8E CPUs at 2.06 GHz and 1,024 GB of RAM. Each result is averaged over 2,000 random queries (except BF-6 and BSH [10]), where the location of the worker was randomly generated in the target geographic areas.

6.1 All SGTS Methods for Comparison

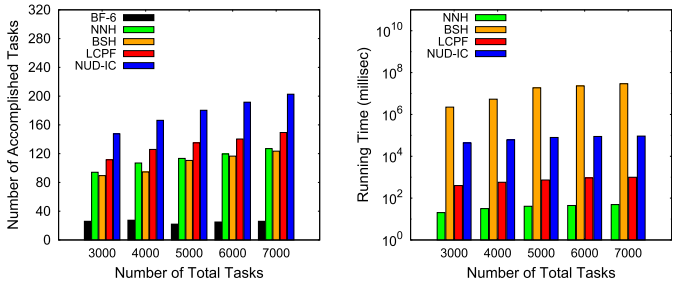
- *BF-6*: The brute force scheduling algorithm, which runs up to 6 hours.
- *NNH* [10]: The method in which the worker always chooses the next task that has the least total cost of the traveling distance plus the task's execution duration.
- *BSH* [10]: The method that stores a predetermined number k (i.e., the beam width) of best partial task sequence in a container (i.e., the beam) and keeps extending them until a solution is found. The beam width k in BSH [10] was set the same as the number of particles *ptcl_no* (default value 5) used in LCPF and NUD-IC.
- *LCPF*: Our proposed method based on particle filtering, as described in our conference paper [27].
- *NUD-IC*: Our proposed method based on the NUD computation and iterative clustering, as described in our conference paper [27].

6.2 Experiments on Synthetic Data

We generated our synthetic data based on uniform and skewed distributions. For the skewed distribution, all tasks were generated according to four Gaussian clusters, each of which had the same radius r and the same deviation σ (with 0.2 as the default value), in a target region with edge length $6r$. We randomly generated the expiration time for each task from [20%, 40%], [40%, 60%], [60%, 80%], and [80%, 100%] of the total expiration time e_{total} (with [40%, 60%] as the default range).

6.2.1 Impact of the Total Number of Tasks.

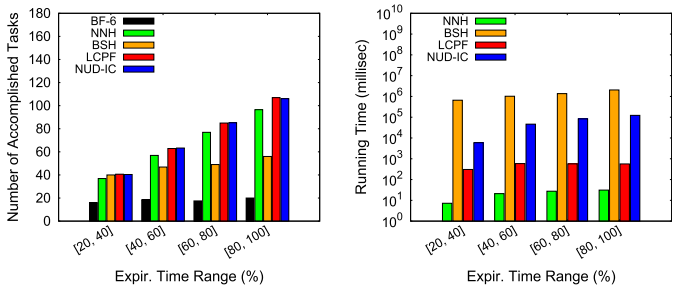
Uniform distribution. As shown in Figure 8, both LCPF and NUD-IC outperformed BF-6, NNH [10], and BSH [10] in terms of the number of accomplished tasks. However, the CPU cost of BSH [10], LCPF, and NUD-IC was higher than NNH [10]. Except BF-6, BSH [10] returned the fewest accomplished tasks and required the longest CPU time among all investigated algorithms because it works well only when the total number of tasks is small.



(a) Number of Accomplished Tasks (Syn-skewed)

(b) CPU Cost (Syn-Skewed)

Fig. 9. Effect of task number on the synthetic data of skewed distribution ($\sigma = 0.2$) with expiration time range [40%, 60%].



(a) Number of Accomplished Tasks (Syn-uniform)

(b) CPU Cost (Syn-uniform)

Fig. 10. Effect of expiration time on the synthetic data of uniform distribution with task number 5,000.

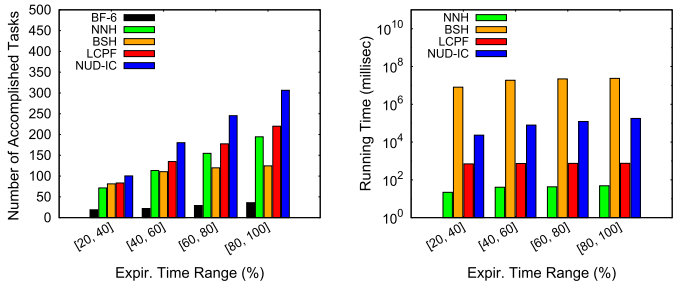
Skewed distribution. As shown in Figure 9, NUD-IC outperformed all other algorithms in terms of the number of accomplished tasks. When the number of total tasks increased, their performance difference became more noticeable. For example, when the total number of tasks equaled 6,000, NUD-IC was able to return 192 tasks, whereas NNH [10] only returned 120 tasks.

6.2.2 Impact of the Task Expiration Time.

Uniform distribution. As shown in Figure 10, when we extended the task expiration time, NUD-IC and LCPF outperformed all other algorithms in terms of the number of accomplished tasks. For example, when the expiration time was [80%, 100%], NUD-IC accomplished 106 tasks, whereas BSH [10] only returned 56 tasks.

Note that BSH [10] showed little improvement in terms of the number of completed tasks when the expiration time increased. This is because BSH [10] always favors the tasks that have more potential successors, leading to a higher probability of choosing unpromising tasks when the expiration time of all tasks is extended. In other words, BSH [10] only works well in the case that all tasks tend to expire soon. If most of the tasks have a long expiration time, the performance of BSH [10] would deteriorate noticeably.

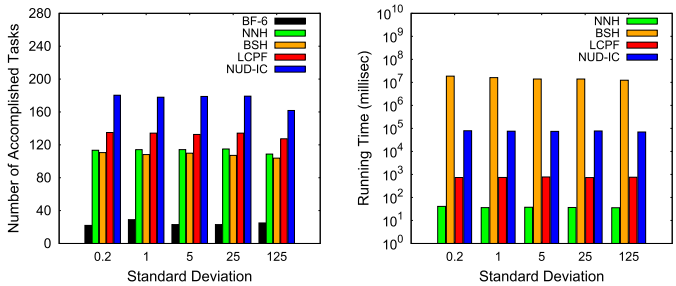
Skewed distribution. Figure 11 shows that as the expiration time was prolonged, the number of accomplished tasks returned by NUD-IC was up to 1.6 times more than the tasks accomplished by NNH [10]. For example, when the expiration time is [60%, 80%], the number of accomplished tasks obtained by NUD-IC and NNH [10] was 245 and 155, respectively.



(a) Number of Accomplished Tasks (Syn-skewed)

(b) CPU Cost (Syn-Skewed)

Fig. 11. Effect of expiration time on the synthetic data of skewed distribution ($\sigma = 0.2$) with task number 5,000.



(a) Number of Accomplished Tasks (Syn-skewed)

(b) CPU Cost (Syn-Skewed)

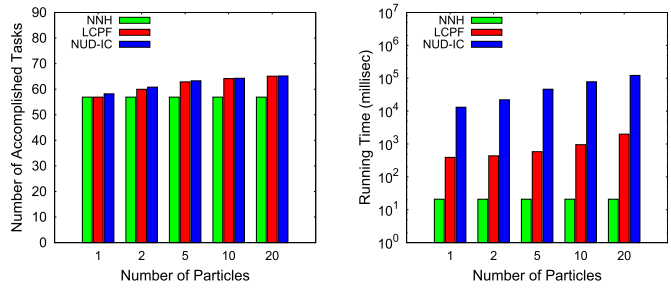
Fig. 12. Effect of σ on the synthetic data of skewed distribution with task number 5,000 and expiration time range [40%, 60%].

6.2.3 Impact of the Task Spatial Distribution. As shown in Figure 12, when we increased σ for skewed distribution, the number of accomplished tasks of all algorithms went down. For example, when σ equaled 0.2, the number of completed tasks returned by NUD-IC and NNH [10] was 180 and 113, respectively, whereas when σ rose to 125, the number of accomplished tasks obtained by NUD-IC and NNH [10] declined to 162 and 109, respectively. In addition, we can observe that the more skewed the task distribution, the larger the advantage of NUD-IC over NNH [10]. NUD-IC outperformed other algorithms in terms of the number of accomplished tasks, and the CPU cost of all algorithms slightly decreased in the increase of σ.

6.2.4 Impact of the Number of Particles. Figures 13 and 14 show the impact of the number of particles on the performance of LCPF and NUD-IC. We can observe that when there was only one particle used in LCPF, it returned the same number of accomplished tasks as NNH [10]. As the number of particles increased, the performance gain of both LCPF and NUD-IC over NNH [10] in terms of the number of accomplished tasks became larger.

6.3 Experiments on Real Geo-Task Data

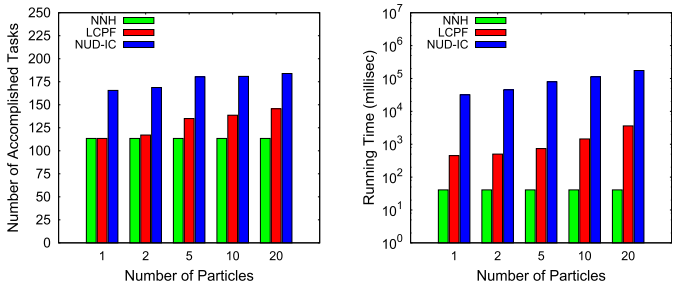
Our real data were collected from Foursquare [36–38], featuring New York City (42,981 tasks) and Austin, Texas (8,569 tasks). For task execution duration, we set *dur_min* to 5 seconds and *dur_max* to 5 minutes. The expiration time of each task was randomly generated from the following four



(a) Number of Accomplished Tasks (Syn-uniform)

(b) CPU Cost (Syn-uniform)

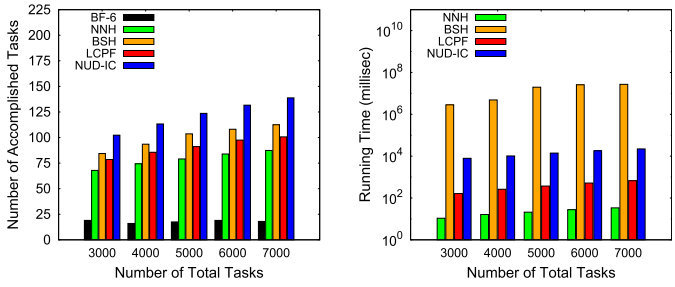
Fig. 13. Effect of the number of particles on the synthetic data of uniform distribution with task number 5,000 and expiration time range [40%, 60%].



(a) Number of Accomplished Tasks (Syn-skewed)

(b) CPU Cost (Syn-skewed)

Fig. 14. Effect of the number of particles on the synthetic data of skewed distribution ($\sigma = 0.2$) with task number 5,000 and expiration time range [40%, 60%].



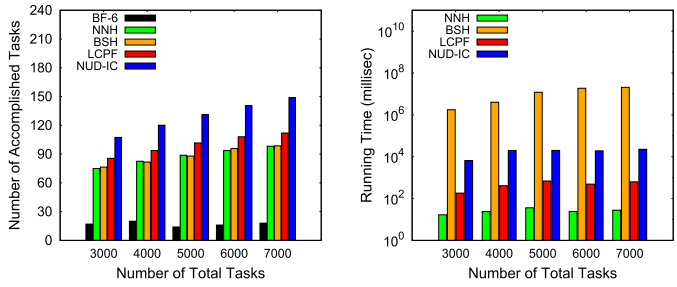
(a) Number of Accomplished Tasks (New York)

(b) CPU Cost (New York)

Fig. 15. Effect of task number on the dataset of New York City with expiration time range [60 min, 90 min].

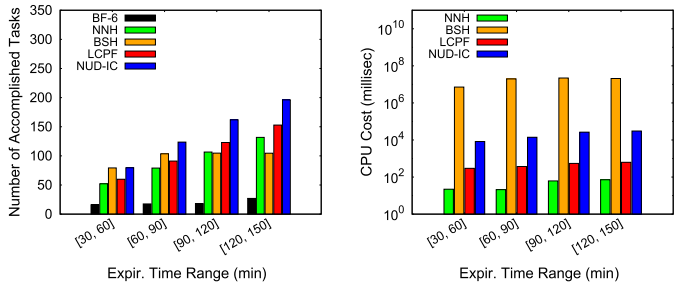
intervals: 30 to 60 minutes, 60 to 90 minutes, 90 to 120 minutes, and 120 to 150 minutes, with 60 to 90 minutes as the default interval.

6.3.1 Impact of the Total Number of Tasks. Figures 15 and 16 show the performance of all algorithms in terms of the number of accomplished tasks and CPU cost based on POI distributions in New York City and Austin, Texas. NUD-IC was able to return more accomplished tasks than



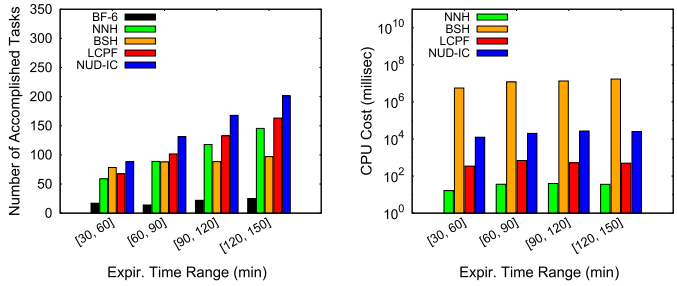
(a) Number of Accomplished Tasks (Austin) (b) CPU Cost (Austin)

Fig. 16. Effect of task number on the dataset of Austin, Texas, with expiration time range [60 min, 90 min].



(a) Number of Accomplished Tasks (New York) (b) CPU Cost (New York)

Fig. 17. Effect of expiration time on the dataset of New York City with task number 5,000.

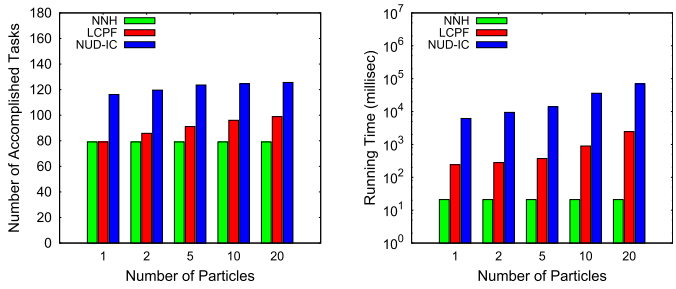


(a) Number of Accomplished Tasks (Austin) (b) CPU Cost (Austin)

Fig. 18. Effect of expiration time on the dataset of Austin, Texas, with task number 5,000.

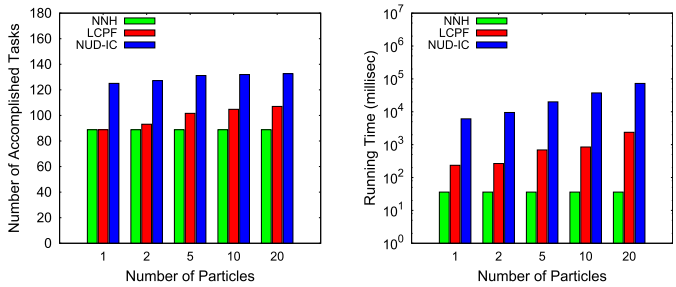
NNH [10] and LCPF at the price of the increased CPU cost. For example, when the total number of tasks was 7,000 in New York City, the number of accomplished tasks returned by NUD-IC was 139, 1.6 times more than that of NNH [10], which was 87. However, the CPU cost of all algorithms increased as the total number of tasks enlarged.

6.3.2 Impact of the Task Expiration Time. As depicted in Figures 17 and 18, when we extended the task expiration time, NUD-IC outperformed the other algorithms in terms of the number of accomplished tasks. For example, when the expiration time was in the interval of [90 min, 120 min]



(a) Number of Accomplished Tasks (New York) (b) CPU Cost (New York)

Fig. 19. Effect of number of particles on the dataset of New York City with task number 5,000, expiration time range [60 min, 90 min].



(a) Number of Accomplished Tasks (Austin) (b) CPU Cost (Austin)

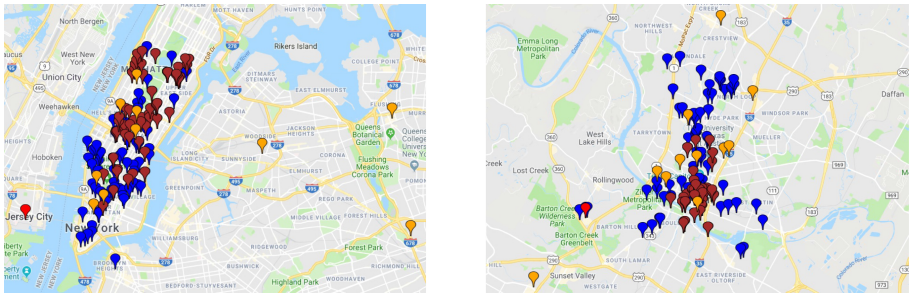
Fig. 20. Effect of number of particles on the dataset of Austin, Texas, with task number 5,000, expiration time range [60 min, 90 min].

in New York City, the number of accomplished tasks returned by NUD-IC was 162, 1.5 times more than that obtained by NNH [10], which was 107. BSH [10] showed little improvement in terms of the number of accomplished tasks when all tasks expire after 90 minutes in both cities.

6.3.3 Impact of the Number of Particles. Here we investigate the impact of the number of particles on the performance of LCPF and NUD-IC. As we raised the number of particles, both the number of accomplished tasks and CPU cost of LCPF and NUD-IC increased, as shown in Figures 19 and 20. In addition, given the same number of particles, NUD-IC successfully returned many more accomplished tasks than LCPF. Note that when we set the number of particles to 1, the performance of LCPF degraded to the same level as NNH [10].

6.4 Case Study

In the case study described in this section, we randomly selected 1,000 spatial tasks from New York City and 1,000 spatial tasks from Austin, Texas, respectively, then launched all algorithms from a random start location of the worker in each city. In the case of New York City, BF-6, NNH [10], BSH [10], LCPF, and NUD-IC returned 23 tasks, 42 tasks, 67 tasks, 117 tasks, and 125 tasks, respectively, whereas for Austin, Texas, BF-6, NNH [10], BSH [10], LCPF, and NUD-IC retrieved 21 tasks, 61 tasks, 61 tasks, 119 tasks, and 132 tasks, respectively. Figure 21(a) and (b) depict the corresponding accomplished tasks returned by NUD-IC and BSH [10] respectively in both New York City and



(a) New York City (b) Austin

Fig. 21. Spatial tasks returned by NUD-IC and BSH [10] respectively in New York City (a) and Austin, Texas (b). The red dot represents a worker's start location, blue dots are tasks accomplished exclusively by NUD-IC, orange dots are tasks accomplished exclusively by BSH [10], and brown dots are tasks accomplished by both.

Austin, Texas. Specifically, as illustrated in Figure 21(a) and (b), NUD-IC was able to return a much higher number of spatial tasks than BSH [10] because BSH [10] works well only when the number of spatial tasks is very few (e.g., less than 100).

7 EXPERIMENTAL EVALUATION FOR MGTS ALGORITHMS

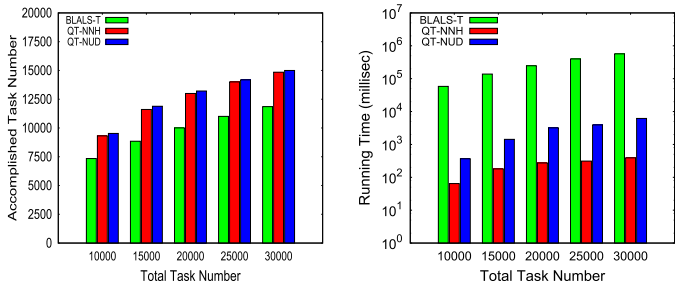
In this section, we investigate the performance of QT-NNH and QT-NUD in terms of the number of accomplished tasks and CPU cost in support of MGTS on real data. All experiments were run in the same setting as in the last section unless otherwise specified. The real data were collected from Foursquare [36–38], featuring New York City (42,981 tasks) and Tokyo (67,124 tasks). For task execution duration, we set *dur_min* to 5 seconds and *dur_max* to 5 minutes. The expiration time of each task was randomly generated from the following three intervals: 30 to 90 minutes, 60 to 120 minutes, and 90 to 150 minutes, with 60 to 120 minutes as the default interval.

7.1 All MGTS Methods for Comparison

- *BLALS-T* [11]: The method that partitions the worker-task flow network from top to down recursively and then merges all partitions from bottom to up.
- *QT-NNH*: Our proposed method that leverages PR quadtree to ensure workload balancing among multiple workers and schedule tasks for each worker via NNH [10].
- *QT-NUD*: Our proposed method that leverages PR quadtree to ensure workload balancing among multiple workers and schedule tasks for each worker via NUD-IC [27].

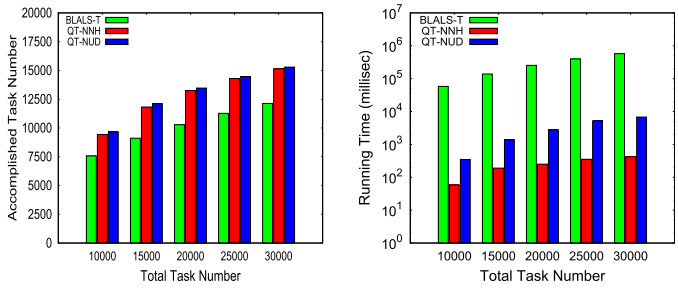
7.2 Impact of the Total Number of Tasks

Figures 22 and 23 show the impact of the total number of tasks on the performance of all algorithms in terms of the number of accomplished tasks and CPU cost in New York City and Tokyo. We can observe that BLALS-T not only returned fewer accomplished tasks than QT-NNH and QT-NUD but also required much longer CPU time in execution. This occurred for three reasons. First, BLALS-T assumed that each worker does not leave her original location too far away, and she can only choose tasks within her neighborhood. However, in reality, workers would be willing to travel far as long as the benefit can cover their travel cost. Second, BLALS-T assumed that each worker cannot be assigned to more tasks than a predefined threshold, which is not the case in reality. Last but not least, the insertion scheduling method used in BLALS-T [11] was extremely inefficient when the ratio of the number of tasks over workers was high .



(a) Number of Accomplished Tasks (New York) (b) CPU Cost (New York)

Fig. 22. Effect of the total number of tasks on the dataset of New York City with total number of workers 300 and expiration time range [60 min, 120 min]; for QT-NUD, $ptcl_no = 1$.



(a) Number of Accomplished Tasks (Tokyo) (b) CPU Cost (Tokyo)

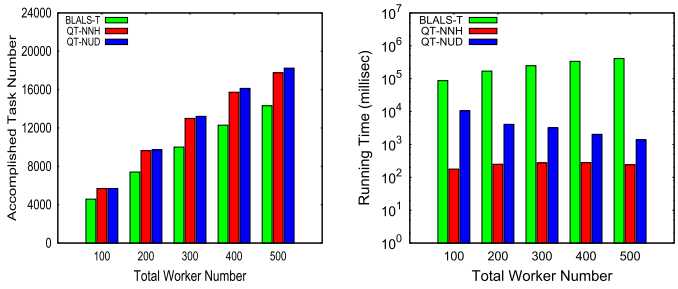
Fig. 23. Effect of the total number of tasks on the dataset of Tokyo with total number of workers 300 and expiration time range [60 min, 120 min]; for QT-NUD, $ptcl_no = 1$.

7.3 Impact of the Total Number of Workers

Figures 24 and 25 show the impact of the total number of workers on the performance of the algorithms in terms of the number of accomplished tasks and CPU cost in New York City and Tokyo. As depicted in these figures, when the number of workers increased, QT-NUD and QT-NNH outperformed BLALS-T in terms of both the number of accomplished tasks and CPU time. Note that the CPU cost of QT-NUD went down as the number of workers was higher. This is because when the number of workers increased and the number of tasks remained the same, the number of tasks assigned to each worker declined, which makes QT-NUD more efficient.

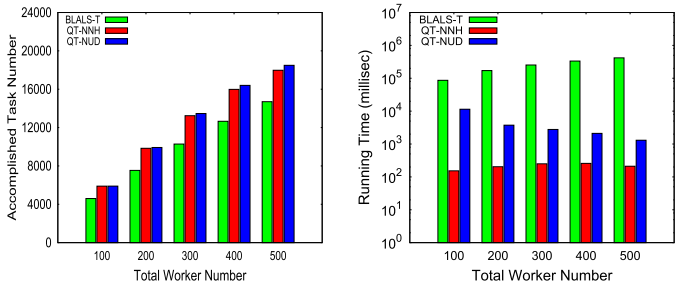
7.4 Impact of the Expiration Time

Figures 26 and 27 show the impact of the expiration time on the performance of all algorithms in terms of the number of accomplished tasks and CPU cost in New York City and Tokyo. As shown in these figures, QT-NUD and QT-NNH outperformed BLALS-T in terms of both the number of accomplished tasks and CPU time as the expiration time increased. With the increased expiration time, all three approaches returned a higher number of completed tasks.



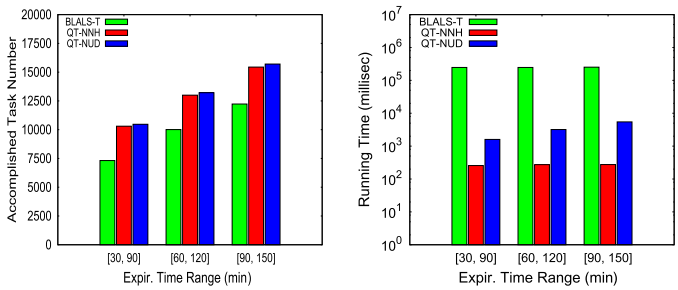
(a) Number of Accomplished Tasks (New York) (b) CPU Cost (New York)

Fig. 24. Effect of the total number of workers on the dataset of New York City with total number of tasks 20,000 and expiration time range [60 min, 120 min]; for QT-NUD, $ptcl_no = 1$.



(a) Number of Accomplished Tasks (Tokyo) (b) CPU Cost (Tokyo)

Fig. 25. Effect of the total number of workers on the dataset of Tokyo with total number of tasks 20,000 and expiration time range [60 min, 120 min]; for QT-NUD, $ptcl_no = 1$.

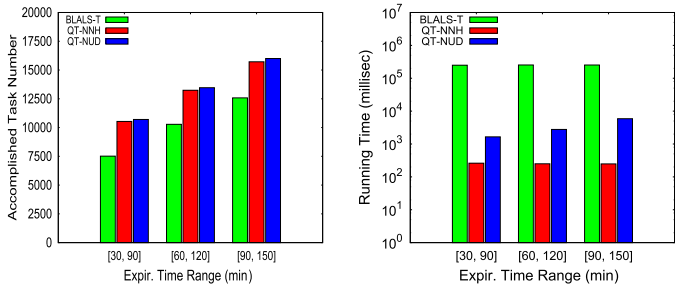


(a) Number of Accomplished Tasks (New York) (b) CPU Cost (New York)

Fig. 26. Effect of the expiration time on the dataset of New York City with total number of workers 300 and total number of tasks 20,000; for QT-NUD, $ptcl_no = 1$.

7.5 Impact of the Number of Particles

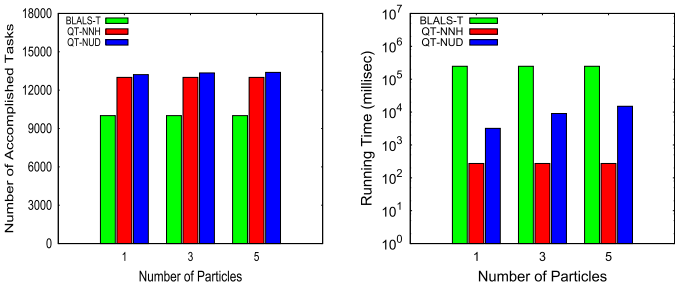
Figures 28 and 29 show the impact of the number of particles on the performance of QT-NNH and QT-NUD in terms of the number of accomplished tasks and CPU cost in New York City and Tokyo. As we can see in these figures, it was observed that when the number of particles increased,



(a) Number of Accomplished Tasks (Tokyo)

(b) CPU Cost (Tokyo)

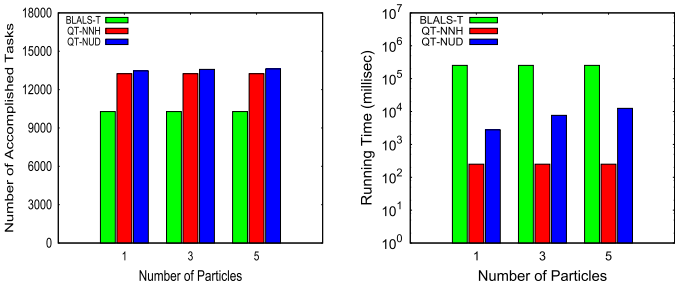
Fig. 27. Effect of the expiration time on the dataset of Tokyo with total number of workers 300 and total number of tasks 20,000; for QT-NUD, $ptcl_no = 1$.



(a) Number of Accomplished Tasks (New York)

(b) CPU Cost (New York)

Fig. 28. Effect of the number of particles on the dataset of New York City with total number of workers 300, total number of tasks 20,000, and expiration time range [60 min, 120 min].



(a) Number of Accomplished Tasks (Tokyo)

(b) CPU Cost (Tokyo)

Fig. 29. Effect of the number of particles on the dataset of Tokyo with total number of workers 300, total number of tasks 20,000, and expiration time range [60 min, 120 min].

the performance gain of QT-NNH and QT-NUD in terms of the number of completed tasks was very limited, whereas the CPU cost of QT-NUD increased. This is because in QT-NNH and QT-NUD, due to the use of PR quadtree for space partitioning, each worker was assigned to only a small number of tasks. Therefore, in this case, adding the number of particles would not lead to a noticeable increase in the number of completed tasks.

8 CONCLUSION AND FUTURE WORK

In this article, we propose a novel framework, Turbo-GTS, in support of large-scale GTS, with the objective of identifying an optimal task assignment for each worker to maximize the total number of tasks that can be completed for an entire worker group, given the geographic locations of each task and each worker. We propose two sub-optimal approaches (LCPF and NUD-IC) based on particle filtering and DBSCAN for the SGTS problem and extended our work to solve the MGTS problem by proposing two space partitioning–based methods (QT-NNH and QT-NUD), which leverage PR quadtree to ensure workload balancing. The effectiveness and efficiency of the four proposed approximate solutions are verified by our extensive experiments using both real and synthetic data. Compared to the state-of-the-art approaches, our proposed solutions are able to return a higher number of completed tasks for the worker group while reducing the computation cost by up to three orders of magnitude when coping with massive tasks distributed in large geographic areas.

REFERENCES

[1] Amelia Acker, Martin Lukac, and Deborah Estrin. 2010. *Participatory Sensing for Community Data Campaigns: A Case Study*. Technical Report. Center for Embedded Network Sensing.

[2] Allan Afuah and Christopher L. Tucci. 2012. Crowdsourcing as a solution to distant search. *Academy of Management Review* 37, 3 (2012), 355–375.

[3] Edward Baig. 2012. Quri App Lets You Make Money While You Shop. Retrieved November 16, 2019 from http://content.usatoday.com/communities/technologylive/post/2012/05/quri-easyshift-shopper-app/1#.Vd6ybPnT4ds.

[4] Daren C. Brabham. 2008. Crowdsourcing as a model for problem solving an introduction and cases. *Convergence* 14, 1 (2008), 75–90.

[5] Nicole Carter. 2012. TaskRabbit: From Start-Up to Global Web Market for Odd Jobs. Retrieved November 16, 2019 from http://www.inc.com/nicole-carter-and-tim-rice/task-rabbit-leah-busque-start-up-to-global-web-marketplace-jobs.html.

[6] Khanh-Hung Dang and Kim-Tuyen Cao. 2013. Towards reward-based spatial crowdsourcing. In *Proceedings of the 2013 13th International Conference on Control, Automation, and Systems (ICCAIS'13)*. 363–368.

[7] Mathijs M. de Weerdt, Yingqian Zhang, and Tomas Klos. 2012. Multiagent task allocation in social networks. *Autonomous Agents and Multi-Agent Systems* 25, 1 (2012), 46–86.

[8] Pierre Del Moral. 1996. Non-linear filtering: Interacting particle resolution. *Markov Processes and Related Fields* 2, 4 (1996), 555–581.

[9] Pierre Del Moral and Laurent Miclo. 2000. *Branching and Interacting Particle Systems Approximations of Feynman-Kac Formulae with Applications to Non-Linear Filtering*. Springer.

[10] Dingxiong Deng, Cyrus Shahabi, Ugur Demiryurek, and Linhong Zhu. 2016. Task selection in spatial crowdsourcing from worker's perspective. *GeoInformatica* 20, 3 (2016), 529–568.

[11] Dingxiong Deng, Cyrus Shahabi, and Linhong Zhu. 2015. Task matching and scheduling for multiple workers in spatial crowdsourcing. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, New York, NY, 21.

[12] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD'96)*. 226–231.

[13] Ina Fried. 2011. Gigwalk Allows iPhone Owners to Stumble into Part-Time Work. Retrieved November 16, 2019 from http://allthingsd.com/20110504/gigwalk-allows-iphone-owners-to-stumble-into-part-time-work/.

[14] Huiji Gao, Geoffrey Barbier, and Rebecca Goolsby. 2011. Harnessing the crowdsourcing power of social media for disaster relief. *IEEE Intelligent Systems* 26, 3 (2011), 10–14.

[15] Leisa Good. 2014. GigWalk Pays You for Doing Short Tasks with Your Phone. Retrieved November 16, 2019 from http://extracashandrewards.com/gigwalk-pays-for-phone-short-tasks/.

[16] Michael F. Goodchild and J. Alan Glennon. 2010. Crowdsourcing geographic information for disaster response: A research frontier. *International Journal of Digital Earth* 3, 3 (2010), 231–241.

[17] Neil J. Gordon, David J. Salmond, and Adrian F. M. Smith. 1993. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEE Proceedings F (Radar and Signal Processing)* 40, 2 (1993), 107–113.

[18] Ellis Hamburger. 2011. How to Make Some Extra Cash Just by Taking Photos Around Town with Your iPhone. Retrieved November 16, 2019 from http://www.businessinsider.com/gigwalk-iphone-app-2011-5

[19] Jeff Howe. 2006. The rise of crowdsourcing. *Wired Magazine* 14, 6 (2006), 1–4.
[20] Matthew Hughes. 2014. What Is Uber and Why Is It Threatening Traditional Taxi Services? Retrieved November 16, 2019 from http://www.makeuseof.com/tag/uber-threatening-traditional-taxi-services/.
[21] Jiuchuan Jiang, Bo An, Yichuan Jiang, Peng Shi, Zhan Bu, and Jie Cao. 2019. Batch allocation for tasks with overlapping skill requirements in crowdsourcing. *IEEE Transactions on Parallel and Distributed Systems* PP, 99, 1.
[22] Yichuan Jiang, Yifeng Zhou, and Wanyuan Wang. 2013. Task allocation for undependable multiagent systems in social networks. *IEEE Transactions on Parallel and Distributed Systems* 24, 8 (2013), 1671–1681.
[23] Leyla Kazemi and Cyrus Shahabi. 2012. GeoCrowd: Enabling query answering with spatial crowdsourcing. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems (SIGSPATIAL'12)*. 189–198.
[24] Seon Ho Kim, Ying Lu, Giorgos Constantinou, Cyrus Shahabi, Guanfeng Wang, and Roger Zimmermann. 2014. MediaQ: Mobile multimedia management system. In *Proceedings of the 5th ACM Multimedia Systems Conference (MMSys'14)*. 224–235.
[25] Aniket Kittur, Ed H. Chi, and Bongwon Suh. 2008. Crowdsourcing user studies with Mechanical Turk. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'08)*. 453–456.
[26] Aniket Kittur, Boris Smus, Susheel Khamkar, and Robert E. Kraut. 2011. CrowdForge: Crowdsourcing complex work. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST'11)*. 43–52.
[27] Wei Li, Haiquan Chen, Wei-Shinn Ku, and Xiao Qin. 2017. Scalable spatiotemporal crowdsourcing for smart cities based on particle filtering. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, New York, NY, 63.
[28] Rafe Needleman. 2012. EasyShift Takes on GigWalk with New Task Marketplace. Retrieved November 16, 2019 from http://www.cnet.com/news/easyshift-takes-on-gigwalk-with-new-task-marketplace/.
[29] Emin Orhan. 2012. *Bayesian Inference: Particle Filtering*. Center for Neural Science, University of Rochester, Rochester, NY.
[30] Anshul Rai, Krishna Kant Chintalapudi, Venkata N. Padmanabhan, and Rijurekha Sen. 2012. Zee: Zero-effort crowdsourcing for indoor localization. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking (Mobicom'12)*. 293–304.
[31] Hanan Samet. 1984. The quadtree and related hierarchical data structures. *ACM Computing Surveys* 16, 2 (1984), 187–260.
[32] Andreas Størksen Stordal. 2008. *Sequential Monte Carlo Methods for Bayesian Filtering*. Available at bora.uib.no/handle/1956/3374.
[33] Maja Vukovic. 2009. Crowdsourcing for enterprises. In *Proceedings of the 2009 Congress on Services (SERVICES'09)*. 686–692.
[34] Wanyuan Wang and Yichuan Jiang. 2014. Community-aware task allocation for social networked multiagent systems. *IEEE Transactions on Cybernetics* 44, 9 (2014), 1529–1543.
[35] Paul Whitla. 2009. Crowdsourcing and its application in marketing activities. *Contemporary Management Research* 5, 1 (2009), 15–28.
[36] Dingqi Yang, Daqing Zhang, Longbiao Chen, and Bingqing Qu. 2015. NationTelescope: Monitoring and visualizing large-scale collective behavior in LBSNs. *Journal of Network and Computer Applications* 55 (2015), 170–180.
[37] Dingqi Yang, Daqing Zhang, and Bingqing Qu. 2016. Participatory cultural mapping based on collective behavior data in location-based social networks. *ACM Transactions on Intelligent Systems and Technology* 7, 3 (2016), Article 30, 23 pages.
[38] Dingqi Yang, Daqing Zhang, Vincent W. Zheng, and Zhiyong Yu. 2015. Modeling user activity preference by leveraging user spatial temporal characteristics in LBSNs. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 45, 1 (2015), 129–142.
[39] Matthew Zook, Mark Graham, Taylor Shelton, and Sean Gorman. 2010. Volunteered geographic information and crowdsourcing disaster relief: A case study of the Haitian earthquake. *World Medical & Health Policy* 2, 2 (2010), 7–33.

Received December 2018; revised July 2019; accepted September 2019