Optimizing Task Throughput for Large-scale Mobile Crowdsourcing in Smart Cities

by

Wei Li

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 7, 2021

Keywords: Mobile Crowdsourcing, Particle Filtering, Quadtree

Approved by

Wei-shinn Ku, Chair, Professor of Computer Science and Software Engineering
Xiao Qin, Co-chair, Professor of Computer Science and Software Engineering
Saad Biaz, Professor of Computer Science and Software Engineering
Alvin Lim, Professor of Computer Science and Software Engineering

Abstract

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 dissertation, 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 in order to maximize the total number of tasks that can be completed for an entire worker group while taking into account various spatial and temporal constraints, such as task execution duration, task expiration time, and worker/task geographic locations. Since the exact solution to the GTS problem is computationally intractable, we first propose two sub-optimal approaches (LCPF and NUD-IC) based on particle filtering and DBSCAN for the Single Worker Geo-Task Scheduling (SGTS) problem. We then extend our work to solve the Multi-Worker Geo-Task scheduling (MGTS) problem by proposing two space partitioning-based methods (QT-NNH and QT-NUD), which leverage the point-region quadtree to ensure workload balancing. We further propose WBT-NNH and WBT-NUD, which build on the algorithms QT-NNH and QT-NUD respectively, and provide more effective and dynamic workload balancing among all workers using the proposed Workload-balancing Bisection Tree (WBT) comparing to QT-NNH and QT-NUD. The effectiveness and efficiency of the six proposed approximate solutions are verified by our extensive experiments using both real and synthetic data. Compared with 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. Last but not least, we present a web-based demo application for Turbo-GTS featuring the aforementioned four MGTS algorithms, which includes an interactive interface for users to load the current task/worker distributions and compare the task assignment of each worker returned by different algorithms in a real-time fashion. We also demonstrate the front-end interface of Turbo-GTS demo with several exploratory use cases in New York City.

Acknowledgments

I am extremely indebted to my advisors, Dr. Wei-Shinn Ku and Dr. Xiao Qin for their great efforts, trust and support in my work. They have always provided me with in-depth and insightful guidance, advice and suggestions on research methodologies, which directed my way and gave me power when I ever felt puzzled and frustrated in study. Their inexhaustible enthusiasm and meticulous attitude in research also keep inspiring and driving me to overcome obstacles and challenges throughout all these years as a doctoral student.

I owe my deepest gratitude to Dr. Haiquan Chen, whose extensive experience and strong understanding in mathematics, analysis and statistics gave me tremendous help in my research. His excellent writing skills also contributed a lot in my publications.

I would like to express my sincere thanks to committee members, Dr. Alvin Lim, Dr. Saad Biaz, and university reader Dr. Shiwen Mao, who spent their time in reviewing my dissertation and attending my defense. I also want to thank Rodrigo Sardinas for his time and efforts in building an excellent experimental environment for me.

Finally, the endless love, support and encouragement from my family is the source of my strength that keeps me moving on in both my work and life, especially my wife Chang Fang.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The ubiquity of mobile platforms and smart phones breeds a large number of mobile crowdsourcing applications like TaskRabbit (a mobile marketplace allowing users to outsource small tasks to workers in their neighborhood [5]), Uber (a mobile application allowing passengers with smartphones to submit trip requests which are then routed to willing drivers [20]), Gigwalk (a crowdsourcing service that helps businesses to 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 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 1.1 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 one hour to finish while all the other four tasks only need one 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 to get more tasks completed.

Limitations of the Existing Approaches. We make the following troubling observations on the existing research for mobile crowdsourcing.

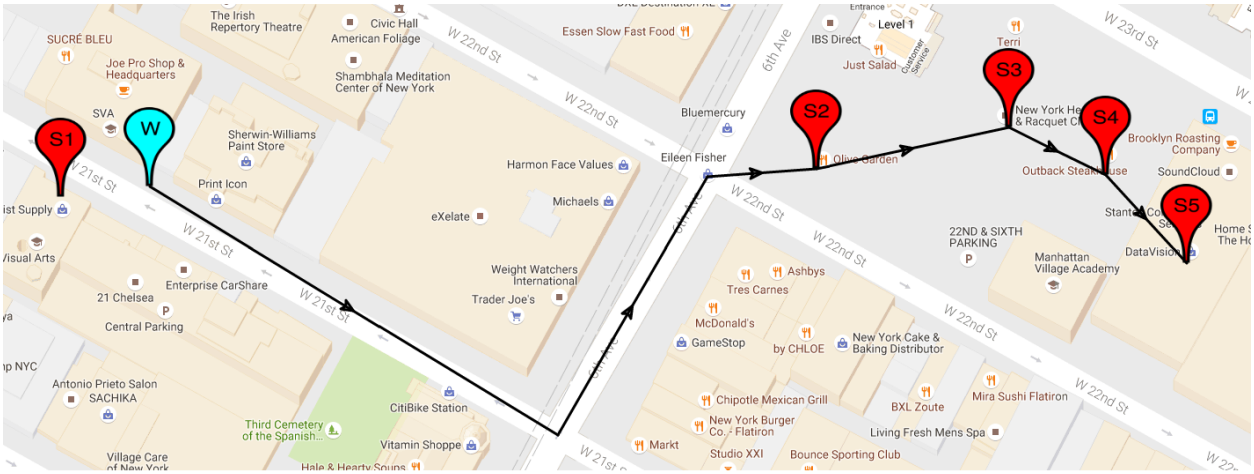


Figure 1.1: An example of spatial task scheduling in Manhattan Island, New York City. w represents the current location of a worker; s_1 represents a task near w but with longer execution duration, whereas s_2, s_3, s_4, and s_5 represent four other tasks located farther from w but with shorter execution duration. The arrow represents the optimal task execution route for w.

- 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 Chapter 7, Chapter 8 and Chapter 9), existing scheduling solutions [10] [23] [11] fail to find optimal/sub-optimal task assignment when coping with massive tasks distributed in large geographic areas. Table 1.1 and Table 1.2 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 [40] [39] [38]. As depicted in Table 1.1 and Table 1.2, 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] [23] [11] suffer from inefficiency in terms of generating task assignment sequence for each worker when handling massive tasks distributed in large geographic areas. As depicted in Table 1.1 and Table 1.2, 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 while BLALS-T [11] needed 11 minutes to assign and schedule 12048 tasks among 300 workers.

- Failure to take into account important spatial and temporal constraints. Existing scheduling solutions [10] [23] [11] 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 one to two hours [13]. A preferred scheduling solution should be able to consider important spatiotemporal characteristics of real-world mobile crowdsourcing applications.

Table 1.1: **Performance of the state-of-the-art single worker task scheduling solution (BSH [10]) in mobile crowdsoucing**

Total number of tasks	3000	4000	5000	6000	7000
CPU cost (hours)	0.8	1.3	5.5	7	7.5
Number of accomplished tasks	84	94	104	108	113

Table 1.2: **Performance of the state-of-the-art multi-worker task scheduling solution (BLALS-T [11]) in mobile crowdsoucing on** 300 **workers**

Total number of tasks	10000	15000	20000	25000	30000
CPU cost (minutes)	< 1	2	4	8	11
Number of accomplished tasks	7483	8879	9861	10926	12048

Our Goal and Contributions. In this dissertation, 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 in order 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 dissertation 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, LCPF and NUD-IC, for the Single Worker Geo-Task Scheduling (SGTS) problem. LCPF solves the problem by incorporating particle filtering while NUD-IC generates the sub-optimal schedule by integrating particle filtering with DBSCAN clustering based on Non-Urgency Degree (NUD).

- Inspired by the proposed LCPF and NUD-IC for the Single Worker Geo-Task Scheduling (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 the point-region quadtree to ensure workload balancing among multiple workers.

- To provide more effective and dynamic workload balancing among all the workers, we propose a novel tree structure, Workload-balancing Bisection Tree (WBT), and further present two more advanced scheduling algorithms WBT-NNH and WBT-NUD to solve the Multi-Worker Geo-Task scheduling (MGTS) problem. WBT-NNH and WBT-NUD build on algorithms QT-NNH and QT-NUD but utilize WBT to perform workload balancing instead of quadtree.

- The effectiveness and efficiency of the six proposed approximate solutions were verified by our extensive experiments using the synthetic data and Foursquare [40] [39] [38]

mobile user check-in data in three highly populated cities (i.e., New York City; Austin, Texas; Tokyo). Compared with the state-of-the-art approaches, all 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.

- We present a web-based demo application for Turbo-GTS featuring the aforementioned four MGTS algorithms, which includes an interactive interface for users to load the current task/worker distributions and compare the task assignment of each worker returned by different algorithms in a real-time fashion. We also demonstrate the front-end interface of Turbo-GTS demo with several exploratory use cases in New York City.

1.1 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 describes the related work on task scheduling in mobile crowdsourcing. Chapter 3 formulates our studied problem and presents the system architecture of Turbo-GTS. In Chapter 4, we elaborate on LCPF and NUD-IC algorithms for single worker task scheduling. QT-NNH and QT-NUD are introduced to solve multiple worker task scheduling in Chapter 5. Workload-balancing Bisection Tree (WBT) along with the two algorithms WBT-NNH and WBT-NUD are presented in Chapter 6 to provide better workload balancing for multiple worker task scheduling. Experimental evaluation is presented in Chapter 7, Chapter 8 and Chapter 9. Exploratory use cases of Turbo-GTS demo is demonstrated in Chapter 10. Chapter 11 concludes the dissertation and presents our future work.

Chapter 2

Related Work

2.1 Mobile Crowdsourcing

Crowdsourcing enables businesses to utilize the spare time of a crowd of people to do jobs which 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 to 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] [32] [26] [4]. The commercial prospects of crowdsourcing have also been discussed in [35] [37] . Beyond the benefits from the perspective of enterprises, crowdsourcing even has the potential to be used for governments to deal with disasters [14] [16] [41]. .

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, namely, reward-based and self-driven based crowdsourcing. In the reward-based mobile crowdsourcing, workers have reward benefits for correctly performed tasks. For example, EasyShift [30] - a smartphone application similar to Gigwalk - pays a user three to eight dollars 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 the 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 the volunteers participating in this campaign are self-motivated users.

2.2 Geo-Task Scheduling

Due to the fact that traditional task scheduling fails to take spatial data into account, solutions from [36] [7] [22] [21] can not 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 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, Quad-tree, 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 *et al.* proposed three strategies [23] 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 *et al.* further improved Kazemi's model to dedicatedly address the assignment problem of special tasks that can only be performed by workers with certain expertise in reward-based mobile crowdsourcing [6]. The problem with their solutions is that the travel cost calculated based on their model can be very inaccurate since 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 [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 Single Worker Geo-Task Scheduling (SGTS) problem. The most closely related work to this dissertation is [11], which attempted to solve the Multi-worker Geo-Task Scheduling (MGTS) problem by maximizing the total number of tasks that can be accomplished by all the workers. However, their work has two limitations. First, they assume that each worker has a maximum number of tasks that can be assigned to her. 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.

Chapter 3

Problem Formulation

Definition A spatial task s is a crowdsourced task located in a spatial region Z, with three innate attributes: its **location** s in Z (in this dissertation, we use the same symbol s to represent the location as well as the task itself for convenience, since 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 dissertation, 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 A task set $S = \{s_1, s_2, \cdots, s_n\}$ is a collection of spatial tasks in Z waiting to be accomplished, in which each task can be accomplished once and only once.

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 dissertation. We also assume that a worker will lose no time to head for the next one once the current task has been finished.

Definition 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.

Table 3.1: **Symbolic Notations**

Symbol	Meaning
S	the whole task set
W	the worker set
w	a worker in W
Z	the spatial region where W and S are located
s	a task in S, representing its location in Z
e_s	the expiration time of s
u_s	the execution duration of s
s_0	the imaginary task representing where w starts in Z
t_0	the time when w starts from s_0
c_{s_i,s_j}	the time cost in traveling between s_i and s_j
t_s	the completion time of s
$ptcl\_no$	number of particles in particle filtering
Q	a priority queue of $ptcl\_no$ task sequence samples
R_i	a task sequence sample in Q
G_i	a task group generated by classifying each task's duration
n_{div}	the number of G_i to divide S into
ddr	a given decimal used to calculate n_{div}
R_{final}	the task sequence returned by an algorithm

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 the 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 A **task sequence** $R_{final} = \{s_{r_1}, s_{r_2}, \cdots, s_{r_m}\}$ $(1 \leq r_k \leq n, 1 \leq k \leq m, 1 \leq m \leq n)$ 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 The **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.

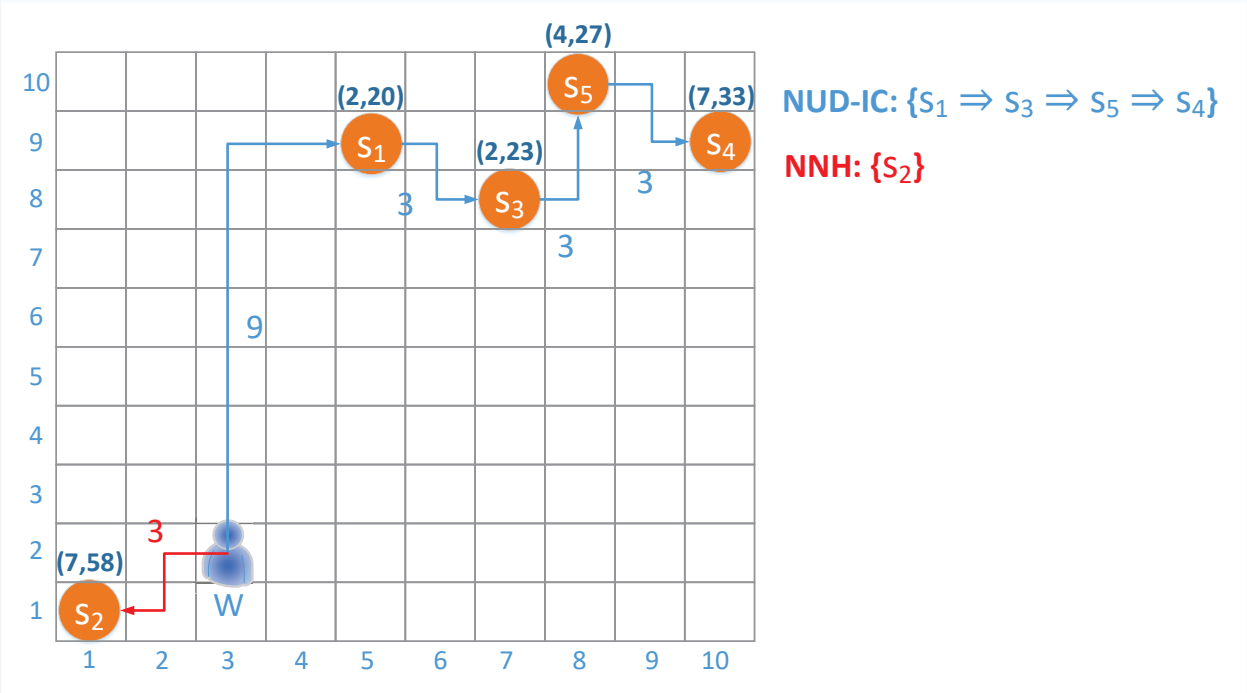


Figure 3.1: Here, w represents the location where the worker starts at time 0. s_{1-5} represent all the tasks at their respective locations. Parenthesized number pairs on top of each task represent this task's execution duration and expiration time with the same time unit as the scales in the grid, where distances are calculated in taxicab geometry.

Figure 3.1 shows an example of the SGTS problem. In Figure 3.1, w indicates her 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 the 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 Chapter 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 The **Multi-worker Geo-Task Scheduling** (MGTS) is designed to find a task scheduling plan for all the 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.

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, MGTS problem has to be solved as a whole to achieve the maximum number of accomplished tasks by all the workers.

Figure 3.2 shows the system architecture of Turbo-GTS. We proposed two algorithms, LCPF and NUD-IC (we will compare them in detail later in Chapter 4), for the SGTS problem and four algorithms, QT-NNH, QT-NUD, WBT-NNH and WBT-NUD (QT-NNH and QT-NUD will be compared in detail later in Chapter 5 while WBT-NNH and WBT-NUD will be discussed in detail later in Chapter 6), 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 in order to maximize the total number of tasks that can be completed by all the workers.

Table 3.1 summarizes the notations used in the problem formulation and the following algorithms.

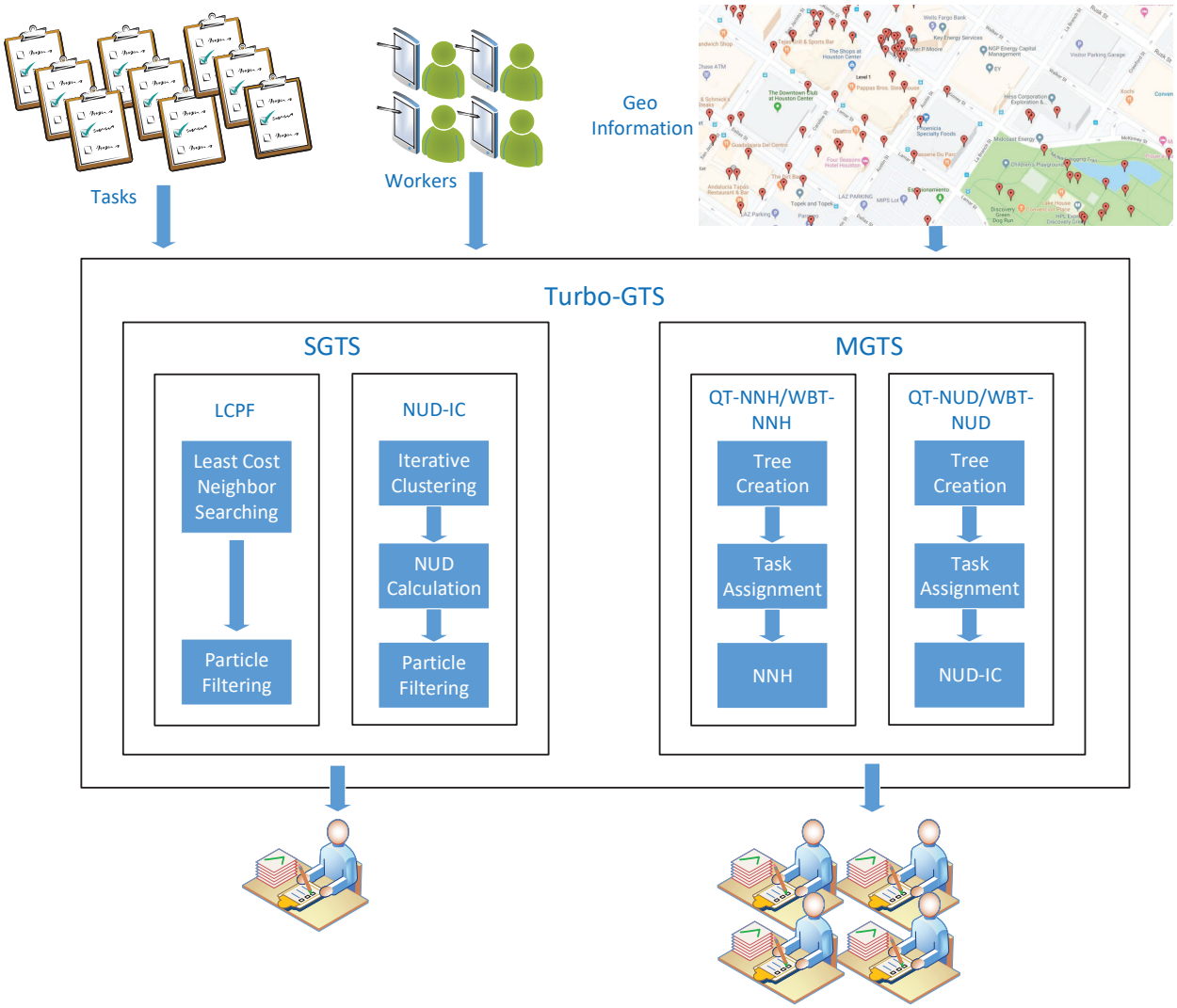


Figure 3.2: System Architecture of Turbo-GTS

Chapter 4

Single-worker Geo-Task Scheduling

In this chapter, we present two approximate solutions to solve the SGTS problem.

4.1 The Least Cost Neighbor with Particle Filtering (LCPF)

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 [31]. It was first proposed to deal with recursive Bayesian estimation [17], then it is 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

$$x_k = f_k(x_{k-1}, v_{k-1}) \tag{4.1}$$

In Equation 4.1, 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

$$z_k = h_k(x_k, n_k) \tag{4.2}$$

In Equation 4.2, h_k is a nonlinear function and n_k a measurement noise sequence. Given the settings above, 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 samples or particles are chosen from $\{X_n\}$ and each sample will be assigned an importance weight through importance sampling [34]. 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 followd 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 formulae:

$$x_k^i \sim p(x_k|x_{k-1}^i) \tag{4.3}$$

$$w_k^i \propto w_{k-1}^i p(z_k|x_k^i) \tag{4.4}$$

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 below:

Definition 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 its potential successors in S.

LCPF is detailed in Algorithm 1. Before 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 - 21 in 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

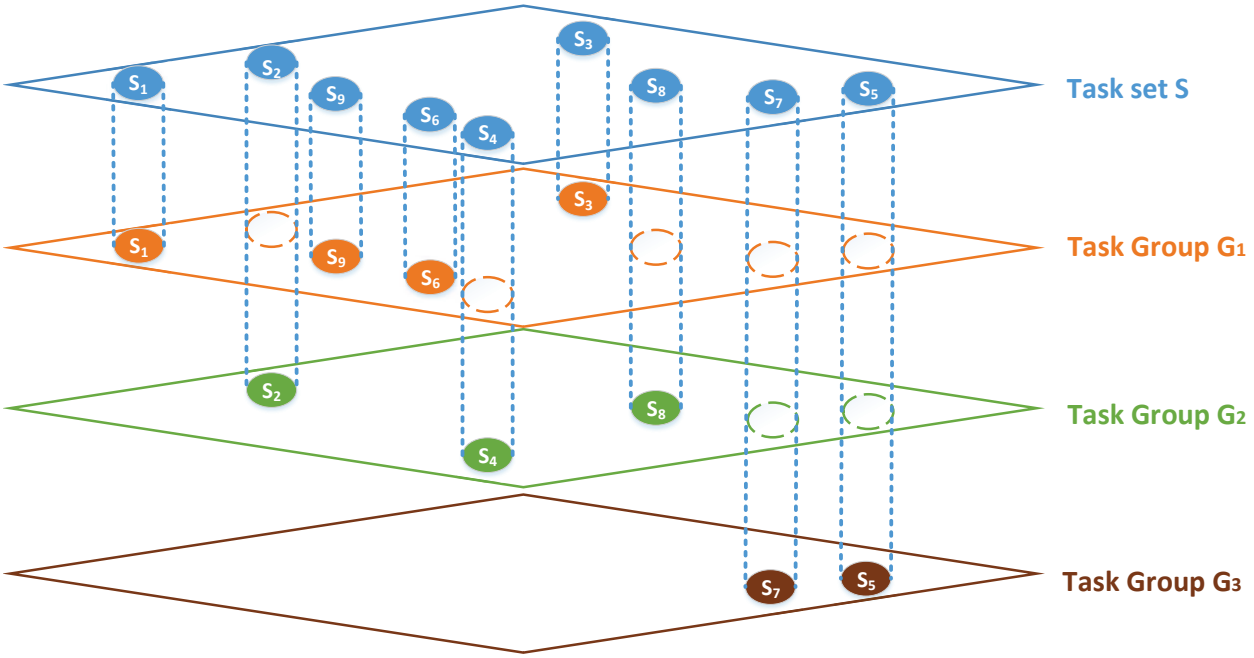


Figure 4.1: An illustration of dividing a task set of nine tasks into three task groups through the duration categorization process.

of the final result (line 4 in Algorithm 1), and Q^* is created as a duplicate of Q (lines 5 - 6 in Algorithm 1) before clearing out all the elements in Q for the upcoming new samples (line 7 in Algorithm 1). Lines 8 - 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 - 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 - 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 (NUD-IC)

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 Non-Urgency

Degree (NUD) that is about 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 *non-urgency degree* (NUD) of s for each task $s_i \in S$ at time t_s can be defined as the following:

$$NUD(s_i, s, t_s) = \frac{e_{s_i} - u_{s_i} - c_{s,s_i} - t_s}{u_{s_i}} \tag{4.5}$$

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 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 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 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 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

Algorithm 1 LCPF(S, s_0, t_0, *ptcl_no*)

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.
Q.insertWithPriority($\{s_0\}, t_0^{-1}$) ▷ Reciprocal of completion time as priority.
while Q.isNotEmpty() **do**
    $R_{final} \leftarrow Q$.getHighestPriority() ▷ Element with highest priority is returned but not removed.
    Create an empty priority queue Q^*.
    Copy all the elements of Q into Q^*.
    Q.removeAllElements()
    while Q^*.isNotEmpty() **do**
        $R \leftarrow Q^*$.pullHighestPriority() ▷ Element with highest priority is returned and removed.
        $s_{end} \leftarrow$ the last task in R
        $t_{end} \leftarrow$ the completion time of R
        $S_{nb} \leftarrow$ at most *ptcl_no* least cost neighbors of s_{end} at t_{end} from S.
        for all $s_j \in S_{nb}$ **do**
            $R_j^* \leftarrow R \cup \{s_j\}$
            Q.insertWithPriority($R_j^*, t_{s_j}^{-1}$) ▷ Reciprocal of completion time as priority.
            if $|Q| >$ *ptcl_no* **then**
                Q.removeLowestPriority() ▷ Element with lowest priority is removed.
            end if
        end for
    end while
end while
return R_{final}

should be divided into according to each task's execution duration, and we call this process 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 subintervals (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} subintervals: $I_1 = [u_{min}, u_{min} + \Delta u]$, $I_2 = (u_{min}+\Delta u, u_{min}+2*\Delta u]$, $\ldots$, $I_{n_{div}-1} = (u_{min}+(n_{div}-2)*\Delta u, u_{min}+(n_{div}-1)*\Delta u]$, $I_{n_{div}} = (u_{min} + (n_{div} - 1) * \Delta u, u_{max}]$; lastly, we put tasks with their respective execution duration falling into the same subinterval I_i into the same task group G_i ($1 \leq i \leq n_{div}$), and thus divide the whole task set S into n_{div} task groups $G_1, \ldots, G_{n_{div}}$. Figure 4.1 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 DBSCAN routine and the outliers thus extracted from G_1 will be added into G_2, which will be processed with DBSCAN algorithm too with its outliers being added into G_3 and so forth. When the last task group $G_{n_{div}}$ has been processed by 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 2 NUD_SUM(S, R)

```
1: sum ← 0
2: s_end ← the last task in R
3: t_end ← the completion time of R
4: for all s_i ∈ S \ R do
5:     if s_i is a potential successor of s_end at t_end then
6:         sum ← sum + NUD(s_i, s_end, t_end) ▷ Definition of NUD is shown in Equation 4.5.
7:     end if
8: end for
9: return sum
```

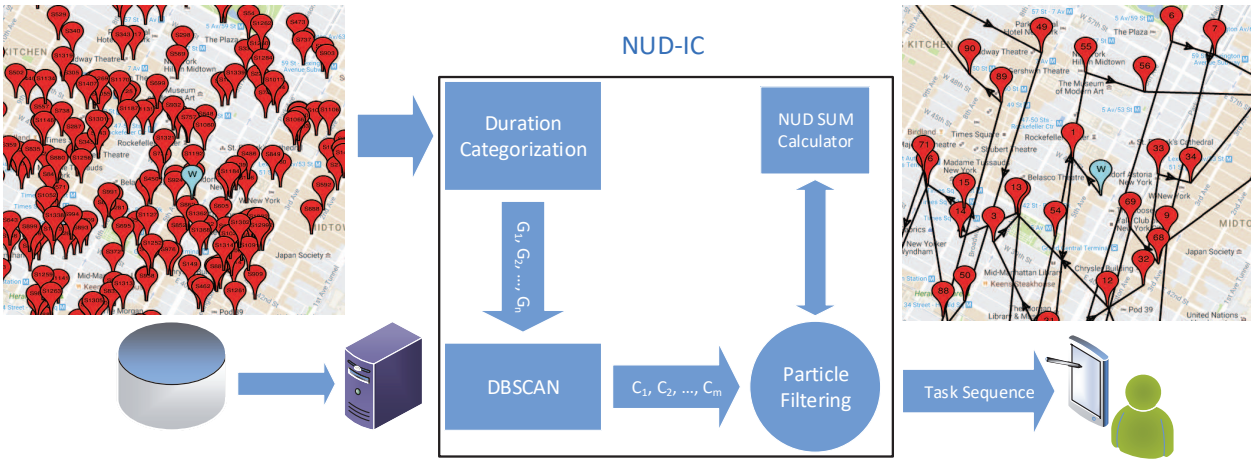


Figure 4.2: Illustration of the NUD-IC algorithm.

Algorithm 3 NUDPF($S, s_0, t_0, ptcl\_no$)

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 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**
    $R_{final} \leftarrow Q$.getHighestPriority() ▷ Element with highest priority is returned but not removed.
    Create an empty priority queue Q^*.
    Copy all the elements of Q into Q^*.
    Q.removeAllElements()
    while Q^*.isNotEmpty() **do**
        $R \leftarrow Q^*$.pullHighestPriority() ▷ Element with highest priority is returned and removed.
        $s_{end} \leftarrow$ the end task of R
        $t_{end} \leftarrow$ the end time of R
        for all $s_j \in S \setminus R$ **do**
            if s_j is a potential successor of s_{end} at t_{end} **then**
                $R^*_j \leftarrow R \cup \{s_j\}$
                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}

Algorithm 4 IterativeCluster($S, \epsilon, MinPts, ddr$)

$n_{div} = \lceil 1/ddr \rceil$
Divide S into $G_1, \ldots, G_{n_{div}}$ by duration categorization.
for all $G_i \in \{G_1, \ldots, G_{n_{div}}\}$ **do**
    DBSCAN($G_i, \epsilon, MinPts$)
    if $i < n_{div}$ **then**
        Put outliers of G_i into G_{i+1}.
    end if
end for
return all the clusters and outliers.

Algorithm 5 NUD-IC(S, s_0, t_0, $ptcl\_no$, ϵ, $MinPts$, ddr)

$R_{final} \leftarrow \emptyset$
$(clusters, outliers) \leftarrow$ IterativeCluster($S, \epsilon, MinPts, ddr$)
$s_{cur} \leftarrow s_0$
$t_{cur} \leftarrow t_0$
for all $clst_i \in clusters$ **do**
    $R_{final} \leftarrow R_{final} \cup$ NUDPF($clst_i, s_{cur}, t_{cur}, ptcl\_no$)
    $s_{cur} \leftarrow$ the last task in R_{final}
    $t_{cur} \leftarrow$ the completion time of R_{final}
end for
$R_{final} \leftarrow R_{final} \cup$ NUDPF($outliers, s_{cur}, t_{cur}, ptcl\_no$)
return R_{final}

The algorithmic description and the block diagram of the Non-Urgency Degree Particle Filtering with Iterative Clustering (NUD-IC) are shown in Algorithm 5 and Figure 4.2, 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 with 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

$$TSL * k * top\_k\_select() = O(n) * k * O(k * n) = O(k^2 * n^2), \tag{4.6}$$

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 subroutines, *IterativeCluster* and *NUDPF* (which dominates the whole algorithm). Thus, the time complexity of NUD-IC can be represented as

$$\begin{aligned} \sum_i (TSL_{C_i} * k * NUD()) &= \sum_i (O(|C_i|) * k * O(|C_i|^2)) \\ &= O(k * \sum_i |C_i|^3) = O(k * (\sum_i |C_i|)^3) = O(k * n^3), \end{aligned} \tag{4.7}$$

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

$$Search\_Tree\_Height * N_{st} * NNH() = O(n) * O(n) * O(n^2) = O(n^4), \tag{4.8}$$

where N_{st} denotes the maximum number of tree nodes in each level in BSH [10] while $NNH()$ represents the cost of the NNH [10] approach which is used as lower bound computation in BSH [10]. Table 4.1 compares the characteristics of each algorithm under different application scenarios, where more check marks indicate better performance.

Table 4.1: **Performance Comparison of ICPF and NUD-IC with Baselines**

Methods	**Optimizing Running Time**	**Optimizing # 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	✓	✓✓✓	✓	$O(k * n^3)$

An Illustrative Example. Figure 3.1 compares NUD-IC with 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 the 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 while s_2 and s_4 will be detected as outliers by IterativeCluster (as shown in Algorithm 4). Next, the subroutine 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 $\text{NUD\_SUM}(S, w \rightarrow s1) = \text{NUD}(s_3, s_1, 11) + \text{NUD}(s_5, s_1, 11) = \frac{23-2-3-11}{2} + \frac{27-4-4-11}{4} = 3.5 + 2 = 5$. 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, $\text{NUD\_SUM}(S, w \rightarrow s_3) = \text{NUD}(s_1, s_3, 12) + \text{NUD}(s_5, s_3, 12) = \frac{20-2-3-12}{2} + \frac{27-4-3-12}{4} = 1.5 + 2 = 3.5$, s_5's NUD_SUM value, $\text{NUD\_SUM}(S, w \rightarrow s_5) = \text{NUD}(s_3, s_5, 17) = \frac{23-2-3-17}{2} = 0.5$. 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 her first task for completion. The aforementioned selection procedure repeats until all the tasks in each cluster have been assigned first or already expired. Once all the 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 3.1, 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.

Chapter 5

Multi-worker Geo-Task Scheduling based on Quadtree

In this chapter, we elaborate on our solutions for multi-worker geo-task scheduling problem. The major challenge in multi-worker geo-task scheduling is how to divide the whole task set into spatial subsets 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 multi-worker geo-task scheduling, inspired by the proposed LCPF and NUD-IC for single worker geo-task scheduling problem. QT-NNH and QT-NUD leverage the point-region quadtree to ensure workload balancing among multiple workers.

5.1 Space Partitioning using Point-Region (PR) Quadtree

Turbo-GTS employs Point-Region (PR) quadtree [33] 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 that 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 5.1 (a) illustrates how a spatial region with six workers and nine tasks can be partitioned using the PR quadtree. The corresponding tree structure is shown in Figure 5.1 (b). Figure 5.1 (c) depicts the structure of a PR quadtree node. A PR quadtree node consists of three parts: the region which the tree node represents, the workers who will be assigned to this region, and all the tasks located in the region. As illustrated in Figure 5.1 (a) and Figure 5.1 (b), at first, a PR quadtree root is created to represent the whole region A containing all the workers and tasks. Next, because region A contains more

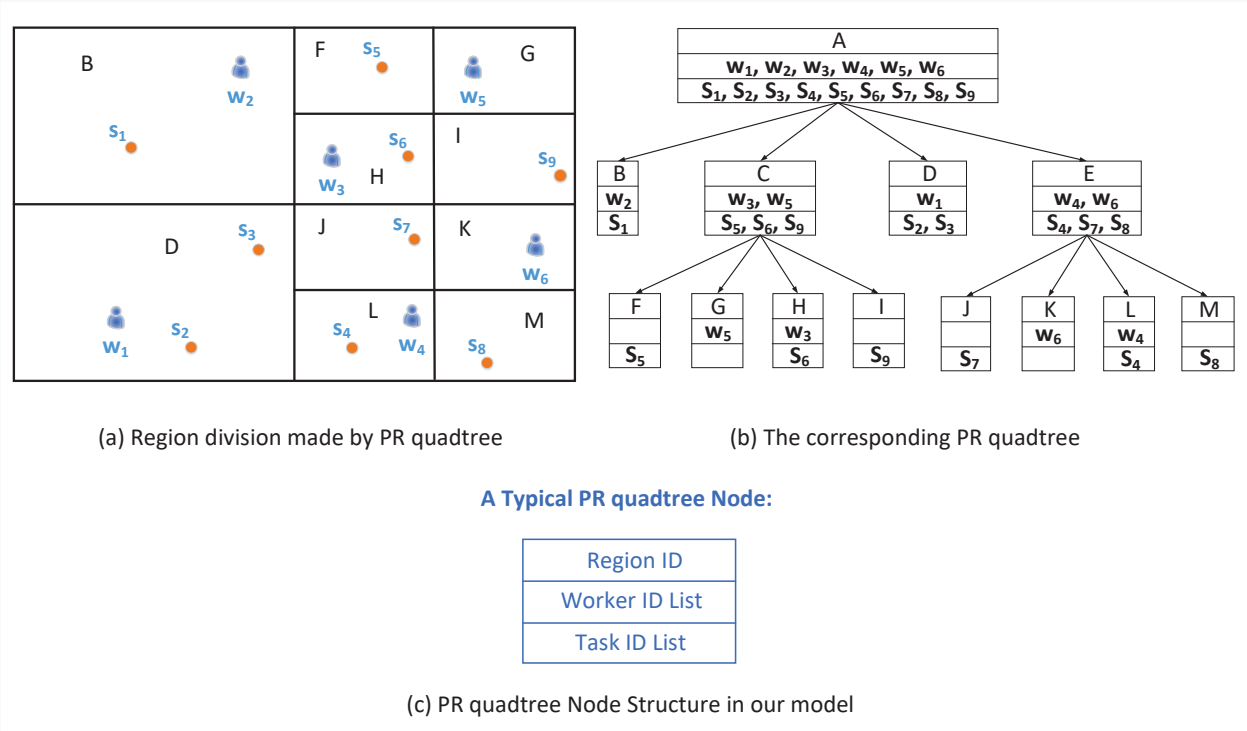


Figure 5.1: An illustration of how a region A is partitioned using a PR quadtree. w_{1-6} represent workers and s_{1-9} represent tasks.

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 region B and region D each 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 the leaf nodes contain no more than one worker.

Algorithm 6 QT-NNH(Z, W, S)

1: Create an empty PR quadtree qt.
2: $qt.root \leftarrow \{Z, W, S\}$.
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 all chld_i ∈ {chld_1, ..., chld_4} do
4:         TopDownDivideNNH(chld_i)
5:     end for
6:     for all chld_i ∈ {chld_1, ..., chld_4} do
7:         wk_set ← ∅
8:         for all 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 Algorithm 6 and Algorithm 8, respectively. Both QT-NNH and QT-NUD are based on PR quadtree discussed in the last 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 - 2 in Algorithm 6). Such 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 - 5 in Algorithm 7). In this way, a PR quadtree is constructed recursively. Once the PR quadtree is created, every leaf node will be checked to determine whether there is one worker associated with it (line 18 in Algorithm 7). If yes, NNH [10] is invoked for the current leaf node (lines 19 - 20 in Algorithm 7) to identify tasks for the associated worker, i.e, we assign each worker to the tasks in the same node first. After all the 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 - 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 that 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($Z, W, S, pctl\_no$)

1: Create an empty PR quadtree qt.
2: $qt.root \leftarrow \{Z, W, S\}$.
3: TopDownDivideNUD($qt.root, pctl\_no$)

Algorithm 9 TopDownDivideNUD($qt\_node, pctl\_no$)

1: **if** qt_node.NumberOfWorkers > 1 **then**
2: $\{chld_1, \ldots, chld_4\} \leftarrow qt\_node$.split()
3: **for all** $chld_i \in \{chld_1, \ldots, chld_4\}$ **do**
4: TopDownDivideNUD($chld_i, pctl\_no$)
5: **end for**
6: **for all** $chld_i \in \{chld_1, \ldots, chld_4\}$ **do**
7: $wk\_set \leftarrow \emptyset$
8: **for all** $chld_j \in \{chld_1, \ldots, chld_4\}$ and $chld_j \neq chld_i$ **do**
9: $wk\_set \leftarrow wk\_set \cup chld_j$.getWorkers()
10: **end for**
11: **while** $wk\_set \neq \emptyset$ **do**
12: $w^* \leftarrow$ argmin$\{w_i$.getCurrentTime() + travel_cost$(w_i \rightarrow chld_i) \mid w_i \in wk_set\}$
13: NUDPF($chld_i$.getTasks(),w^*.getCurrentLocation(),w^*.getCurrentTime(),$pctl\_no$)
14: $wk\_set \leftarrow wk\_set \setminus \{w^*\}$
15: **end while**
16: **end for**
17: **else**
18: **if** qt_node.NumberOfWorkers $= 1$ **then**
19: $w \leftarrow qt\_node$.getWorkers()
20: NUDPF(qt_node.getTasks(),w.getCurrentLocation(),w.getCurrentTime(),$pctl\_no$)
21: **end if**
22: **end if**

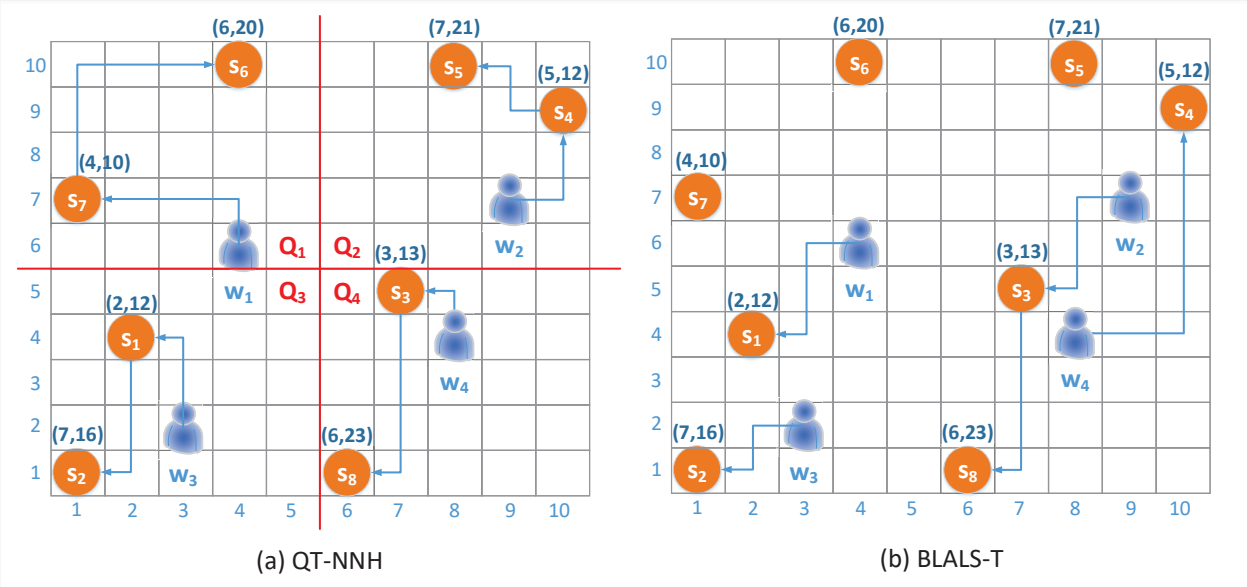


Figure 5.2: A comparison of the results when QT-NNH and BLALS-T are applied respectively to the same scenario, where w_{1-4} represent workers and s_{1-8} represent tasks. Parenthesized number pairs on top of each task represent this task's execution duration and expiration time with the same time unit as the scales in the grid, where distances are calculated in taxicab geometry.

5.3 Algorithmic Comparison

In this section, we first compare our proposed methods with 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 that 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

Table 5.1: **Performance Comparison of QT-NNH and QT-NUD with Baselines**

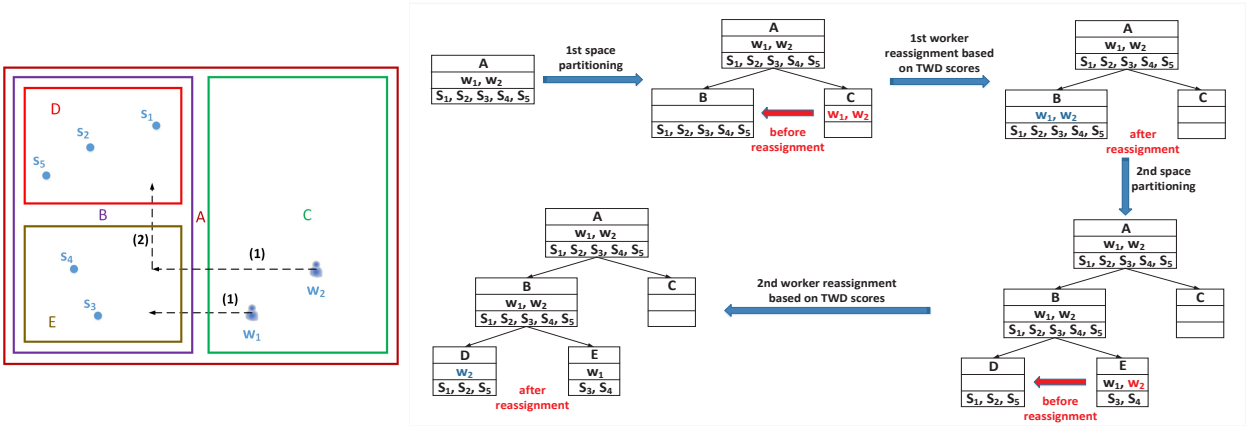
Methods	Optimizing Running Time	Optimizing # of Accomplished Tasks	Detecting Outliers	Time Complexity
BLALS-T [11]	×	×	×	$\Omega((n_w + n_t)^2 * n_w * n_t)$
QT-NNH	✓✓	✓	×	$O(n_t^2)$
QT-NUD	✓	✓✓	✓	$O(k * n_t^3)$

$\sum_i [H_{partition}(i) + H_{merge}(i)] + H_r = \Omega(H_{merge}(1)) = \Omega((n_w + n_t)^2 * n_w * n_t)$, 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 the remaining tasks when all the recursive top-down bisection partitioning and bottom-up merging are finished. Table 5.1 compares the characteristics of each algorithm under different application scenarios, where more check marks indicate better performance.

An Illustrative Example. Figure 5.2 compares QT-NNH with BLALS-T [11] as an illustrative example. As shown in Figure 5.2 (a), QT-NNH at first employs the proposed quadtree to divide the whole target spatial region into four subregions (i.e., four tree leaf nodes), let's 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 the 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 5.2 (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 which 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 5.2 (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) WBT space partitioning (b) WBT expansion and worker reassignment

Figure 6.1: An illustration of the space partitioning using the proposed workload-balancing bisection tree, which results in two occurrences of worker reassignments (highlighted as red arrows) based on the estimated task-worker-density scores for tree nodes.

Chapter 6

Multi-worker Geo-Task Scheduling based on Workload-balancing Bisection Tree (WBT)

In this chapter, we propose two new space partitioning-based methods, WBT-NNH and WBT-NUD, which buid on the two algorithms QT-NNH and QT-NUD proposed in [28] respectively, to ensure a more effective and dynamic workload balancing among multiple workers by leveraging a novel tree structure called Workload-balancing Bisection Tree (WBT).

6.1 Workload-balancing Bisection Tree (WBT)

6.1.1 General Description

Similar to point-region quadtree [33] that recursively divides a region into four equal quadrants until there is at most one data point in the region, WBT divides a rectangle region recursively into two equal parts transversely as long as this region contains more than one worker. The differences between WBT and point-region quadtree reside in the following

three aspects. First, a region will only be partitioned when it contains more than one worker, whereas the number of tasks within has no impact on the behavior of partitioning. Second, WBT divides a region into two equal parts instead of four, and the division is always made transversely. Third, immediately after a partition is made on a region, a workload balancing procedure will be triggered to balance the number of workers between the two partitions.

6.1.2 Task Worker Density (TWD)

Workload balancing in WBT is performed based on the calculation of the task-worker-density of each tree node, which can be estimated using Equation 6.1, where N represents a WBT node, w and s represent a worker and a task in N respectively, e_s and u_s represent the expiration time and execution duration of s respectively, and t_w represents the current time of w. The workload balancing in WBT is performed based on the calculation of Task-Worker-Density (TWD) of each tree node, which can be estimated using Equation 6.1. The TWD of each tree node is to estimate the likelihood that the workers currently in that node are able to finish all the tasks currently in the same node, assuming the locations of all the tasks are uniformly distributed. In Equation 6.1, $RPD(N)$ is the Random Point Distance (RPD), which is the average distance of the two uniformly distributed random points in a rectangular region and can be calculated by Equation 6.2 [29].

$$task\_worker\_density(N) = \frac{\sum_{s_i \in N} \frac{1}{u_{s_i}}}{\sum_{w_i \in N} \frac{max\{e_{s_i} | s_i \in N\} - t_{w_i}}{RPD(N)}} \tag{6.1}$$

$$\begin{aligned} RPD(a,b) = \frac{1}{15}\Bigg(\frac{a^3}{b^2} + \frac{b^3}{a^2} + d * \Big(3 - \frac{a^2}{b^2} - \frac{b^2}{a^2}\Big) + \\ \frac{5}{2} * \Big(\frac{b^2}{a} * \ln\Big(\frac{a+d}{b}\Big) + \frac{a^2}{b} * \ln\Big(\frac{b+d}{a}\Big)\Big)\Bigg). \end{aligned} \tag{6.2}$$

6.1.3 Dynamic Workload Balancing

In WBT, node splitting is based on the bisection tree while worker reassignment is conducted to ensure all the leaf nodes reach a similar TWD score for achieving load-balancing. The whole procedure of the dynamic workload balancing on each pair of node siblings is shown in Algorithm 10. After figuring out which node has a greater workload between a pair of siblings in lines 1 - 11, workers keep being reassigned from the sibling with a lesser workload to the sibling with a greater workload in lines 12 - 28. As we want to make sure the chosen worker to be reassigned still has enough time to do something in her new working area after moving, each time the worker with the earliest arrival time in the destination partition is selected (line 13). If the workload relation between the two sibling nodes still holds after reassigning the chosen worker with her current time updated by adding the traveling cost between her and the destination partition, the reassignment will be committed and the information of the two nodes will also be updated accordingly (line 15 - 20); otherwise, the reassignment will be rolled back with the worker's current time being reverted (line 22) and the whole procedure quits (line 23). The procedure also quits when the number of workers is no longer less than the number of tasks in the node with a greater workload (line 12), or when there is no worker left in the node with a lesser workload (line 26).

Figure 6.1 illustrates how a spatial region with two workers and five tasks can be partitioned following Algorithm 10. As shown in Figure 6.1 (b), each WBT tree node consists of three parts: the region which the tree node represents, the workers belonging to that region, and the tasks belonging to that region. First the whole region A, as illustrated in Figure 6.1 (a), is divided into two sub-regions B and C, with all workers located in region C and all tasks located in B. By calculating the task-worker-density scores of B and C, the workers will be reassigned to B from C. Next, the region B is divided into D and E. Similarly, by checking the task-worker-density scores of D and E, w_2 in E is reassigned to D. As a result, the final WBT tree will have three leaf nodes, C, D, and E, with C having no workers or tasks, D having one worker and three tasks, and E having one worker and two tasks.

Algorithm 10 WorkloadBalance(N_1, N_2)

if TWD(N_1) > TWD(N_2) **then**
    $N_{big} \leftarrow N_1$
    $N_{sml} \leftarrow N_2$
else
    if TWD(N_1) < TWD(N_2) **then**
        $N_{big} \leftarrow N_2$
        $N_{sml} \leftarrow N_1$
    else
        return
    end if
end if
while workers are less than tasks in N_{big} **do**
    $w \leftarrow \text{argmin}\{t_{w_i} + \text{travel\_cost}(w_i \rightarrow N_{big}) \mid w_i \in N_{sml}\}$
    if $w \neq$ null **then**
        $t_{w_i} \leftarrow t_{w_i} + travel\_cost(w_i \rightarrow N_{big})$
        $N^*_{big} \leftarrow N_{big} \cup \{w^*\}$
        $N^*_{sml} \leftarrow N_{sml} \setminus \{w^*\}$
        if TWD(N^*_{big}) > TWD(N^*_{sml}) **then**
            $N_{big} \leftarrow N^*_{big}$
            $N_{sml} \leftarrow N^*_{sml}$
        else
            $t_{w_i} \leftarrow t_{w_i} - \text{travel\_cost}(w_i \rightarrow N_{big})$
            break
        end if
    else
        break
    end if
end while

6.2 WBT-NNH and WBT-NUD

The algorithmic design of WBT-NNH and WBT-NUD are shown in Algorithm 11 and Algorithm 13, respectively. Besides adding dynamic workload balancing (shown in Algorithm 10) into each algorithm, WBT-NUD uses NUD-IC [28] instead of NUDPF [28] to improve the efficiency.

Both WBT-NNH and WBT-NUD are based on WBT discussed in the last section. In WBT-NNH, a WBT root node is created first to represent the whole region Z including the worker set W and task set S (lines 1 - 2 in Algorithm 11). Such root node is passed as a parameter to a recursive function TopDownWbtNNH (line 3 in Algorithm 11). TopDownWbtNNH then checks the received parameter *wbt_node* to determine whether *wbt_node* contains more than one worker (line 1 in Algorithm 12). If *wbt_node* contains more than one worker, *wbt_node* will then be split into two children and TopDownWbtNNH will be invoked on each child recursively after workload being balanced between them (lines 2 - 6 in Algorithm 12). In this way, a WBT is constructed recursively. Once the WBT is created, every leaf node will be checked to determine whether there is one worker associated with it (line 22 in Algorithm 12). If yes, NNH [10] is invoked for the current leaf node (lines 23 - 27 in Algorithm 12) to identify tasks for the associated worker, i.e, we assign each worker to the tasks in the same node first. After all the 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 sibling node (lines 7 - 20 in Algorithm 12). A worker with an earlier arrival time will have a higher priority to select a task (line 13 in Algorithm 12). This process repeats iteratively from the leaf nodes to the root node. WBT-NUD follows the same process except that WBT-NUD employs NUD-IC (as illustrated in Algorithm 14) in each tree node and its corresponding sibling to dispatch workers to tasks iteratively.

Algorithm 11 WBT-NNH(Z, W, S)

Create an empty WBT wbt.
$wbt.root \leftarrow \{Z, W, S\}$.
TopDownWbtNNH($wbt.root$)

Algorithm 12 TopDownWbtNNH(wbt_node)

if wbt_node.NumberOfWorkers > 1 **then**
    $\{chld_1, chld_2\} \leftarrow qt\_node$.split()
    WorkloadBalance($chld_1, chld_2$)
    for all $chld_i \in \{chld_1, chld_2\}$ **do**
        TopDownWbtNNH($chld_i$)
    end for
    for all $chld_i \in \{chld_1, chld_2\}$ **do**
        $wk\_set \leftarrow \emptyset$
        for all $chld_j \in \{chld_1, chld_2\}$ and $chld_j \neq chld_i$ **do**
            $wk\_set \leftarrow wk\_set \cup chld_j$.getWorkers()
        end for
        while $wk\_set \neq \emptyset$ **do**
            $w^* \leftarrow \text{argmin}\{w_i.\text{getCurTime}() + \text{travel\_cost}(w_i \rightarrow chld_i) \mid w_i \in wk\_set\}$
            $S \leftarrow chld_i$.getTasks()
            $s_0 \leftarrow w^*$.getCurLocation()
            $t_0 \leftarrow w^*$.getCurTime()
            NNH(S, s_0, t_0)
            $wk\_set \leftarrow wk\_set \setminus \{w^*\}$
        end while
    end for
else
    if wbt_node.NumberOfWorkers $= 1$ **then**
        $S \leftarrow wbt\_node$.getTasks()
        $w \leftarrow wbt\_node$.getWorkers()
        $s_0 \leftarrow w$.getCurLocation()
        $t_0 \leftarrow w$.getCurTime()
        NNH(S, s_0, t_0)
    end if
end if

Algorithm 13 WBT-NUD($Z, W, S, pctl\_no$)

Create an empty WBT wbt.
$wbt.root \leftarrow \{Z, W, S\}$.
TopDownWbtNUD($wbt.root, pctl\_no$)

Algorithm 14 TopDownWbtNUD($wbt\_node, pctl\_no$)

1: **if** wbt_node.NumberOfWorkers > 1 **then**
2: $\{chld_1, chld_2\} \leftarrow qt\_node$.split()
3: WorkloadBalance($chld_1, chld_2$)
4: **for all** $chld_i \in \{chld_1, chld_2\}$ **do**
5: TopDownWbtNUD($chld_i, pctl\_no$)
6: **end for**
7: **for all** $chld_i \in \{chld_1, chld_2\}$ **do**
8: $wk\_set \leftarrow \emptyset$
9: **for all** $chld_j \in \{chld_1, chld_2\}$ and $chld_j \neq chld_i$ **do**
10: $wk\_set \leftarrow wk\_set \cup chld_j$.getWorkers()
11: **end for**
12: **while** $wk\_set \neq \emptyset$ **do**
13: $w^* \leftarrow \text{argmin}\{w_i.getCurTime()+ \text{travel\_cost}(w_i \rightarrow chld_i) \mid w_i \in wk\_set\}$
14: $S \leftarrow chld_i$.getTasks()
15: $s_0 \leftarrow w^*$.getCurLocation()
16: $t_0 \leftarrow w^*$.getCurTime()
17: Find optimal $\epsilon, MinPts, ddr$.
18: NUD-IC($S, s_0, t_0, pctl\_no, \epsilon, MinPts, ddr$)
19: $wk\_set \leftarrow wk\_set \setminus \{w^*\}$
20: **end while**
21: **end for**
22: **else**
23: **if** qt_node.NumberOfWorkers $= 1$ **then**
24: $S \leftarrow wbt\_node$.getTasks()
25: $w \leftarrow wbt\_node$.getWorkers()
26: $s_0 \leftarrow w$.getCurLocation()
27: $t_0 \leftarrow w$.getCurTime()
28: Find optimal $\epsilon, MinPts, ddr$.
29: NUD-IC($S, s_0, t_0, pctl\_no, \epsilon, MinPts, ddr$)
30: **end if**
31: **end if**

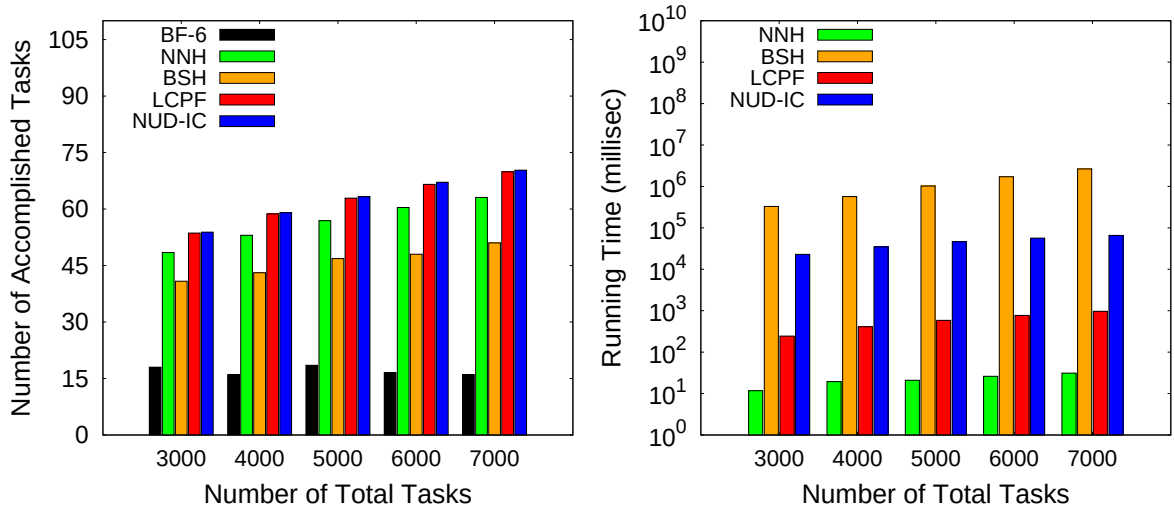
Chapter 7

Experimental Evaluation for SGTS Algorithms

In this chapter, we investigate the performance of LCPF and NUD-IC in terms of the number of accomplished tasks and CPU cost in support of the single worker geo-task scheduling using both real and synthetic datasets. All the experiments were run on a machine with POWER8E CPUs at 2.06 GHz and 1024 GB RAM. Each result is averaged over 2000 random queries (except BF-6 and BSH [10]), where the location of the worker was randomly generated in the target geographic areas.

7.1 All the 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 which has the least total cost of the traveling distance plus the task's execution duration.
- BSH [10]: the method which 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].



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

(b) CPU Cost (Syn-uniform)

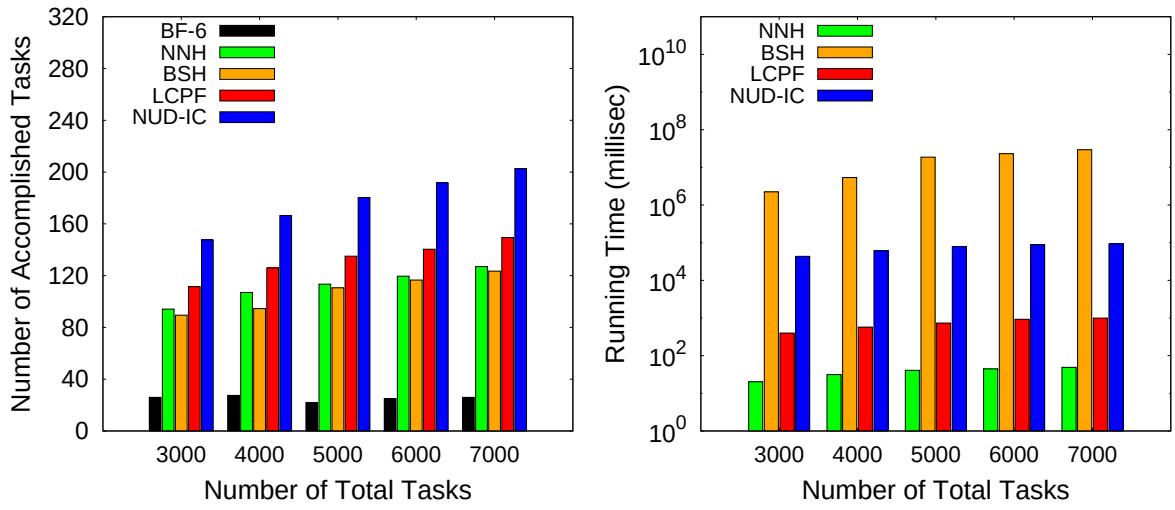
Figure 7.1: Effect of task number on the synthetic data of uniform distribution with expiration time range $[40\%, 60\%]$.

7.2 Experiments on Synthetic Data

We generated our synthetic data based on uniform and skewed distributions. For the skewed distribution, all the 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).

7.2.1 Impact of the Total Number of Tasks

Uniform Distribution. As shown in Figure 7.1, both LCPF and NUD-IC outperformed BF-6, NNH [10] and BSH [10] in terms of the number of accomplished tasks. On the other hand, 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 the 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)

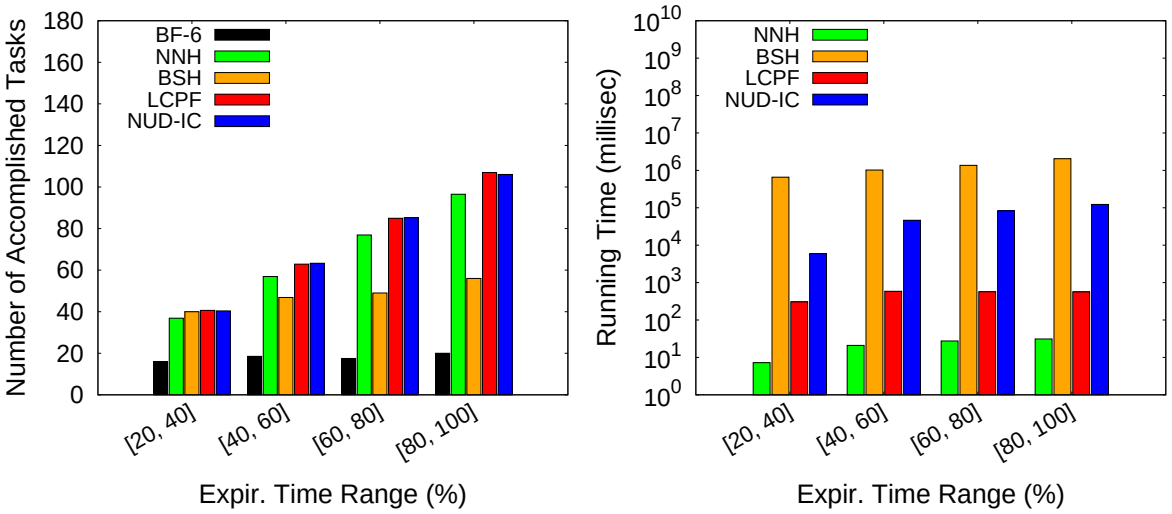
Figure 7.2: Effect of task number on the synthetic data of skewed distribution ($\sigma = 0.2$) with expiration time range $[40\%, 60\%]$.

Skewed Distribution. As shown in Figure 7.2, NUD-IC outperformed all the 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 6000, NUD-IC was able to return 192 tasks while NNH [10] only returned 120 tasks.

7.2.2 Impact of the Task Expiration Time

Uniform Distribution. As shown in Figure 7.3, 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, while 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 which 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



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

(b) CPU Cost (Syn-uniform)

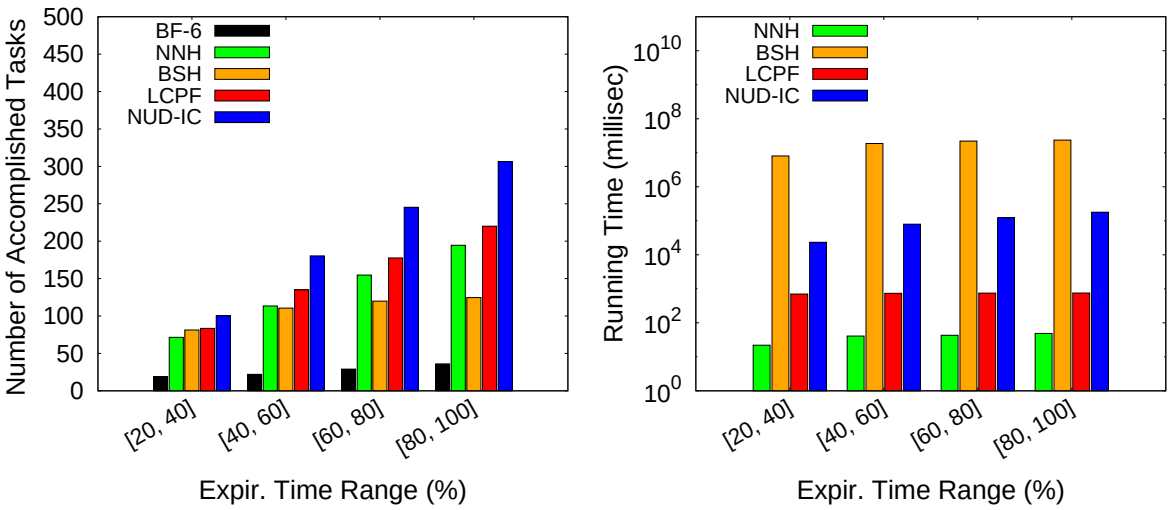
Figure 7.3: Effect of expiration time on the synthetic data of uniform distribution with task number 5000.

in the case that all the 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 7.4 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.

7.2.3 Impact of the Task Spatial Distribution

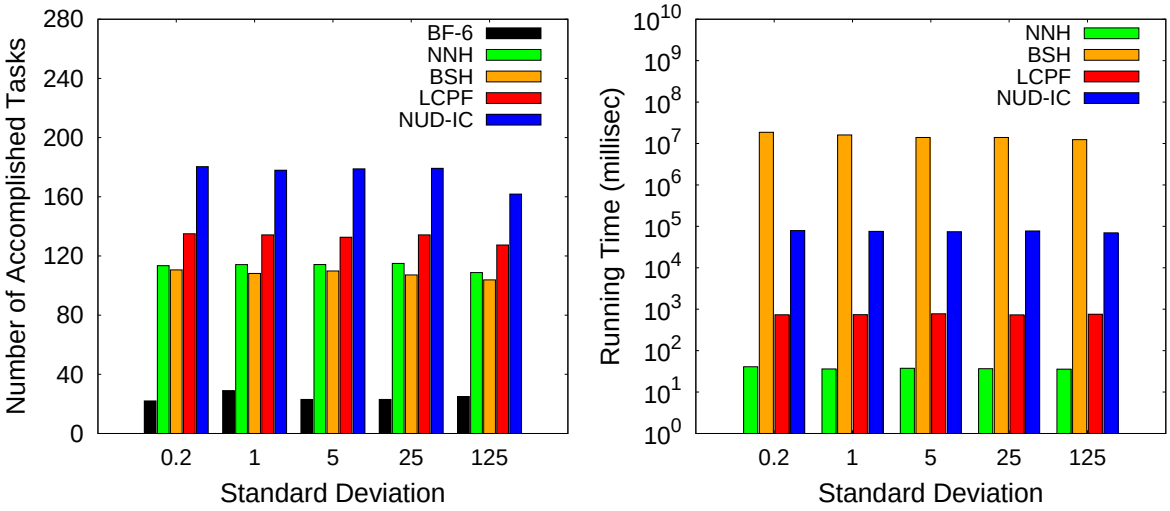
As shown in Figure 7.5, when we increased σ for skewed distribution, the number of accomplished tasks of all the algorithms went down. For example, when σ equaled 0.2, the number of completed tasks returned by NUD-IC and NNH [10] was 180 and 113, respectively, whereas when σ rose to 125, the number of accomplished tasks obtained by NUD-IC and NNH [10] declined to 162 and 109, respectively. Also, we can observe that the more skewed the task distribution is, the larger the advantage of NUD-IC over NNH [10] is. NUD-IC



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

(b) CPU Cost (Syn-Skewed)

Figure 7.4: Effect of expiration time on the synthetic data of skewed distribution ($\sigma = 0.2$) with task number 5000.

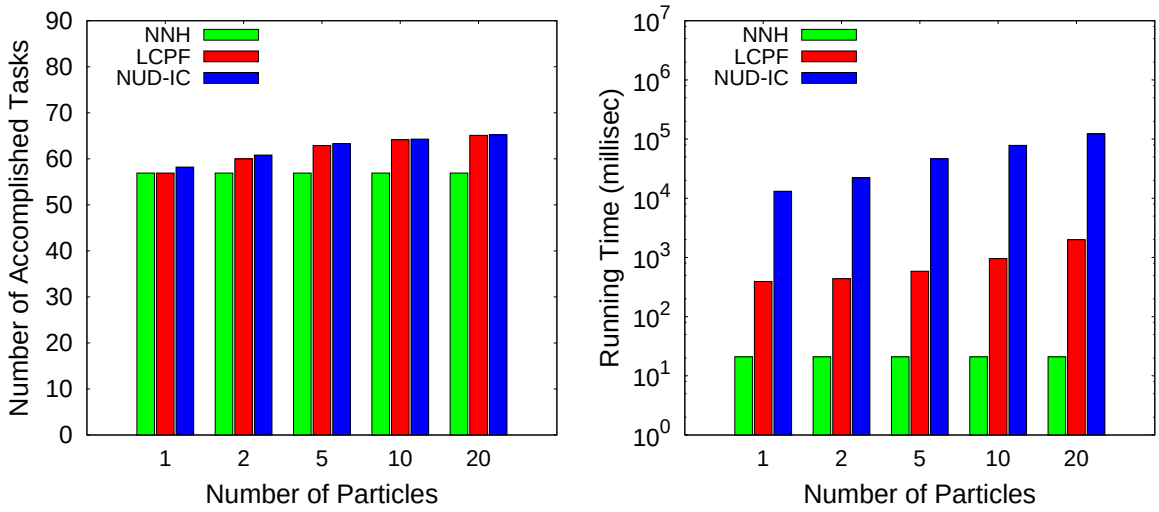


(a) Number of Accomplished Tasks (Syn-skewed)

(b) CPU Cost (Syn-Skewed)

Figure 7.5: Effect of σ on the synthetic data of skewed distribution with task number 5000 and expiration time range $[40\%, 60\%]$.

outperformed other algorithms in terms of the number of accomplished tasks and the CPU cost of all the algorithms slightly decreased in the increase of σ.



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

(b) CPU Cost (Syn-uniform)

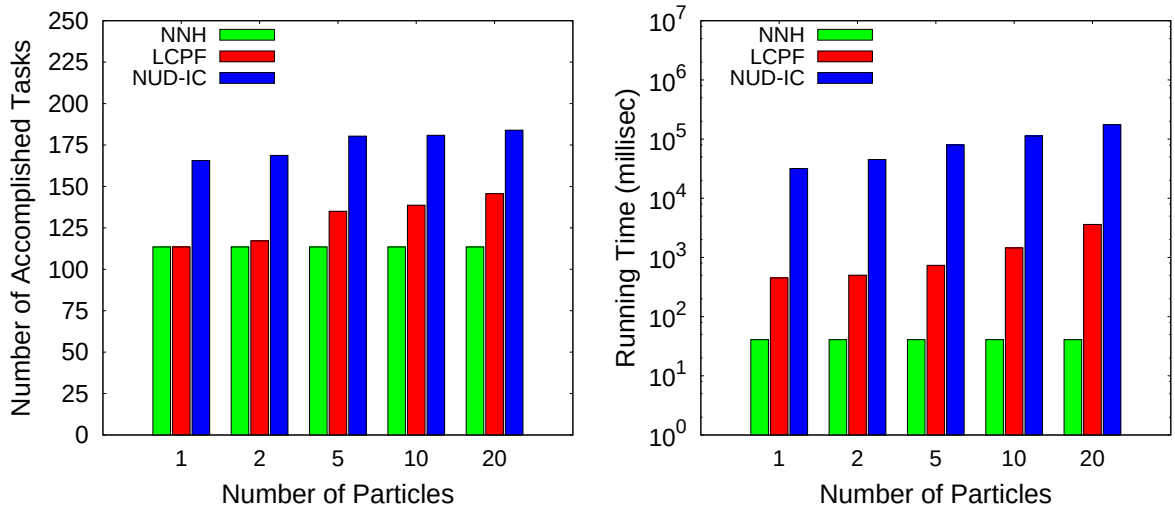
Figure 7.6: Effect of the number of particles on the synthetic data of uniform distribution with task number 5000 and expiration time range $[40\%, 60\%]$.

7.2.4 Impact of the Number of Particles

Figures 7.6 and 7.7 show the impact of the number of particles on the performance of LCPF and NUD-IC. We can observe that when there was only 1 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.

7.3 Experiments on Real Geo-Task Data

Our real data were collected from Foursquare [40] [39] [38], featuring New York City (42981 tasks) and Austin, Texas (8569 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 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.



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

(b) CPU Cost (Syn-Skewed)

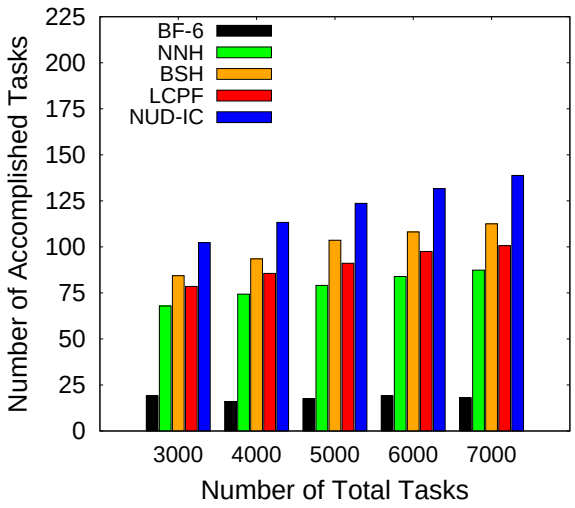
Figure 7.7: Effect of the number of particles on the synthetic data of skewed distribution ($\sigma = 0.2$) with task number 5000 and expiration time range $[40\%, 60\%]$.

7.3.1 Impact of the Total Number of Tasks

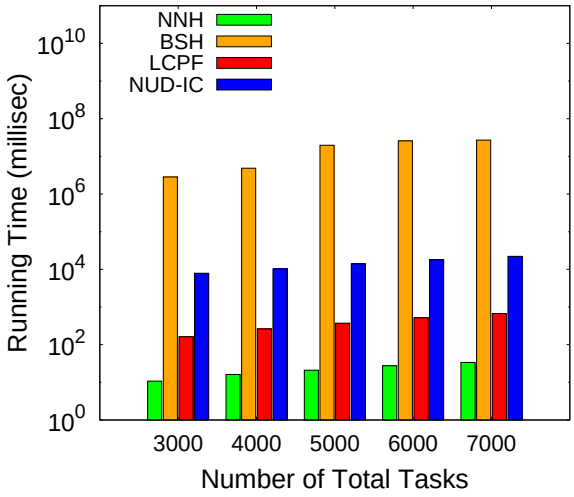
Figures 7.8 and 7.9 show the performance of all the algorithms in terms of the number of accomplished tasks and CPU cost based on POI distributions in New York City and Austin, Texas. NUD-IC was able to return more accomplished tasks than NNH [10] and LCPF at the price of the increased CPU cost. For example, when the total number of tasks was 7000 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. On the other hand, the CPU cost of all the algorithms increased as the total number of tasks enlarged.

7.3.2 Impact of the Task Expiration Time

As depicted in Figures 7.10 and 7.11, 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 mins, 120 mins] in New York City, the number of accomplished tasks returned by NUD-IC was 162, 1.5 times more than

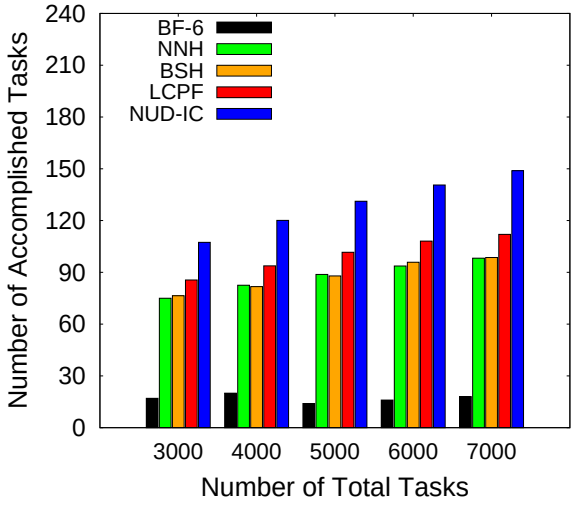


(a) Number of Accomplished Tasks (New York)

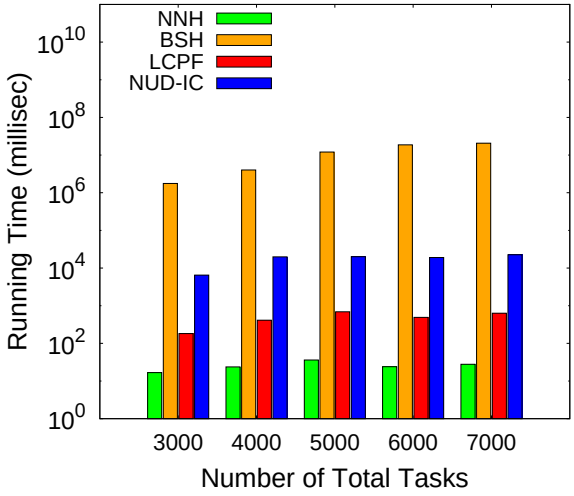


(b) CPU Cost (New York)

Figure 7.8: Effect of task number on the dataset of New York City with expiration time range [60 mins, 90 mins].



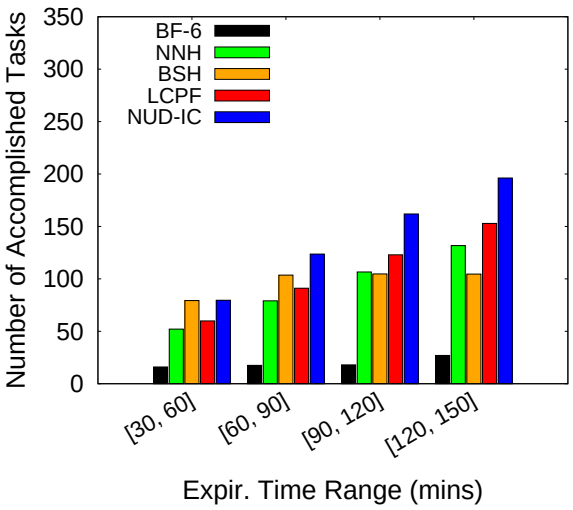
(a) Number of Accomplished Tasks (Austin)



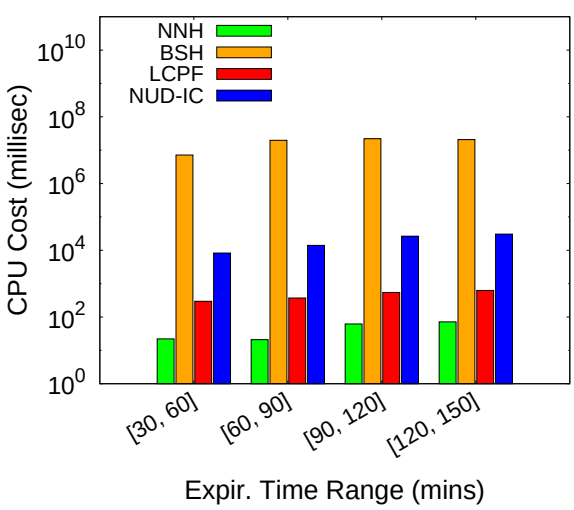
(b) CPU Cost (Austin)

Figure 7.9: Effect of task number on the dataset of Austin, Texas, with expiration time range [60 mins, 90 mins].

that obtained by NNH [10], which was 107. BSH [10] showed little improvement in terms of the number of accomplished tasks when all the tasks expire after 90 minutes in both cities.

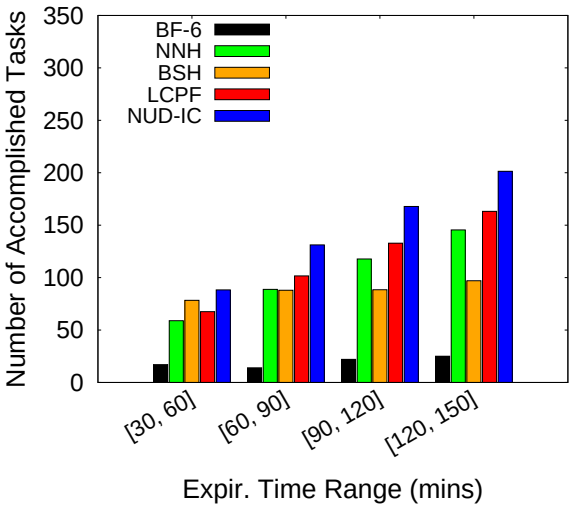


(a) Number of Accomplished Tasks (New York)

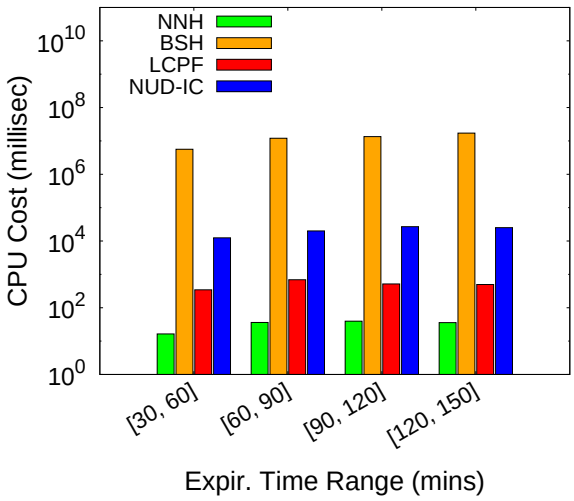


(b) CPU Cost (New York)

Figure 7.10: Effect of expiration time on the dataset of New York City with task number 5000.



(a) Number of Accomplished Tasks (Austin)

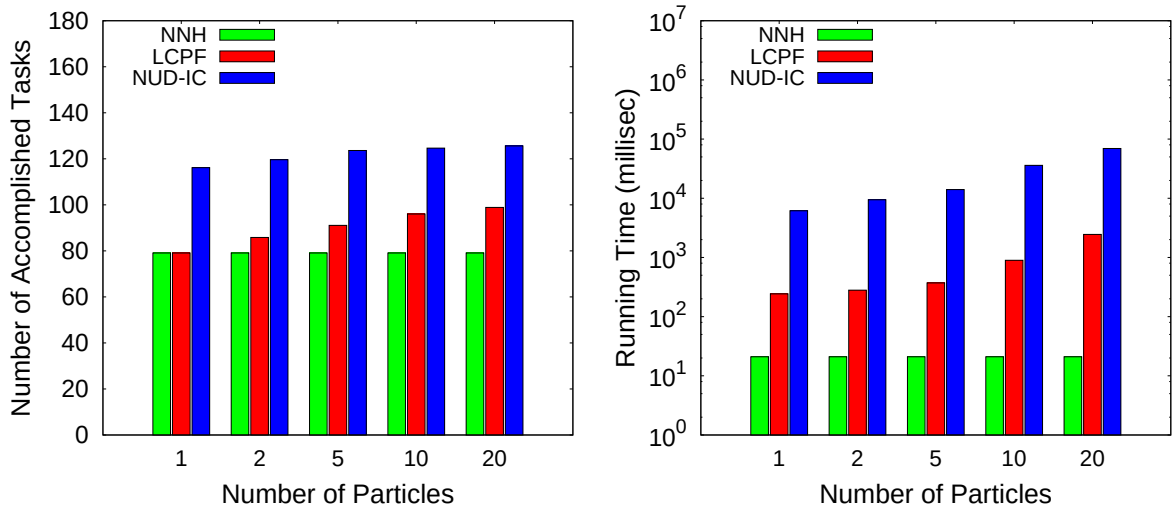


(b) CPU Cost (Austin)

Figure 7.11: Effect of expiration time on the dataset of Austin, Texas, with task number 5000.

7.3.3 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



(a) Number of Accomplished Tasks (New York)

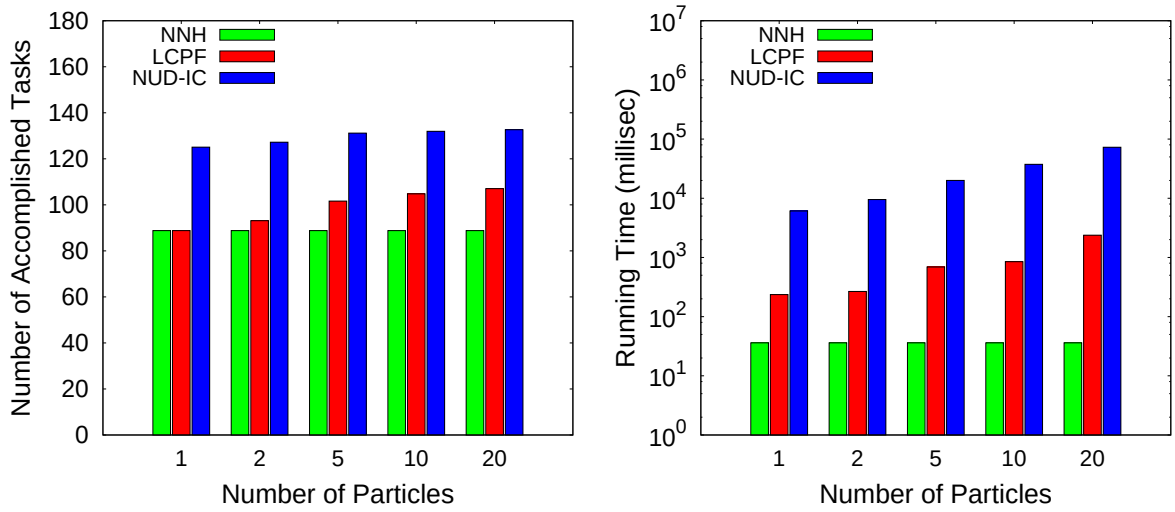
(b) CPU Cost (New York)

Figure 7.12: Effect of number of particles on the dataset of New York City with task number 5000, expiration time range [60 mins, 90 mins].

and CPU cost of LCPF and NUD-IC increased, as shown in Figures 7.12 and 7.13. Also, given the same number of particles, NUD-IC successfully returned much 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].

7.4 Case Study

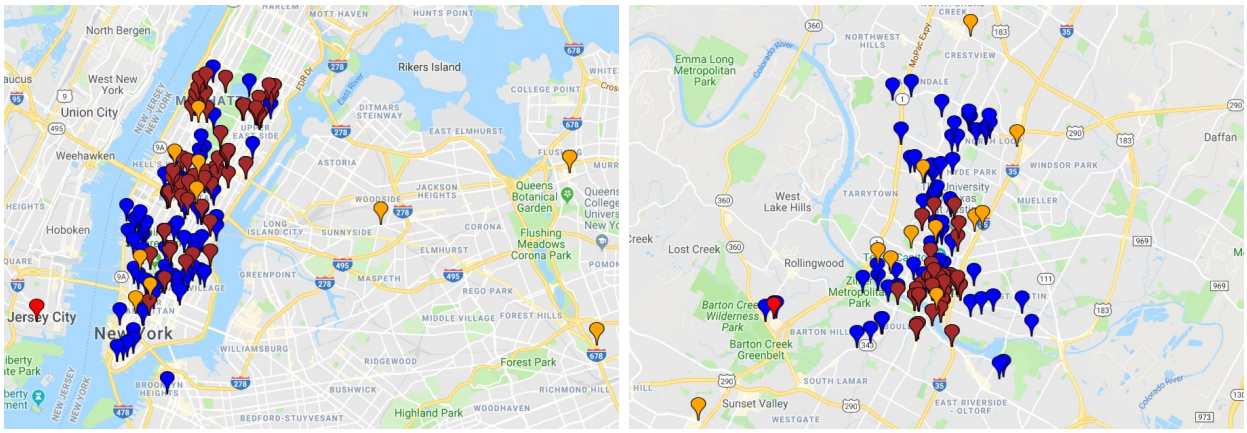
In this section, we randomly selected 1000 spatial tasks from New York City, and 1000 spatial tasks from Austin, Texas, respectively, and then launched all the algorithms from a random start location of the worker in each city. In the case of New York City, BF-6, NNH [10], BSH [10], LCPF and NUD-IC returned 23 tasks, 42 tasks, 67 tasks, 117 tasks and 125 tasks, respectively, while 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. Figures 7.14 (a) and (b) depict the corresponding accomplished tasks returned by NUD-IC and BSH [10] respectively in both New York City and Austin, Texas. Specifically, as illustrated in Figures 7.14 (a) and (b), NUD-IC was able to return a much higher number of



(a) Number of Accomplished Tasks (Austin)

(b) CPU Cost (Austin)

Figure 7.13: Effect of number of particles on the dataset of Austin, Texas, with task number 5000, expiration time range [60 mins, 90 mins].



(a) New York City

(b) Austin

Figure 7.14: Spatial tasks returned by NUD-IC and BSH [10], respectively, in New York City (a) and Austin, Texas (b). Red dot represents 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.

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).

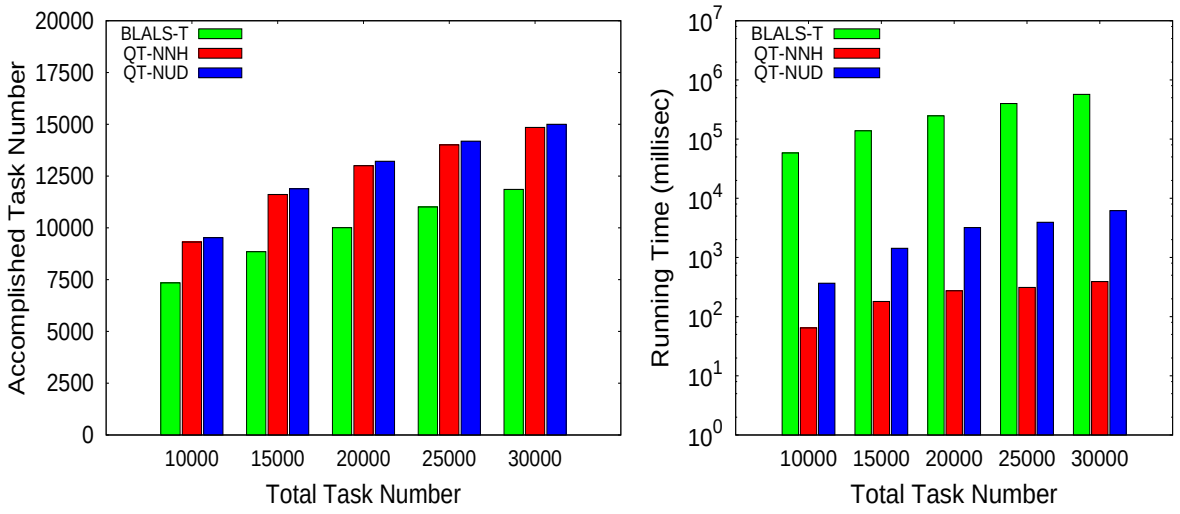
Chapter 8

Experimental Evaluation for Quadtree based MGTS Algorithms

In this chapter, we investigate the performance of QT-NNH and QT-NUD in terms of the number of accomplished tasks and CPU cost in support of the multiple worker geo-task scheduling on real data. All the experiments were run in the same setting as in the last chapter unless otherwise specified. The real data were collected from Foursquare [40] [39] [38], featuring New York City (42981 tasks) and Tokyo (67124 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, 90 to 150 minutes, with 60 to 120 minutes as the default interval.

8.1 All the MGTS Methods for Comparison

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



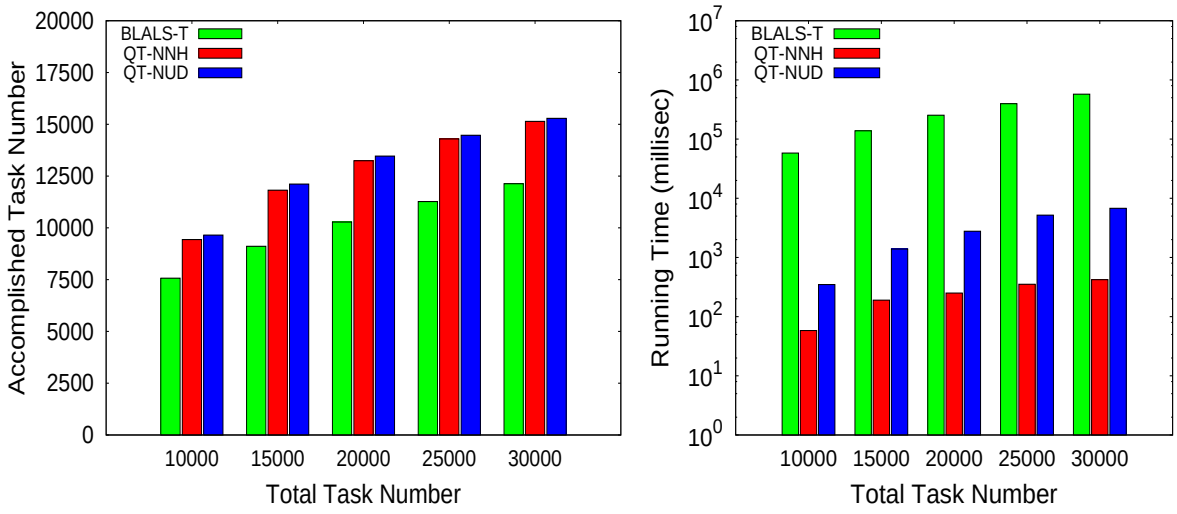
(a) Number of Accomplished Tasks (New York)

(b) CPU Cost (New York)

Figure 8.1: 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 mins, 120 mins]; for QT-NUD, $ptcl\_no = 1$.

8.2 Impact of the Total Number of Tasks

Figures 8.1 and 8.2 show the impact of the total number of tasks on the performance of all the 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 is due to 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 can not be assigned to more tasks than a predefined threshold, which is not the case in reality. Last but not the 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 (Tokyo)

(b) CPU Cost (Tokyo)

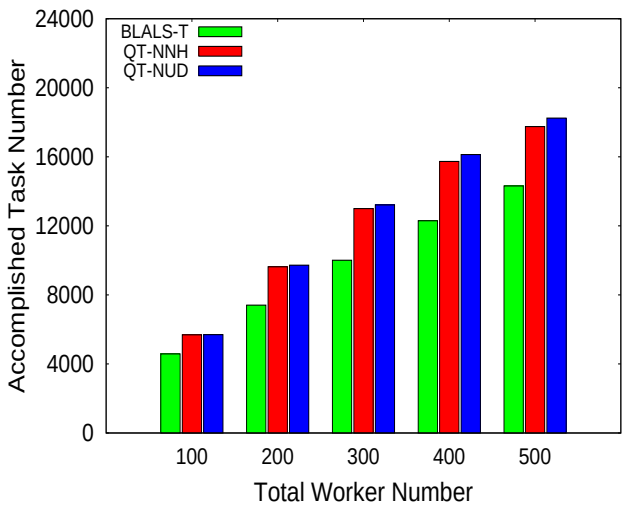
Figure 8.2: Effect of the total number of tasks on the dataset of Tokyo with total number of workers 300 and expiration time range [60 mins, 120 mins]; for QT-NUD, $ptcl\_no = 1$.

8.3 Impact of the Total Number of Workers

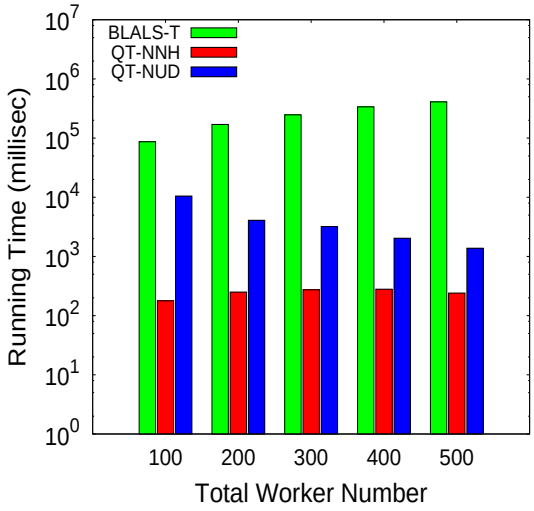
Figures 8.3 and 8.4 show the impact of the total number of workers on the performance of all the algorithms in terms of the number of accomplished tasks and CPU cost in New York City and Tokyo. As depicted in Figures 8.3 and 8.4, when the number of workers increased, QT-NUD and QT-NNH outperformed BLALS-T 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.

8.4 Impact of the Expiration Time

Figures 8.5 and 8.6 show the impact of the expiration time on the performance of all the algorithms in terms of the number of accomplished tasks and CPU cost in New York City and Tokyo. As shown in Figures 8.5 and 8.6, QT-NUD and QT-NNH outperformed

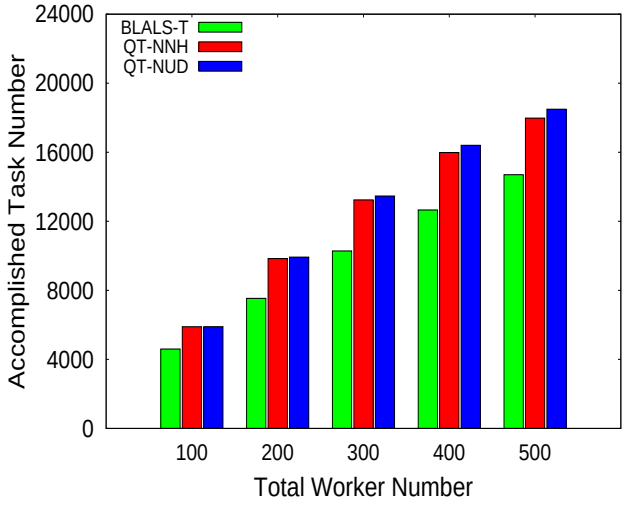


(a) Number of Accomplished Tasks (New York)

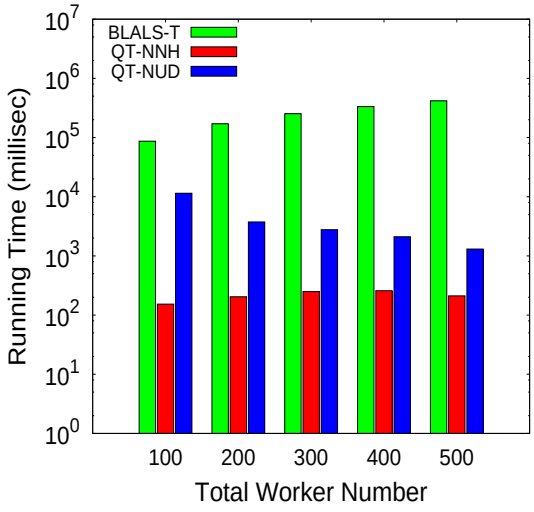


(b) CPU Cost (New York)

Figure 8.3: Effect of the total number of workers on the dataset of New York City with total number of tasks 20000 and expiration time range [60 mins, 120 mins]; for QT-NUD, $ptcl\_no = 1$.



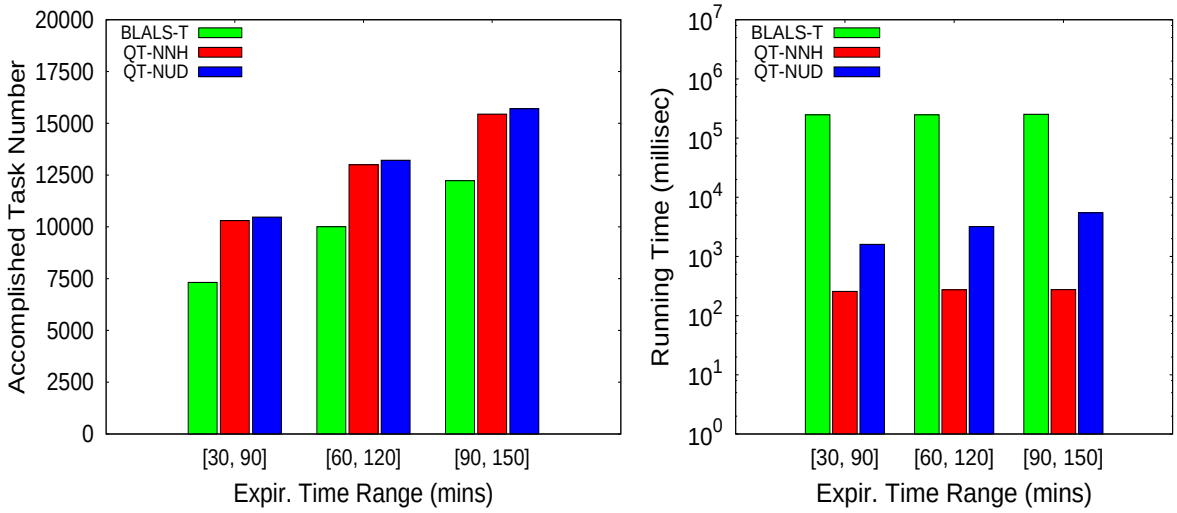
(a) Number of Accomplished Tasks (Tokyo)



(b) CPU Cost (Tokyo)

Figure 8.4: Effect of the total number of workers on the dataset of Tokyo with total number of tasks 20000 and expiration time range [60 mins, 120 mins]; for QT-NUD, $ptcl\_no = 1$.

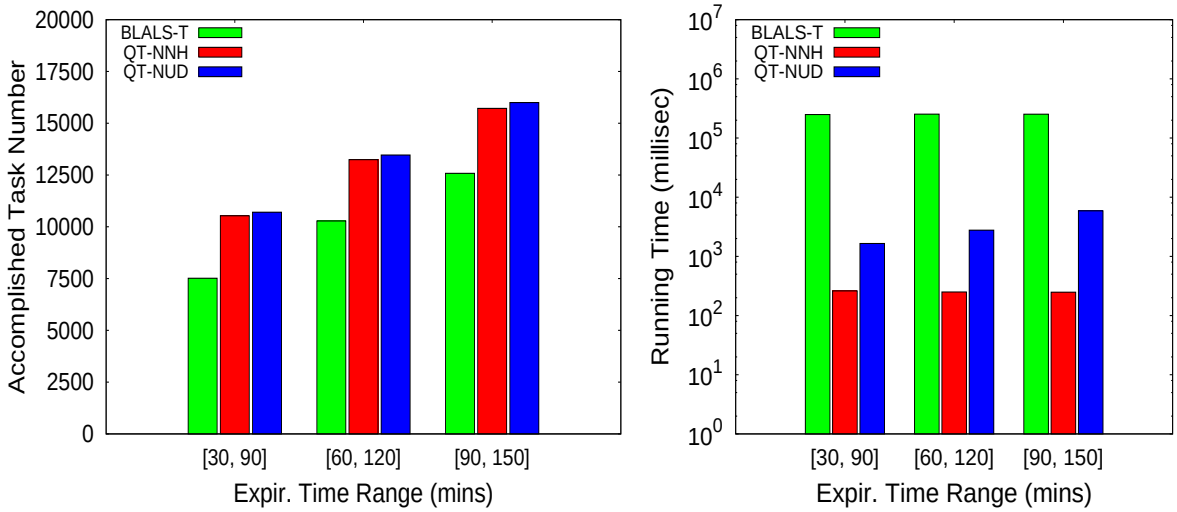
BLALS-T in terms of both the number of accomplished tasks and CPU time as the expiration time increased. With the increased expiration time, all three approaches returned a higher number of completed tasks.



(a) Number of Accomplished Tasks (New York)

(b) CPU Cost (New York)

Figure 8.5: Effect of the expiration time on the dataset of New York City with total number of workers 300 and total number of tasks 20000; for QT-NUD, $ptcl\_no = 1$.



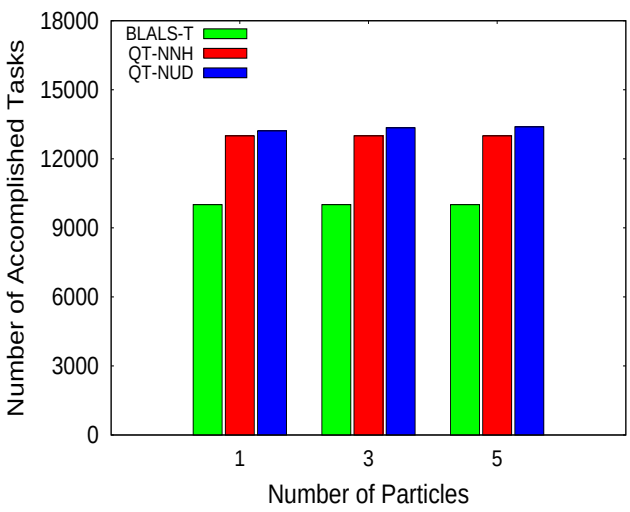
(a) Number of Accomplished Tasks (Tokyo)

(b) CPU Cost (Tokyo)

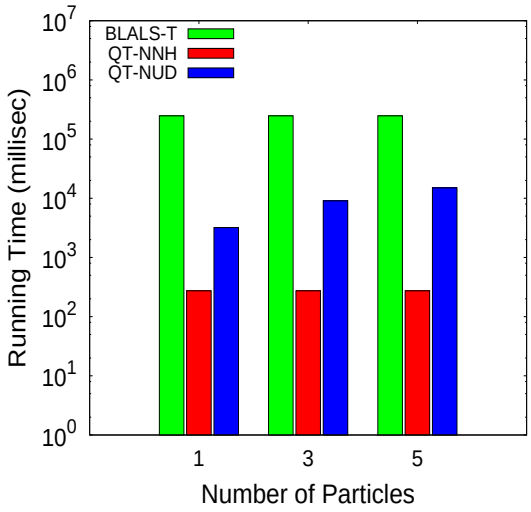
Figure 8.6: Effect of the expiration time on the dataset of Tokyo with total number of workers 300 and total number of tasks 20000; for QT-NUD, $ptcl\_no = 1$.

8.5 Impact of the Number of Particles

Figures 8.7 and 8.8 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



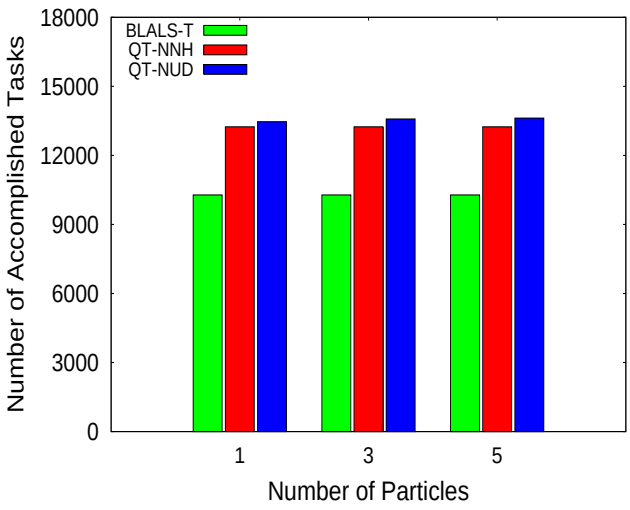
(a) Number of Accomplished Tasks (New York)



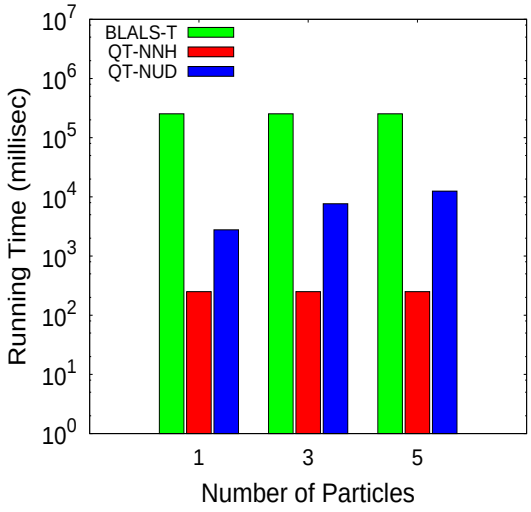
(b) CPU Cost (New York)

Figure 8.7: Effect of the number of particles on the dataset of New York City with total number of workers 300, total number of tasks 20000, and expiration time range [60 mins, 120 mins].

in New York City and Tokyo. As we can see in Figures 8.7 and 8.8, it was observed that when the number of particles increased, the performance gain of QT-NNH and QT-NUD in terms of the number of completed tasks was very limited while 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.



(a) Number of Accomplished Tasks (Tokyo)



(b) CPU Cost (Tokyo)

Figure 8.8: Effect of the number of particles on the dataset of Tokyo with total number of workers 300, total number of tasks 20000, and expiration time range [60 mins, 120 mins].

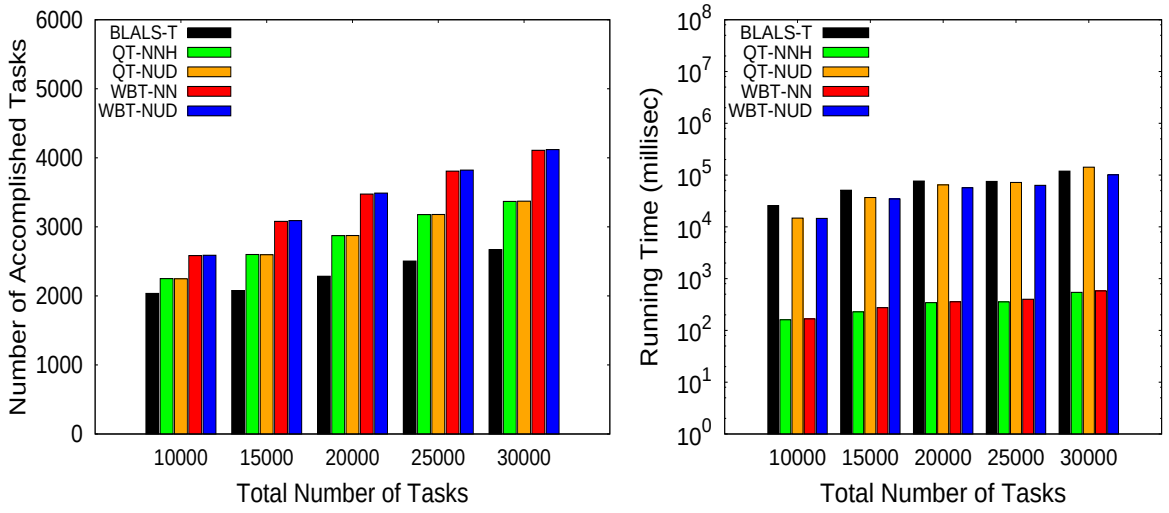
Chapter 9

Experimental Evaluation for WBT based MGTS Algorithms

In this chapter, we compare the performance of all the algorithms in support of multiple worker geo-task scheduling in terms of the number of accomplished tasks and CPU cost on real data. All the experiments were run in the same setting as in the last chapter unless otherwise specified. The real data were collected from Foursquare [40] [39] [38], featuring New York City (42981 tasks) and Tokyo (67124 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 interval 60 to 120 minutes.

All the MGTS Methods for Comparison:

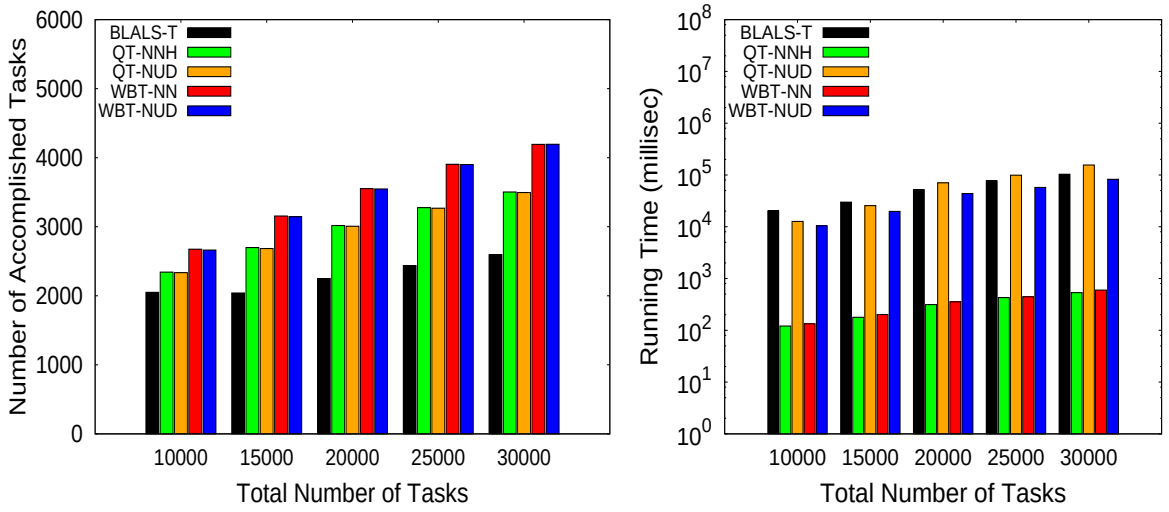
- BLALS-T [11]: the method which partitions the worker-task flow network from top to down recursively and then merges all the partitions from bottom to up.
- QT-NNH [28]: our proposed method which leverages the point-region quadtree to ensure workload balancing among multiple workers and schedule tasks for each worker via NNH [10].
- QT-NUD [28]: our proposed method which leverages the point-region quadtree to ensure workload balancing among multiple workers and schedule tasks for each worker via NUD-IC [27].
- WBT-NNH: our proposed method which leverages a novel workload-balancing bisection tree (WBT) to enable dynamic workload balancing among multiple workers and schedule tasks for each worker via NNH.



(a) Number of Accomplished Tasks (New York)

(b) CPU Cost (New York)

Figure 9.1: Effect of the total number of tasks on the dataset of New York City with total number of workers 40 and expiration time range [60 mins, 120 mins]; for WBT-NUD, $ptcl\_no = 1$.

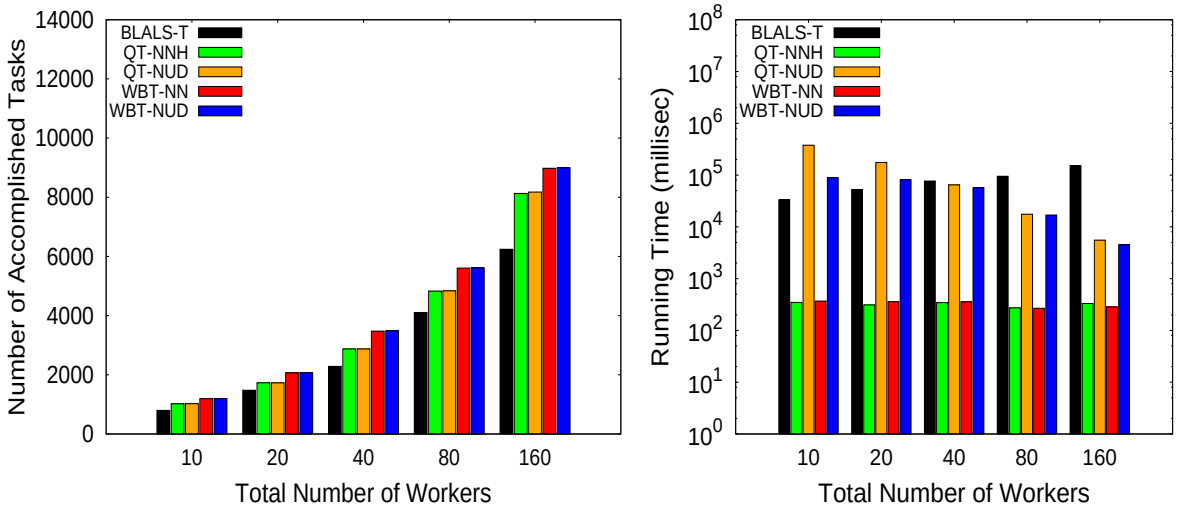


(a) Number of Accomplished Tasks (Tokyo)

(b) CPU Cost (Tokyo)

Figure 9.2: Effect of the total number of tasks on the dataset of Tokyo with total number of workers 40 and expiration time range [60 mins, 120 mins]; for WBT-NUD, $ptcl\_no = 1$.

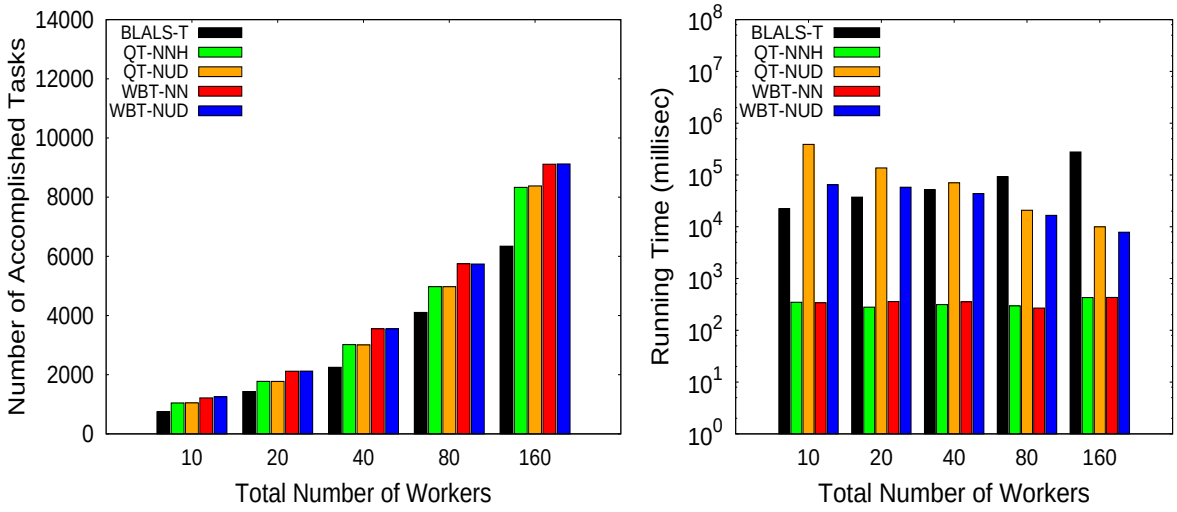
- WBT-NUD: our proposed method which leverages a novel workload-balancing bisection tree (WBT) to enable dynamic workload balancing among multiple workers and schedule tasks for each worker via NUD-IC.



(a) Number of Accomplished Tasks (New York)

(b) CPU Cost (New York)

Figure 9.3: Effect of the total number of workers on the dataset of New York City with total number of tasks 20000 and expiration time range [60 mins, 120 mins]; for WBT-NUD, $ptcl\_no = 1$.



(a) Number of Accomplished Tasks (Tokyo)

(b) CPU Cost (Tokyo)

Figure 9.4: Effect of the total number of workers on the dataset of Tokyo with total number of tasks 20000 and expiration time range [60 mins, 120 mins]; for WBT-NUD, $ptcl\_no = 1$.

Impact of the total number of tasks. Figures 9.1 and 9.2 show the impact of the total number of tasks on the performance of all the algorithms in terms of the number of tasks accomplished and the running time in New York City and Tokyo respectively. As

shown in Figures 9.1 and 9.2, WBT-NNH and WBT-NUD consistently outperformed all the other methods in terms of the number of tasks accomplished without compromising their efficiency.

Impact of the Total Number of Workers. Figures 9.3 and 9.4 show the impact of the total number of workers on the performance of all the algorithms in terms of the number of tasks accomplished and CPU cost in New York City and Tokyo. As depicted in Figures 9.3 and 9.4, WBT-NNH and WBT-NUD consistently outperformed all the other methods in terms of the number of tasks accomplished without compromising their efficiency.

Chapter 10

Demonstration

We built Turbo-GTS demo using Java on Google Maps API and a back-end MySQL database under JSP framework. Turbo-GTS demo incorporates WBT-NNH and WBT-NUD, our two newly developed scheduling algorithms, as well as BLALS-T (as baseline to represent the state-of-the-art), QT-NNH, and QT-NUD. The project source code is available at https://github.com/WeiTerryLi/Turbo-GTS-Demo.

Turbo-GTS demo shows task statistics/pattern for users to compare different methods in terms of total task count, task assignment distribution over all workers, and task distribution for any worker or for any spatial region specified by users. Here we demonstrate three essential interactive use cases: (1) comparing task assignments for the whole worker group returned by different algorithms, (2) visualizing the task assignment distribution over all workers, and (3) identifying the optimal task assignment for a specific worker. Figure 10.1, Figure 10.2 and Figure 10.3 show the task assignments using different scheduling algorithms in Turbo-GTS demo given $3,000$ randomly distributed tasks and 50 workers randomly located in New York City. Specifically, Figure 10.1 presents statistical results of spatial tasks returned by BLALS-T [11] and WBT-NNH, respectively, in New York City, in which the table shows the performance comparison between BLALS-T [11] and WBT-NNH in summary, whereas the chart compares the number of tasks accomplished by each worker between BLALS-T [11] and WBT-NNH, and visualizes the task assignment distribution over all workers. It is conspicuous that WBT-NNH not only resulted in more number of tasks accomplished and much less CPU cost, but also distributed tasks to each worker more evenly, which means workers have had a better cooperation under the scheduling of WBT-NNH. Figure 10.2 further shows visualized results of spatial tasks returned by BLALS-T [11] and WBT-NNH,

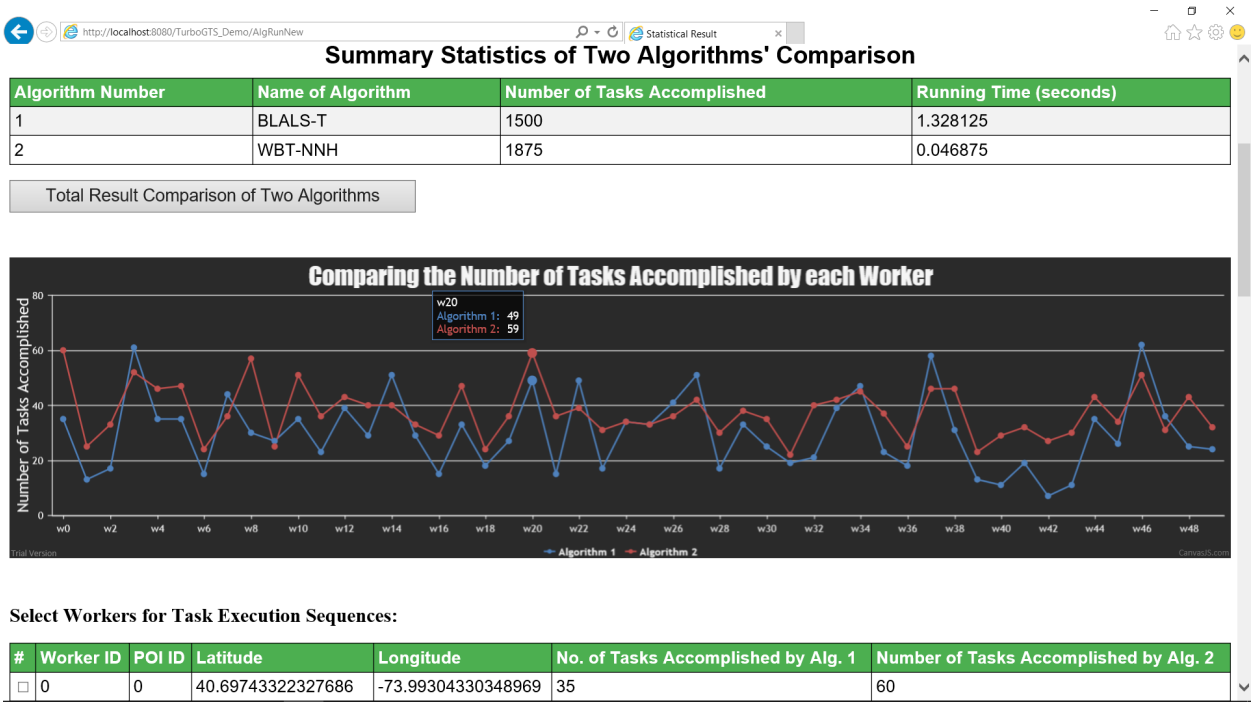


Figure 10.1: Statistical results of spatial tasks returned by BLALS-T [11] and WBT-NNH, respectively, in New York City. The table shows the performance comparison between BLALS-T [11] and WBT-NNH in summary, whereas the chart compares the number of tasks accomplished by each worker between BLALS-T [11] and WBT-NNH, and visualizes the task assignment distribution over all workers.

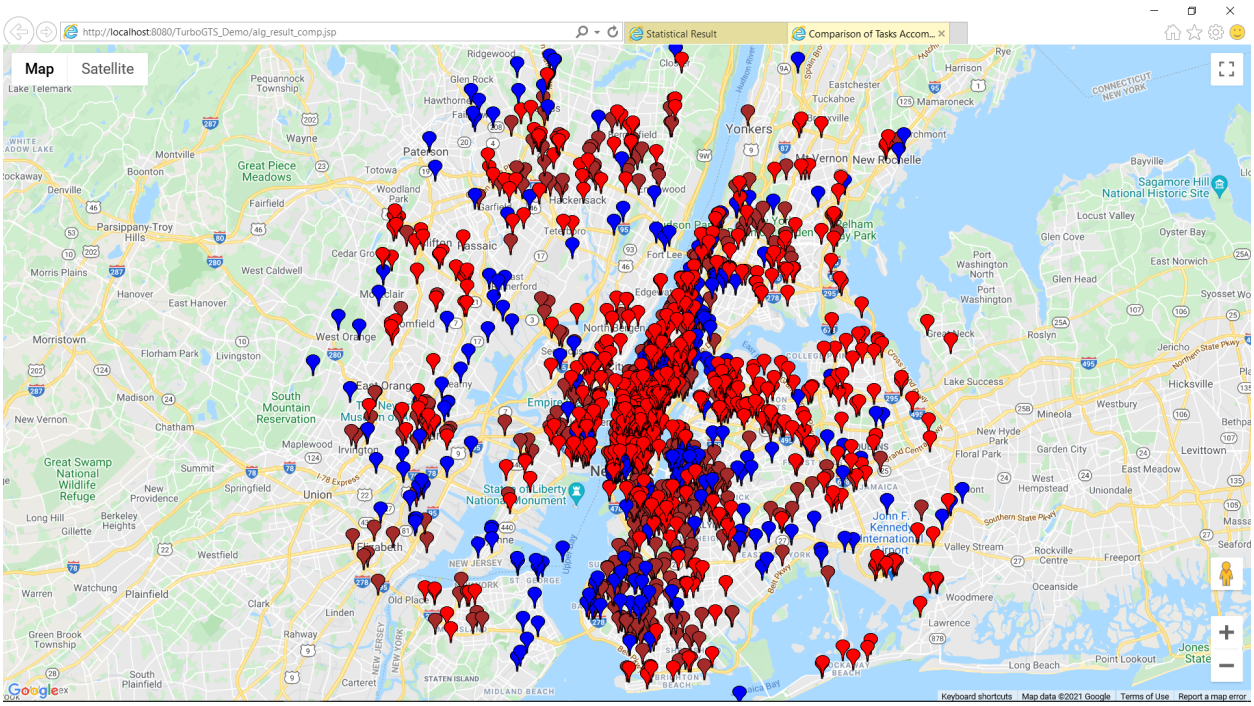
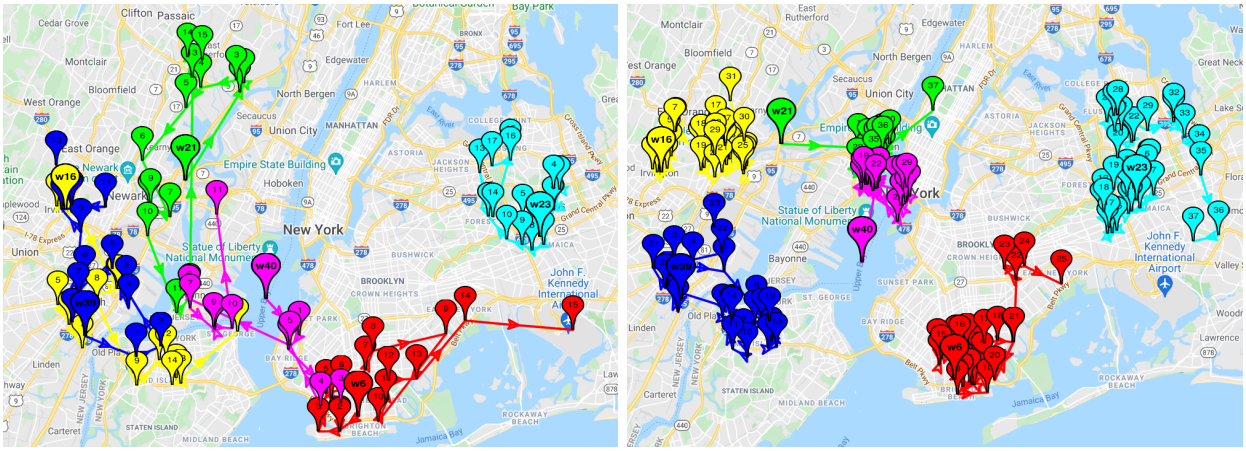


Figure 10.2: Visualized results of spatial tasks returned by BLALS-T [11] and WBT-NNH, respectively, in New York City. Blue dots are tasks accomplished exclusively by BLALS-T [11], red dots are tasks accomplished exclusively by WBT-NNH, and brown dots are tasks accomplished by both.



(a) Task sequences returned by BLALS-T (b) Task sequences returned by WBT-NNH

Figure 10.3: Comparison of task execution sequences of selected workers obtained from BLALS-T [11] and WBT-NNH, respectively, in New York City. Each large dot with w followed by a number inside represents a selected worker with her own id, while each small dot represents a task on a worker's route and the number inside indicates the order of that task to be performed en route.

respectively, in New York City. Blue dots are tasks accomplished exclusively by BLALS-T [11], red dots are tasks accomplished exclusively by WBT-NNH, and brown dots are tasks accomplished by both. Figure 10.3 compares the task execution sequences of selected workers obtained from BLALS-T [11] and WBT-NNH, respectively, in New York City, where each worker's route is marked with a distinct color different from others. Each large dot with w followed by a number inside represents a selected worker with her own id, while each small dot represents a task on a worker's route and the number inside indicates the order of that task to be performed en route. It is also obvious that each worker has got more tasks to perform along her route following the assignment and scheduling of WBT-NNH than BLALS-T [11].

Chapter 11

Conclusion and Future Work

In this dissertation, 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 in order 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. This chapter summarizes all the contributions made in this dissertation study and discusses the future work as an extension of this dissertation.

11.1 Main Contributions

The ubiquity of mobile platforms and smart phones breeds a large number of mobile crowdsourcing applications. A common feature of all 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.

11.1.1 Single-worker Geo-Task Scheduling

We define the Non-Urgency Degree (NUD) for task assignment and present two approximate solutions, LCPF and NUD-IC, for the Single Worker Geo-Task Scheduling (SGTS) problem. LCPF solves the problem by incorporating particle filtering while NUD-IC generates the sub-optimal schedule by integrating particle filtering with DBSCAN clustering based on Non-Urgency Degree (NUD). We even compare our proposed methods with the baseline methods theoretically and then shed light on the performance improvement of the

proposed algorithms against the baseline methods using an illustrative example. The effectiveness and scalability of the two proposed solutions are further verified by our extensive experiments using both real and synthetic data. Specifically, our city-scale experiments show that both LCPF and NUD-IC can yield a much higher number of completed tasks than the state-of-the-art approaches in the literature.

11.1.2 Multi-worker Geo-Task Scheduling based on Quadtree

Inspired by the proposed LCPF and NUD-IC for the Single Worker Geo-Task Scheduling (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 the point-region quadtree to ensure workload balancing among multiple workers. We even compare our proposed methods with the baseline methods theoretically and then shed light on the performance improvement of the proposed algorithms against the baseline methods using an illustrative example. The effectiveness and efficiency of QT-NNH and QT-NUD are verified by our extensive experiments using real data from metropolises like New York City and Tokyo. Compared with 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.

11.1.3 Multi-worker Geo-Task Scheduling based on Workload-balancing Bisection Tree (WBT)

To provide more effective and dynamic workload balancing among all the workers, we propose a novel tree structure, Workload-balancing Bisection Tree (WBT), and further present two more advanced scheduling algorithms WBT-NNH and WBT-NUD to solve the Multi-Worker Geo-Task scheduling (MGTS) problem. WBT-NNH and WBT-NUD build on algorithms QT-NNH and QT-NUD but utilize WBT to perform workload balancing instead

of quadtree. Using the Foursquare mobile user check-in data in New York City and Tokyo, we show the superiority of WBT-NNH and WBT-NUD over the state of the art including our previously proposed QT-NNH and QT-NUD in terms of the total number of the tasks that can be accomplished by the entire worker group and the corresponding running time.

11.1.4 Demonstration

We built Turbo-GTS demo using Java on Google Maps API and a back-end MySQL database under JSP framework. Turbo-GTS demo incorporates WBT-NNH and WBT-NUD, our two newly developed scheduling algorithms, as well as BLALS-T (as baseline to represent the state-of-the-art), QT-NNH, and QT-NUD. Turbo-GTS demo shows task statistics/pattern for users to compare different methods in terms of total task count, task assignment distribution over all workers, and task distribution for any worker or for any spatial region specified by users. We demonstrate three essential interactive use cases: (1) comparing task assignments for the whole worker group returned by different algorithms, (2) visualizing the task assignment distribution over all workers, and (3) identifying the optimal task assignment for a specific worker.

11.2 Future Work

11.2.1 Task Rewards in Real-world Mobile Crowdsourcing

In real-world mobile crowdsourcing, different tasks tend to have different weight. The commonest task weight in real-world crowdsourcing is reflected by the reward for accomplishing a task. Obviously for workers, tasks with high rewards are more worth doing than tasks with low rewards. As a result, the motivation of workers in mobile crowdsourcing will be changed from accomplishing as many tasks as possible to obtaining a reward as high as possible. This change has to be reflected in the algorithmic design accordingly, so I plan to take task rewards into consideration in our future work.

11.2.2 Competition and Collaboration in Real-world Mobile Crowdsourcing

Although our framework provides solutions to Multi-Worker Geo-Task scheduling (MGTS) problem in maximizing the total number of tasks accomplished by the whole group of workers, this is only feasible for workers who are willing to sacrifice their own interests for the greater good. Nevertheless, considering the complexity of the human nature, workers who are only concerned about their own benefits may not want to strictly follow the plan and will try to gain as many tasks or as much reward as possible. As a result, more than one worker are very likely to compete the same task as long as this task has not yet been marked as accomplished in the server end. Thus, I plan to introduce either a competition avoiding mechanism to get around this kind of situations, or some other techniques to coordinate the collaboration of workers and distribute the rewards among them according to their respective contributions in my future work.

11.2.3 Personal Factors in Real-world Mobile Crowdsourcing

There are several personal factors in real-world mobile crowdsourcing that our solutions have not covered yet, as a worker's personal conditions tend to have a considerable impact on her productivity, like her available time, her age, her health and financial conditions, expertise, and so on. Available time, age and health conditions determine how long a worker can be available each day. Financial conditions determine the modes of transportation that a worker chooses on her way to the spots of tasks. If a worker is young and healthy, owns a car, and has money enough to afford the gas for a long journey within a single day, then she has much bigger potential to complete more tasks than others and should be assigned with more tasks. However, if a worker is old and poor, has health conditions, and has to go to tasks on foot, by bike or taking public transport, then her priority of task assignment should be adjusted lower than others no matter how many tasks she is close to. Expertise sometimes also plays an important role in mobile crowdsourcing, as in some applications,

tasks need workers to have certain technical background to get them done. In my future work, I plan to integrate all of these personal factors into our solutions.

11.2.4 Environmental Factors in Real-world Mobile Crowdsourcing

I plan to take the environmental factors into account in my future work, such as the real-time traffic, some real-time incidents like road construction/block, weather, and so on. These environmental factors may not seem that noticeable at the first sight, but could incur considerable impact on the performance of the solutions, for exceptions and accidents might happen at any time. To counter this, I will add a real-time traffic detector into my current framework to reflect the latest effects of traffic on the traveling costs among workers and tasks.

11.2.5 Privacy Issues in Real-world Mobile Crowdsourcing

In most mobile crowdsourcing applications, the information of workers and tasks are just exposed to the platform, which incurs serious threats to the privacy of both workers and tasks. A curious and dishonest platform could compromise and exploit the information of both workers and tasks in many different ways. For example, workers may be given the wrong information about the locations and rewards of the tasks published by someone whom the platform does not like so that those tasks can never be done; or the platform can sell the workers' private information like home addresses, ages, health and financial conditions, expertise, and so on for profit. Hence, I plan to incorporate the latest cloud security strategies into my current work on mobile crowdsourcing to propose a secured mobile crowdsourcing scheme, in which the privacy of workers and tasks can be protected from the curious and dishonest platform without compromising the computational power of platform in generating optimal or near-optimal task assignment plans and execution schedules for all the workers.

Bibliography

[1] A. Acker, M. Lukac, and D. Estrin. Participatory Sensing for Community Data Campaigns: A case study. *Center for Embedded Network Sensing*, 2010.

[2] A. Afuah and C. L. Tucci. Crowdsourcing as a Solution to Distant Search. *Academy of Management Review*, 37(3):355–375, 2012.

[3] E. Baig. Quri App Lets You Make Money While You Shop. `http://content.usatoday.com/communities/technologylive/post/2012/05/quri-easyshift-shopper-app/1#.Vd6ybPnT4ds`, May 2012.

[4] D. C. Brabham. Crowdsourcing as a Model for Problem Solving an Introduction and Cases. *Convergence*, 14(1):75–90, 2008.

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

[6] K.-H. Dang and K.-T. Cao. Towards Reward-based Spatial Crowdsourcing. In *ICCAIS*, pages 363–368, 2013.

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

[8] P. Del Moral. Non-linear Filtering: Interacting Particle Resolution. *Markov Processes and Related Fields*, 2(4):555–581, 1996.

[9] P. Del Moral and L. Miclo. *Branching and Interacting Particle Systems Approximations of Feynman-Kac Formulae with Applications to Non-linear Filtering*. Springer, 2000.

[10] D. Deng, C. Shahabi, U. Demiryurek, and L. Zhu. Task selection in spatial crowdsourcing from workers perspective. *GeoInformatica*, 20(3):529–568, 2016.

[11] D. Deng, C. Shahabi, and L. Zhu. Task matching and scheduling for multiple workers in spatial crowdsourcing. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 21. ACM, 2015.

[12] M. Ester, H. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD*, pages 226–231, 1996.

[13] I. Fried. Gigwalk Allows iPhone Owners to Stumble Into Part-Time Work. http://allthingsd.com/20110504/gigwalk-allows-iphone-owners-to-stumble-into-part-time-work/, MAY 2011.

[14] H. Gao, G. Barbier, and R. Goolsby. Harnessing the Crowdsourcing Power of Social Media for Disaster Relief. *IEEE Intelligent Systems*, 26(3):10–14, 2011.

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

[16] M. F. Goodchild and J. A. Glennon. Crowdsourcing Geographic Information for Disaster Response: A Research Frontier. *Int. J. Digital Earth*, 3(3):231–241, 2010.

[17] N. J. Gordon, D. J. Salmond, and A. F. Smith. Novel Approach to Nonlinear/Non-Gaussian Bayesian State Estimation. In *IEE Proceedings F (Radar and Signal Processing)*, volume 140, pages 107–113. IET, 1993.

[18] E. Hamburger. How To Make Some Extra Cash Just By Taking Photos Around Town With Your iPhone. http://www.businessinsider.com/gigwalk-iphone-app-2011-5, May 2011.

[19] J. Howe. The Rise of Crowdsourcing. *Wired magazine*, 14(6):1–4, 2006.

[20] M. Hughes. What Is Uber And Why Is It Threatening Traditional Taxi Services? http://www.makeuseof.com/tag/uber-threatening-traditional-taxi-services/, July 2014.

[21] J. Jiang, B. An, Y. Jiang, P. Shi, Z. Bu, and J. Cao. Batch allocation for tasks with overlapping skill requirements in crowdsourcing. *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[22] Y. Jiang, Y. Zhou, and W. Wang. Task allocation for undependable multiagent systems in social networks. *IEEE Transactions on Parallel and Distributed Systems*, 24(8):1671–1681, 2013.

[23] L. Kazemi and C. Shahabi. GeoCrowd: Enabling Query Answering with Spatial Crowdsourcing. In *SIGSPATIAL*, pages 189–198, 2012.

[24] S. H. Kim, Y. Lu, G. Constantinou, C. Shahabi, G. Wang, and R. Zimmermann. MediaQ: Mobile Multimedia Management System. In *MMSys*, pages 224–235, 2014.

[25] A. Kittur, E. H. Chi, and B. Suh. Crowdsourcing User Studies with Mechanical Turk. In *CHI*, pages 453–456, 2008.

[26] A. Kittur, B. Smus, S. Khamkar, and R. E. Kraut. CrowdForge: crowdsourcing complex work. In *UIST*, pages 43–52, 2011.

[27] W. Li, H. Chen, W.-S. Ku, and X. Qin. 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*, page 63. ACM, 2017.

[28] W. Li, H. Chen, W.-S. Ku, and X. Qin. Turbo-gts: A fast framework of optimizing task throughput for large-scale mobile crowdsourcing. *ACM Transactions on Spatial Algorithms and Systems (TSAS)*, 6(1):1–29, 2020.

[29] A. Mathai, P. Moschopoulos, and G. Pederzoli. Random points associated with rectangles. *Rendiconti del Circolo Matematico di Palermo*, 48(1):163–190, 1999.

[30] R. Needleman. EasyShift Takes on GigWalk with New Task Marketplace. http://www.cnet.com/news/easyshift-takes-on-gigwalk-with-new-task-marketplace/, May 2012.

[31] E. Orhan. Particle Filtering. *Center for Neural Science, University of Rochester, Rochester, NY*, 8(11), 2012.

[32] A. Rai, K. K. Chintalapudi, V. N. Padmanabhan, and R. Sen. Zee: Zero-effort Crowdsourcing for Indoor Localization. In *Mobicom*, pages 293–304, 2012.

[33] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.

[34] A. S. Stordal. Sequential Monte Carlo Methods for Bayesian Filtering. 2008.

[35] M. Vukovic. Crowdsourcing for Enterprises. In *SERVICES*, pages 686–692, 2009.

[36] W. Wang and Y. Jiang. Community-aware task allocation for social networked multiagent systems. *IEEE transactions on cybernetics*, 44(9):1529–1543, 2014.

[37] P. Whitla. Crowdsourcing and its application in marketing activities. *Contemporary Management Research*, 5(1), 2009.

[38] D. Yang, D. Zhang, L. Chen, and B. Qu. NationTelescope: Monitoring and Visualizing Large-scale Collective Behavior in LBSNs. *J. Network and Computer Applications*, 55:170–180, 2015.

[39] D. Yang, D. Zhang, and B. Qu. Participatory Cultural Mapping Based on Collective Behavior Data in Location-Based Social Networks. *ACM TIST*, 7(3):30:1–30:23, 2016.

[40] D. Yang, D. Zhang, V. W. Zheng, and Z. Yu. Modeling User Activity Preference by Leveraging User Spatial Temporal Characteristics in LBSNs. *IEEE Trans. Systems, Man, and Cybernetics: Systems*, 45(1):129–142, 2015.

[41] M. Zook, M. Graham, T. Shelton, and S. Gorman. Volunteered Geographic Information and Crowdsourcing Disaster Relief: A Case Study of the Haitian Earthquake. *World Medical & Health Policy*, 2(2):7–33, 2010.