Multi-Threaded Dynamic Taxi Sharing with Meeting Points

Bachelor's Thesis of

Ha Linh Nguyen

At the KIT Department of Informatics
Institute of Theoretical Informatics, Algorithm Engineering

First examiner:	Prof. Dr. Peter Sanders
Second examiner:	T.-T. Prof. Dr. Thomas Bläsius
First advisor:	Moritz Laupichler, M. Sc.

01. December 2023 – 02. April 2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Karlsruhe, 2nd April 2024

. . . .

(Ha Linh Nguyen)

Abstract

Transportation and its associated challenges remain a pressing concern, particularly in densely populated urban areas. Taxi sharing services have emerged as a potential solution to alleviate traffic congestion by enabling multiple passengers to share a vehicle. However, scaling these services to accommodate a growing user base presents the new challenge of efficiently assigning passengers to vehicles among the multiple simultaneous incoming requests.

We introduce a parallel algorithm for the dynamic taxi sharing problem with meeting points – an algorithm for scheduling a fleet of shared vehicles. Our algorithm considers each incoming request dynamically, as used by taxi sharing services such as *Uber* or *Lyft*. Building upon an existing sequential dispatching algorithm, we have adopted and optimized it for parallel execution across multiple threads. Our adjustments enable parallel processing in every phase of the original sequential algorithm, with additional utilization of vector instructions in bundled searches to leverage instruction-level parallelism.

Through empirical evaluation, our parallel algorithm demonstrates improved computational efficiency in matching passengers with vehicles compared to its sequential counterpart, particularly noticeable in scenarios involving numerous meeting points.

Zusammenfassung

Der Verkehr und die damit verbundenen Herausforderungen sind nach wie vor ein dringendes Problem, insbesondere in größeren Städten mit hoher Bevölkerungsdichte. Taxi-Sharing-Dienste haben sich als potenzielle Lösung zur Entlastung des Verkehrs entwickelt, da sie es mehreren Fahrgästen ermöglichen, sich ein einziges Fahrzeug zu teilen. Dies kann ein zusätzliches Problem mit sich bringen: Wenn die Zahl der Nutzer steigt, muss der Algorithmus, der die Fahrgäste dem passenden Fahrzeug zuordnet, in der Lage sein, zahlreiche eingehende Anfragen gleichzeitig und sofort zu bearbeiten.

Wir stellen einen parallelen Algorithmus für das dynamische Taxi-Sharing-Problem mit Treffpunkten vor – einen Algorithmus zur dynamischen Planung einer Flotte von gemeinsam genutzten Fahrzeugen. Wir betrachten die eingegangene Anfrage dynamisch, wie von Taxi-Sharing-Diensten wie *Uber* oder *Lyft* verwendet wird. Aufbauend auf einem sequenziellen Dispatching Algorithmus, haben wir ihn für die parallele Ausführung über mehrere Threads übernommen und optimiert. Unsere Anpassungen ermöglichen die parallele Verarbeitung in jeder Phase des ursprünglichen sequenziellen Algorithmus, mit zusätzlicher Nutzung von Vektoranweisungen bei gebündelten Suchen, um Parallelität auf Instruktionssebene zu nutzen.

Durch empirische Evaluierung zeigt unser paralleler Algorithmus eine verbesserte Rechenleistung bei der Zuordnung von Passagieren zu Fahrzeugen im Vergleich zu dem sequenziellen Algorithmus, insbesondere in Szenarien mit zahlreichen Treffpunkten.

Contents

Abstract i

Zusammenfassung iii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem . . . 1
- 1.3 Contributions . . . 2

2 Problem Statement 3
- 2.1 Definitions and Notations . . . 3
- 2.2 Cost Function and Constraints . . . 4

3 Related Work 7
- 3.1 Taxi Sharing and Related Problems . . . 7
- 3.2 Computing Shortest Path Distances . . . 8
- 3.3 Parallelization in Taxi Sharing Problems . . . 10

4 Background 13
- 4.1 Shortest Path Algorithms . . . 13
  - 4.1.1 Dijkstra's Algorithm . . . 13
  - 4.1.2 Contraction Hierarchies . . . 14
  - 4.1.3 Bundling Searches . . . 16
- 4.2 Dynamic Taxi Sharing Algorithms . . . 16
  - 4.2.1 LOUD . . . 16
  - 4.2.2 KaRRi . . . 18

5 Parallel Algorithm 23
- 5.1 Elliptic BCH Searches . . . 23
- 5.2 Pickup and Dropoff After Last Stop . . . 24
  - 5.2.1 Last Stop BCH Searches . . . 25
  - 5.2.2 Enumerating Assignments . . . 25
- 5.3 Pickup Before Next Stop . . . 26
  - 5.3.1 Finding Distances Between Current Vehicle's Location and Pickup 26
  - 5.3.2 Enumerating Assignments . . . 27
- 5.4 Pickup-Dropoff Searches . . . 28
  - 5.4.1 BCH Strategy . . . 28
  - 5.4.2 CH Strategy . . . 29

6 Implementation Details **31**
6.1 Thread Local Storage . . . 31
6.2 Allocating Query Results . . . 31
6.2.1 Static Allocation . . . 33
6.2.2 Semi-Static Allocation . . . 34
6.2.3 Dynamic Allocation . . . 34
6.3 Accessing and Updating Minimum Distances . . . 36
6.4 Thread-Safe Subset . . . 37
6.5 Updating Global Best Assignment . . . 37
6.6 Collecting Statistics . . . 38

7 Evaluation **39**
7.1 Parameter Tuning . . . 40
7.1.1 Elliptic BCH Searches . . . 40
7.1.2 PALS and DALS . . . 42
7.1.3 PBNS . . . 45
7.1.4 PD-Distance Searches . . . 45
7.2 Scalability . . . 47
7.2.1 Strong Scaling . . . 47
7.2.2 Weak Scaling . . . 48
7.3 Comparison to KaRRi . . . 51
7.4 Discussion . . . 53

8 Conclusion **57**

Bibliography **59**

List of Figures

4.1 Dijkstra search spaces in comparison. The search space of the unidirectional Dijkstra search is twice as large as the search space of the bidirectional Dijkstra search. 14

6.1 Example for dynamic allocation. s, s_1, s_2, s_3 are IDs of stops and p is the ID of a pickup. N_{stops} denotes the total number of stops currently scheduled for all vehicles. $\perp$ represents an invalid index value. 32

6.2 Example for static allocation. s is the ID of a stop and p is the ID of a pickup. 33

7.1 Runtimes for phases of elliptic BCH searches (initialization, pickup, dropoff) with different allocation types (static, semi-static, dynamic) and maximum number of threads ($T \in \{8, 16\}$). Runtime without SIMD instructions (top), runtime with SIMD instructions, $K = 16$ (bottom). Experimented on `Berlin-1pct` with $\rho = 600s$. Note the different y-axes. 41

7.2 Speedup for elliptic BCH searches with dynamic allocation on different number of threads ($T \in \{8, 16\}$) when using SIMD instructions ($K \in \{1, 8, 16, 32\}$). Experimented on `Berlin-1pct` with $\rho = 600s$. 42

7.3 Sum of thread local runtimes (search time, try assignments time) for PALS and DALS assignments without SIMD instructions and with SIMD instructions ($K = 16$). Experimented on `Berlin-1pct` with $\rho = 600s$ and $T = 8$. 43

7.4 Speedup for PALS (left) and DALS (right) assignments on different number of threads ($T \in \{8, 16\}$) when using SIMD instructions ($K \in \{1, 8, 16, 32\}$). Experimented on `Berlin-1pct` with $\rho = 600s$ (top) and $\rho = 900s$ (bottom). Note the different y-axes. 44

7.5 Speedup for PD-distances with BCH Searches on different number of threads ($T \in \{8, 16\}$) when using SIMD instructions ($K \in \{1, 8, 16, 32\}$). Experimented on `Berlin-1pct` with $\rho = 600s$. 46

7.6 Total runtimes with different number of threads ($T \in \{1, 8, 16, 32, 64\}$). This plot does not include outliers, maximum runtime for each number of thread written at the top of the corresponding box. Experimented on `Berlin-10pct` with $\rho = 900s$ and without SIMD instructions. 48

7.7 Speedup for PALS assignments in relation to the number of pickups (N_ρ^p) (top left). Speedup for DALS assignments in relation to the number of dropoffs (N_ρ^d) (bottom left). Distribution of requests over N_ρ^p (top right) and over N_ρ^d (bottom right), next to the corresponding speedup plot. Experimented on `Berlin-10pct` with $\rho = 900s$, without SIMD instructions and on different number of threads ($T \in \{8, 16, 32, 64\}$). Note the different y-axes. 49
7.8 Speedup for elliptic BCH searches in relation to the sum of pickups and dropoffs ($N_\rho^p + N_\rho^d$) (top left). Speedup for PD-distance searches in relation to the product of pickups and dropoffs ($N_\rho^p \cdot N_\rho^d$) (bottom left). Distribution of requests over $N_\rho^p + N_\rho^d$ (top right) and over $N_\rho^p \cdot N_\rho^d$ (bottom right), next to the corresponding speedup plot. Experimented on `Berlin-10pct` with $\rho = 900s$, without SIMD instructions and on different number of threads ($T \in \{8, 16, 32, 64\}$). Note the different y-axes. 50

List of Tables

7.1 Key figures of our benchmark instances. Shows number of vertices ($|V|$), edges ($|E|$), vehicles (#veh.), and requests (#req.). In addition, shows average number (rounded down) of pickups (N_ρ^p) and dropoffs (N_ρ^d) for walking radius $\rho \in \{300s, 600s, 900s\}$ on the `Berlin-1pct`, `Berlin-10pct`, `Ruhr-1pct` and `Ruhr-10pct` instances. 40

7.2 Comparison of the PD-distance searches running times (in μs) with different maximum number of threads ($T \in \{8, 16\}$) using BCH searches (BCH) and CH searches (CH), with and without SIMD instructions. Shows average times for initialization phase (t_{init}), preprocessing phase (dropoff bucket generation in BCH searches) ($t_{preprocessing}$) and search phase (t_{search}). 46

7.3 Geometric mean speedups on the `Berlin-10pct` instance with $\rho = 900s$ for the different phases, including elliptic BCH searches (ell. BCH), PD-distance searches (PD), PALS assignments (PALS), DALS asssignments (DALS) and for the total runtime of the algorithm (total). 48

7.4 Running times (in μs) of different phases of the KaRRi algorithm (Ka), parallel algorithm (P), and combination of KaRRi and parallel algorithm (Ka + P) with different radii ($\rho \in \{300s, 600s, 900s\}$) on `B-1%`, `B-10%`, `R-1%`, and `R-10%`. Shows mean times for initialization, PD-distance searches, elliptic BCH searches, enumerating ordinary insertions, enumerating PBNS insertions, PALS and DALS searches as well as the mean total time per request. The smallest times per radius are marked in bold. 54

7.4 Running times (in μs) of different phases of the KaRRi algorithm (Ka), parallel algorithm (P), and combination of KaRRi and parallel algorithm (Ka + P) with different radii ($\rho \in \{300s, 600s, 900s\}$) on `B-1%`, `B-10%`, `R-1%`, and `R-10%`. Shows mean times for initialization, PD-distance searches, elliptic BCH searches, enumerating ordinary insertions, enumerating PBNS insertions, PALS and DALS searches as well as the mean total time per request. The smallest times per radius are marked in bold. 55

1 Introduction

1.1 Motivation

In today's developing world, the traffic problem has grown significantly, becoming increasingly serious for daily commuters. This includes traffic congestion, delays, and inefficiencies in transportation systems, all of which impact the travel time of vehicles and riders [30]. As cities grow in population, this problem increases in severity [38, 39]. Since personal transportation still relies mainly on individual vehicles and inconvenient public transport, the problem of traffic jams at peak hours remains unsolved [19, 42]. Transportation with individual cars often consumes a lot of space and energy because there are on average few people in each car [3, 37].

Taxi sharing, in which multiple passengers with various origins and destinations share a vehicle, offers a more environmentally sustainable alternative [46]. Not only does this alternative utilize the vacant seats on vehicles on the road, but it also reduces the number of vehicles participating in traffic, thus decreasing the total emissions and the probability of traffic congestion [1, 46]. Taxi sharing services, such as Uber and Lyft, facilitate this process by providing platforms for passengers to connect with nearby drivers willing to accommodate their rides [2, 5, 11]. These services utilize algorithms to match passengers with compatible routes and optimize the allocation of vehicles, offering passengers a more cost-effective alternative to traditional taxi services. We consider here, in particular, the variant of taxi sharing with meeting points for the vehicles and the passengers [4, 25, 40].

1.2 Problem

The taxi sharing problem involves efficiently matching multiple passengers with similar origins or destinations to a fleet of vehicles to minimize travel time, cost, or environmental impact [2, 40]. The basic objective of taxi sharing is to maximize the efficient usage of transportation resources while offering customers convenient and economically viable travel alternatives. To enhance the customer experience, sharing services must minimize customer wait times and avoid significant detours for pickups and dropoffs, since these detours affect the trip times of passengers who are already in the car [11, 45]. Therefore, an efficient algorithm is crucial for coordinating incoming requests with the existing fleet of vehicles [1, 2]. This involves effectively making a complex decision as to which vehicle and where, in its route, a request should be assigned. With the introduction of meeting points, there are two additional degrees of freedom to consider, namely the pickup and dropoff locations [40].

The frequency of requests increases with a rising demand for taxi sharing services. The system should thus be able to handle these requests promptly to keep up with the demand [1]. On the one hand, most dispatching algorithms calculate the shortest path distances in advance and use them when assigning requests to speed up the process [21, 34, 14, 25, 17]. On the other hand, there are existing online dynamic taxi sharing algorithms that calculate these distances while dispatching to increase the flexibility of the algorithm. Our work builds upon two such dispatching algorithms, namely LOUD and KaRRi [6, 23]. LOUD utilizes the problem structure to speed up the shortest path searches during the dispatching phase. KaRRi extends and complements these techniques for the case with meeting points. Nonetheless, LOUD and KaRRi might still not be fast enough for large road networks and a high number of vehicles and requests, as they run sequentially.

1.3 Contributions

Our focus lies in parallelizing the existing dynamic taxi sharing algorithm with meeting points – KaRRi [23]. Specifically, we concentrate on parallelizing the algorithm across the relevant meeting points of a request. This includes selecting vehicles with sufficient capacity and minimal detours, as well as determining suitable pickup and dropoff locations from the set of relevant meeting points. We parallelize all shortest path searches that calculate the necessary distances between vehicle stops and meeting points in the original algorithm. Additionally, we consider different allocation methods to make the searches more memory- and cache-efficient. For specific phases of the algorithm, we also transform the process of enumerating the assignments to run in parallel on multiple threads. Lastly, we adjust the parallel searches to be able to execute in batches, allowing us to exploit single instructions with multiple data (SIMD) parallelism and thread parallelism at the same time.

We provide a comprehensive overview of our work, as follows: In Chapter 2, we delve into the problem statement, providing formal definitions and notations essential for understanding dynamic taxi sharing. Chapter 3 offers a thorough review of related literature, exploring topics such as challenges in taxi sharing, methodologies for computing shortest paths, and parallelization strategies. Based on this groundwork, Chapter 4 introduces the required background information, explaining fundamental concepts like shortest path algorithms and dynamic taxi sharing algorithms, with a specific focus on the two dispatching algorithms, LOUD and KaRRi. The main contribution of our research can be found in Chapter 5, where we describe the high-level parallel algorithm. Here, we discuss the individual phases of the parallel algorithm. Chapter 6 clarifies the important implementation details of our algorithm, with key considerations such as thread local storage and query result allocation. Finally, in Chapter 7, we conduct various experiments to assess the performance of the parallel algorithm, including comparisons with the original KaRRi algorithm.

2 Problem Statement

This segment outlines and provides the theoretical basis for the dynamic taxi sharing problem, which has been adopted from the notations used in LOUD and KaRRi [6, 23].

2.1 Definitions and Notations

Road Network. A road network is considered to be a directed graph $G = (V, E)$, with road intersections as vertices in V and road segments as edges in E. For each edge $e = (v, w) \in E$, a travel time between the two intersections is defined as $\ell(e) = \ell(v, w)$. In addition, the shortest path distance from node v to node w can be described with $dist(v, w)$. As in KaRRi, we use two separate road networks G_{veh} and G_{psg} to model roads and paths accessible to taxi vehicles and passengers, respectively. The associated edge travel times and shortest path distances are denoted as ℓ_{veh}, ℓ_{psg}, $dist_{veh}$, $dist_{psg}$.

Vehicle, Stop. The algorithm keeps track of a fleet F of vehicles. Each vehicle ν is defined as $\nu = (l_i, c, t_{serv}^{min}, t_{serv}^{max})$ with an initial location of l_i in the road network G_{veh} and a capacity of c passengers that the vehicle can carry at once. The remaining two parameters represent the service time of the taxi vehicle, which operates in the time interval of $[t_{serv}^{min}, t_{serv}^{max})$. A current route of each vehicle is described as $R(\nu) = \langle s_0, ..., s_{k(\nu)} \rangle$ with stops s_i for $0 \leq i \leq k(\nu)$ and associated stop locations $l(s_i) \in V$. We may write stops as vertices in short, for example, s_i instead of $l(s_i)$ and $dist(s_i, s_j)$ instead of $dist(l(s_i), l(s_j))$. The vehicle is always located between stops s_0 and s_1. More specifically, if the vehicle is currently idle, it is stopping at s_0 and if the vehicle is moving, it is traveling from stop s_0 to s_1. Therefore, $k(\nu)$ represents the number of stops the vehicle has yet to visit. Every vehicle's route is updated accordingly when the vehicle reaches its upcoming stop or is assigned new stops.

Request. In our scenario, the dispatching algorithm receives incoming ride requests from passengers and has to instantly match them to a feasible vehicle in the fleet. An incoming request can be denoted as $r = (orig, dest, t_{req})$, where $orig$ is the original location, $dest$ is the desired destination of the rider, and t_{req} is the time when the request is submitted. Since we do not allow pre-booking, t_{req} is also the earliest departure time of the request $t_{dep}^{min}(r)$.

Meeting Points. KaRRi considers an expanded definition of taxi sharing that includes meeting spots, also known as pickup and dropoff locations (pickups and dropoffs, in short). In our work, we will refer to any means of local transportation for riders as *walking* for simplicity. Hence, a passenger can also be referred to as a pedestrian. Here, the passenger road network $G_{psg} = (V_{psg}, E_{psg})$ is utilized to find the shortest path distance from the passenger's location to the meeting points. The set of possible pickup locations for request r is made up of all the vertices $v \in V_{veh} \cap V_{psg}$ that can be reached from the request's origin in G_{psg} within time radius ρ. Similarly, the set of possible dropoff locations for request r is made up of all the vertices $v \in V_{veh} \cap V_{psg}$ that can be reached from the request's destination in G_{psg} within time radius ρ. Formally, these two sets are denoted as:

$$P_\rho(r) := \{p \in V_{veh} \cap V_{psg} \mid dist_{psg}(orig, p) \leq \rho\}$$
$$D_\rho(r) := \{d \in V_{veh} \cap V_{psg} \mid dist_{psg}(d, dest) \leq \rho\}$$

We define the number of relevant pickups and relevant dropoffs for a request r as $N_\rho^p(r) := |P_\rho(r)|$ and $N_\rho^d(r) := |D_\rho(r)|$, respectively. These notations can be shortened to P_ρ, D_ρ, N_ρ^p, and N_ρ^d when the context is clear enough. Pickups and dropoffs are referred to together as *PD-locations*, and a pair of pickup and dropoff is called a *PD-pair*. The shortest path distance between a pickup and dropoff is called *PD-distance*.

Insertion. The task of the algorithm is to find a feasible insertion of a PD-pair into an existing vehicle's route for each incoming request. A so-called insertion of a request can be represented by a 6-tuple $\iota = (r, p, d, v, i, j)$, which describes the scenario where the vehicle v picks up the passenger at pickup location $p \in P_\rho(r)$ immediately after stop s_i in its route and drops off the passenger at dropoff location $d \in D_\rho(r)$ immediately after stop s_j with $0 \leq i \leq j \leq k(v)$.

2.2 Cost Function and Constraints

Cost Function. For each request r, our algorithm searches for a feasible insertion ι with minimal cost according to a cost function $c(\iota)$. The cost function is given as follows:

$$\begin{aligned} c(\iota) = t_{detour}(\iota) + \tau \cdot (t_{trip}(\iota) + t^+_{trip}(\iota)) \\ + \omega \cdot t_{walk}(\iota) + c^{vio}_{wait}(\iota) + c^{vio}_{trip}(\iota) \end{aligned} \tag{2.1}$$

This is a sum of the added vehicle operation time $t_{detour}(\iota)$, the trip time of the insertion $t_{trip}(\iota)$, the total increased trip times of other passengers on the shared vehicle $t^+_{trip}(\iota)$, the walking time of the passenger to the PD-locations $t_{walk}(\iota)$ and the penalties for violating the time constraints $c^{vio}_{wait}(\iota) + c^{vio}_{trip}(\iota)$.

The detour time of the insertion $t_{detour}(\iota)$ presents the time which the vehicle needs to accommodate pickup p and dropoff d in its route. The trip time $t_{trip}(\iota)$ is the time between the submission of the request $t_{req}(r)$ and the arrival of the rider at their desired destination

$dest(r)$. This includes the wait time as well as the walking time. Since picking up and dropping off new passengers may delay the trip time of other previously assigned riders on the same vehicle, the sum of the increases in trip time for all affected riders $t^{+}_{trip}(\iota)$ must be taken into consideration. Finally, the walking time of the passenger in G_{psg} from their original location $orig(r)$ to the pickup p and from the dropoff d to their desired destination $dest(r)$ is also added. Note that the cost function is defined with two model parameters, τ and ω, in order to weigh the importance of the time parameters based on the needs of different services. For example, when the passenger walking time is considered essential, the model parameter ω should then be increased. It is also known that in the case of $\tau = \omega = 0$, the situation is identical to that considered by LOUD, where the passengers are picked up and dropped off directly at their desired locations without having to travel to any given meeting points.

Constraints. There are four constraints that have to be examined when checking the feasibility of an insertion. The capacity constraint ensures that the occupancy of a vehicle remains less than or equal to its capacity at all times. The end-of-service time constraint guarantees that the vehicle reaches its last stop within its service time interval $[t^{min}_{serv}, t^{max}_{serv})$. The remaining two constraints make sure that for every request already assigned to vehicle v, the passengers are picked up within the maximum wait time t^{max}_{wait} and dropped off within the maximum trip time t^{max}_{trip}.

The capacity and service time constraints are always hard constraints, which means that the cost is set to ∞ if they are broken. The maximum wait time and maximum trip time constraints are considered hard constraints only for existing passengers who have already been assigned to a vehicle v beforehand. While checking an insertion ι for a request r yet to be inserted into the vehicle's route, the wait and trip time constraints are considered soft constraints. As a consequence, violating these soft constraints will increase the cost function formalized in Equation (2.1). The cost penalties can be formally defined as:

$$\begin{aligned} c^{vio}_{wait}(\iota) &= \gamma_{wait} \cdot max\{t_{dep}(\iota) - t_{req}(r) - t^{max}_{wait}, 0\} \\ c^{vio}_{trip}(\iota) &= \gamma_{trip} \cdot max\{t_{trip}(\iota) - t^{max}_{trip}, 0\} \end{aligned} \tag{2.2}$$

The model parameters γ_{wait} and γ_{trip} are used to modify the severity of the violation penalties.

3 Related Work

Research on taxi sharing, a more sustainable alternative to private cars, has been extensive and diverse. Taxi sharing comes in various forms, each with its own set of objectives, leading to different approaches for tackling the issue. In this chapter, we give an overview of the taxi sharing problem and its many related problems. Additionally, we consider the important sub-problem of computing the shortest paths in road networks, with a focus on routing taxi sharing vehicles. Finally, we explain how existing approaches to the taxi sharing problem exploit parallelism.

3.1 Taxi Sharing and Related Problems

The taxi sharing problem involves the efficient matching of multiple passengers who have different starting points or destinations into shared rides provided by a fleet of vehicles. Key objectives of this problem include minimizing travel duration, maximizing vehicle occupancy, and reducing expenses [1].

Taxi sharing is a special case of the dial-a-ride problem (DARP). In this problem, a fleet of vehicles is tasked with serving a set of pickup and delivery requests while holding constraints such as vehicle capacity, time windows, and passenger preferences. There are two main variants of DARP and taxi sharing, namely static and dynamic [8, 16]. In the static variant, all requests are known in advance, with pre-planned routes and limited real-time adjustments. This variant is often the standard assumption in many studies and is proven to be NP-hard [43]. When addressing the static variant of the DARP, several common approaches are typically employed. These include heuristic algorithms, which, for instance, utilize insertion heuristics [21, 44] or regret-based heuristics [9]. Additionally, metaheuristic algorithms, such as simulated annealing [24], exact algorithms using integer linear programming for smaller instances [35], and hybrid approaches that combine heuristics and exact methods [10], are also commonly applied.

We consider the dynamic variant of the taxi sharing problem, where the requests arrive gradually, requiring real-time adjustments and adaptive routing [8]. Many studies on dynamic taxi sharing primarily concentrate on identifying assignments and evaluating their feasibility considering the time constraints of the passengers [28, 31]. In order to accomplish this, the algorithm must be aware of the vehicle detours taken to accommodate any additional passengers. Frequently, it is assumed that these detours have been given in advance [14, 18, 21]. Nevertheless, determining the shortest paths that constitute the detours in the road network presents a significant time burden and might hinder the effectiveness of a taxi sharing dispatcher [23].

3.2 Computing Shortest Path Distances

As the overhead of the shortest path distance computation can account for the bottleneck of the taxi sharing algorithm's total runtime, different approaches have been proposed to speed up this computation.

In Preprocessing. Many studies consider the computation of shortest path distances as part of the preprocessing phase. Ota et al. propose a scalable taxi sharing simulation model that is capable of simulating different taxi sharing variants [34]. The simulation model consists of a taxi fleet, passengers, a scheduler, trip data, and the road network. The overall goal remains fixed, which is to minimize the total cost given through a cost function or to maximize the sharing while holding a set of constraints. Their paper does not consider the shortest path algorithm in much detail but refers to it as a preprocessing step. The Dijkstra algorithm computes all-pair shortest path distances between the intersections in the road network during the preprocessing phase. These shortest path distances can be queried in the course of the simulation later on.

The dynamic taxi sharing algorithm T-Share aims to generate ride-sharing schedules that reduce the total travel distances of the vehicles [27]. This approach uses pre-computed distances to estimate the shortest paths. T-Share partitions the road network into a grid and finds distances between the centers of grid cells, which are written into the so-called *grid distance matrix*. During the simulation, these distances are used to approximate shortest path distances and apply lazy shortest path calculation. Rather than immediately computing the shortest path, the technique initially determines the minimum trip time between grid cells by utilizing the pre-computed distances and the triangle inequality. This acts as a lower bound for the exact shortest path distance, and only if this lower bound satisfies all given constraints is the exact shortest path distance calculated.

Querying shortest path distances during the simulation phase becomes challenging on larger road networks due to high memory usage. To address this issue, Mounesan et al. propose a method that significantly reduces memory consumption while sacrificing minimal query time, called the shortest path caching scheme [33]. This scheme focuses on storing subsets of shortest paths in a compact manner to reduce memory requirements. Instead of storing all shortest paths, the scheme identifies common subpaths for subsets of the shortest paths, leveraging the concept that paths from a common source often share a longer common path before diverging to reach their respective destinations.

The remaining problem of computing shortest path distances in advance is the significant time required for preprocessing and the constantly changing travel times in the road network [26]. As a consequence, precomputed shortest path distances are only valid for a limited time window and need to be recomputed often, which can be expensive. In some cases, it can even occur that the shortest past distances are out-of-date, right after the preprocessing phase.

In Simulation Some dynamic taxi sharing algorithms apply state-of-the-art shortest path algorithms to find the required distances during the dispatching process. The efficiency of the techniques applied is a deciding factor in making this on-the-fly computation of shortest paths viable. On-the-fly computation allows for the integration of real-time travel times into the road network, which change regularly due to factors like congestion, accidents, and other variables. LOUD and KaRRi [6, 23] are examples of such dynamic taxi sharing dispatching algorithms that apply shortest path algorithm based on contraction hierarchy searches [12, 22]. As our work is based on these algorithms, we only give a brief overview here and explain the approaches more thoroughly in Section 4.2.

LOUD [6] is an algorithm for the dispatching problem that utilizes bucket-based contraction hierarchies for an on-the-fly computation of the shortest paths. An essential contribution of LOUD is the idea of elliptic pruning. This technique prunes the search space of the required BCH searches for each request using the time constraints of previously assigned passengers [6]. This allows the shortest path algorithm itself to act as a filter for feasible assignments.

The KaRRi taxi sharing dispatcher [23] extends LOUD with additional adjustments, the most significant of which is the introduction of meeting points. Meeting points are points located near the passengers' origin or destination, where the passengers may meet up with the vehicle [23]. These meeting points introduce many-to-many shortest path problems, for which KaRRi must find efficient solutions. In addition to finding the best vehicle for a request, KaRRi has to find the optimal pickup and dropoff location. As KaRRi continues to use bucked-based contraction hierarchies similarly to LOUD, the algorithm implements numerous many-to-many routing techniques to speed up the original queries when applied to multiple pickups and dropoffs.

Parallel Computation. Different research has shown that parallelizing shortest path algorithms may not be simple, despite the strong desire to speed up this fundamental problem. Dijkstra's algorithm, for instance, is difficult to parallelize as the order in which it processes vertices is based on a priority queue [20]. Here, the all-to-all shortest path problem is considered. In Dijkstra's algorithm, the head of queue element is considered one-by-one, and during this process, the priorities of other elements in the queue may change. Thus, it is not correct to consider multiple elements in an iteration other than the head of queue element. Jasika et al. develop two parallelization strategies for Dijkstra's algorithm, namely *source partitioned* and *source parallel* [20]. Source partitioned involves assigning a one-to-one shortest path problem to individual processor. Each processor then calculates the distance from a shared source to a different target. In contrast, source parallel utilizes a parallel formation of the shortest path problem to boost concurrency. This entails sequentially running Dijkstra's one-to-many shortest path algorithm on each processor, starting from a designated source to all other vertices. The experiments have shown that the parallel approach did not result in significant speedups. This might be associated with the limited number of nodes since parallelization and synchronization could take up more time than actually running the function.

The Δ-stepping algorithm is a parallel one-to-all shortest path algorithm [32]. The algorithm utilizes a bucket array as its primary data structure to store vertices along with their found distances. There are three main phases: bucket expansion, reinsertion, and termination. The algorithm progresses by expanding buckets and processing vertices in each bucket. This involves relaxing the edges of the vertices in the bucket and updating the corresponding buckets based on the relaxed distances. The updated distances may require the reinsertion of some vertices into earlier buckets. The algorithm iteratively repeats these steps until it determines the shortest path to all vertices. Here, fine-grained parallelism is exploited by allowing multiple edge relaxations to be performed simultaneously. This means that individual tentative distance updates can be done in parallel. This allows the algorithm to relax light edges (with weights less than or equal to a threshold Δ) and perform the resulting updates in parallel. The efficiency of the Δ-stepping technique in addressing the single source shortest path problem on large networks has been shown on large graph instances [29]. Here, the algorithm demonstrates significant parallel speedup, scalability, and near-linear performance.

3.3 Parallelization in Taxi Sharing Problems

A few parallel approaches have been made towards the taxi sharing problem. However, these mainly focus on parallelizing the processing over geographical subproblems. Zhang et al. introduce the parallel clustering simulated annealing (PCSA) algorithm, an efficient parallel heuristic method to address the global optimization problem associated with ridesharing systems [47]. Simulated annealing (SA) is a probabilistic optimization algorithm. Here, requests for a certain time window are gathered and scheduled at the same time in batches. The PCSA method employs domain decomposition, clustering, and merging for parallel execution. Domain decomposition divides the problem into smaller subdomains by separating the requests and vehicles by geographical locations, while merging combines subdomains with low acceptance rates of new solutions. Clustering groups pickup points and vehicle positions into clusters for parallel processing. The algorithm achieves a significant speedup when compared to other state-of-the-art SA algorithms.

Xue et al. present a similar attempt to parallelize the dynamic taxi sharing problem [45]. The authors of this work consider a multi-objective problem, which involves different perceived values of passengers that have to be optimized simultaneously. Perceived value is introduced as the assessed advantage of the taxi sharing trip for the individual passenger. Due to the complexity of this problem, the authors refrain from parallelizing the actual dispatching process. Instead, they opt to partition the geographical area into different operating regions so that the taxi dispatching problem can be solved simultaneously for each region. As different threads process the partitioned regions, it is important to ensure that the regions all have similar request numbers so that the threads do not have to wait for one another. After the parallel execution, a rebalancing method is carried out with the accumulated data from all different regions. The authors construct this rebalancing method to effectively balance vehicle supply and demand by relocating idle vehicles.

Overall, the exploration of parallelism in taxi sharing problems has primarily focused on partition-based approaches, with limited attention given to parallelizing the dispatching algorithm itself. This research gap underscores the need for additional study into how parallelization strategies might improve the dispatching algorithms' efficiency. In our study, we aim to bridge this gap by implementing fine-grained parallelism in KaRRi, a state-of-the-art dispatching algorithm that incorporates meeting points.

4 Background

This chapter aims to introduce the required background knowledge regarding our topic of research, which is dynamic taxi sharing. We will introduce and explain some algorithms that we employ in our work and discuss the contributions of previous works in the field.

4.1 Shortest Path Algorithms

Since the underlying problem to be solved is the shortest path problem, and bucket contraction hierarchies serve as foundational components of LOUD [6] and KaRRi [23], we will introduce these required preliminaries in the following subsections. First, we will explain Dijkstra's algorithm in its base form along with the bidirectional form. Next, we will discuss contraction hierarchies (CH), a speedup technique for shortest path computations in road networks [12]. Along with standard CH, we also describe bucket-based CH in the following.

4.1.1 Dijkstra's Algorithm

Single-source shortest path problem. Dijkstra's algorithm is one of the most well-known and widely used shortest path algorithms. Given a weighted graph $G = (V, E)$ with edges that have non-negative weights defined by the function $w : E \longrightarrow \mathbb{R}_{\geq 0}$, the algorithm searches for shortest paths from a given source node $s \in V$ to all vertices $v \in V$. Dijkstra's algorithm grows a shortest path tree from the source node s.

The algorithm uses a priority queue Q to keep track of the different nodes and their found tentative distance $d_s(v)$ from s. Initially, all nodes except for s itself have the distance ∞. The priority queue only contains the source vertex s at first. Then, the nodes are sequentially taken out of the queue in increasing order of the found distance, and are *settled* one-by-one. Settling a node u means considering all outgoing edges (u, v) from u and checking whether the distance over u: $d_s(u) + w(u, v)$ is better than the currently stored distance $d_s(v)$ of v. This process of trying to improve the distance with a specific edge is called *relaxing* the edge. If $d_s(u) + w(u, v) < d_s(v)$, the node v is updated to the queue with the new found distance, and the algorithm carries on with the next head of queue node. Once the queue is empty, all required distances have been found. Nodes in the graph are either *reached*, *unreached* or *settled* [12]. A *settled* node u is a node that is already included in the shortest path tree, whose shortest path from s has been found. A

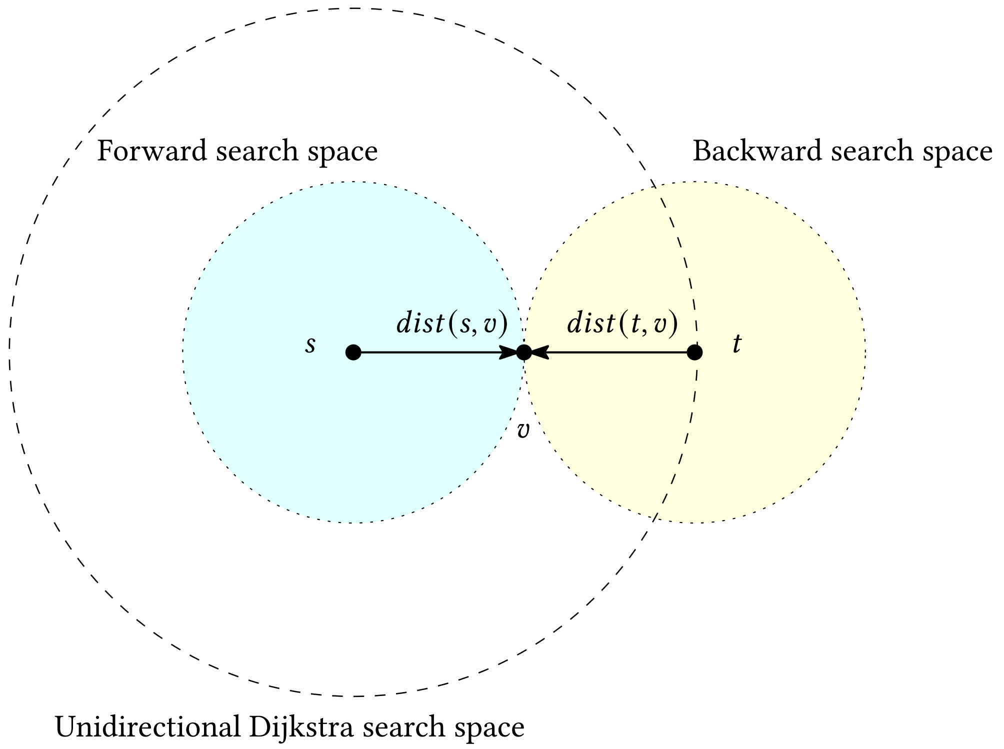

Figure 4.1: Dijkstra search spaces in comparison. The search space of the unidirectional Dijkstra search is twice as large as the search space of the bidirectional Dijkstra search.

node u is *reached* if it is adjacent to a settled node v but not yet settled itself. Nodes that are neither settled nor reached are considered *unreached*.

Bidirectional Dijkstra. Given a specific source node s and a specific target node t, it is possible to run a bidirectional Dijkstra search. This can be done by running a forward search from s and a backward search from t. Once a node $v \in V$ is settled by both searches, we have found the shortest path distance between s and t. While an unidirectional search has to search a radius of $dist(s, t)$, the forward (backward) search only searches a radius of $dist(s, v)$ ($dist(t, v)$). If we imagine the search space of the original Dijkstra search as a ball of radius $dist(s, t)$, we can reduce its surface area by half, since the forward and backward search space each has a radius of $\frac{dist(s,t)}{2}$ [36]. This is visualized in Figure 4.1. The surface area is an indication of the computational cost of the searches.

4.1.2 Contraction Hierarchies

Standard Contraction Hierarchies. The fact that road networks do not change much over time suggests a benefit if we preprocess the graph beforehand in order to speed up the shortest path queries. One such preprocessing technique is the construction of a CH of the graph [12]. To speed up the process of finding the shortest paths, we add shortcut edges to the graph that take advantage of the hierarchy of road networks. This hierarchy can be given through a heuristic for the nodes' importance. Nodes in the original graph are contracted individually in ascending order of importance based on the given heuristic. The term $rank(v)$ for node $v \in V$ denotes a node's rank in this hierarchy. In order to contract a node u, the node u is eliminated from the network, and shortcut edges are added to its neighboring nodes to ensure that the shortest paths that pass through u are still preserved. Once all nodes have been contracted, the found shortcut edges are then added to the original graph, and the newly constructed graph $G^+ = (V, E^+)$ with $E^+ = E \cup \{e : e \textit{ is a shortcut edge}\}$ is called the contraction hierarchy.

The contraction hierarchy is utilized in the query phase to find the shortest path with a bidirectional Dijkstra search. In the CH G^+, an edge $e = (v, w)$ is upward when $rank(v) < rank(w)$ and downward otherwise. The algorithm queries for the shortest paths that are up-down paths – those that consist of a sequence of upward edges followed by a sequence of downward edges. For each source s and target t given, there is an up-down path, which is also the shortest path, thanks to the construct of the CH. The bidirectional Dijkstra runs two specialized searches: a forward search from source s specifically for upward edges and a backward search from target t only for downward edges [12]. Because of the shortcut edges added, the two searches will eventually meet on node u, which is the node with the highest rank in the contraction hierarchy on the shortest path. Thus, the bidirectional query will find the shortest path. In order to easily differentiate between searches for explicitly upward or downward edges, the two subgraphs $G^{\uparrow} = (V, E^{\uparrow})$ and $G^{\downarrow} = (V, E^{\downarrow})$ are defined, with their edge sets $E^{\uparrow}$ and $E^{\downarrow}$ being subsets of E^+ containing only upward and only downward edges, respectively.

Bucket-based Contraction Hierarchies. CH with buckets is introduced by Knopp et al. as an extension to speedup algorithms such as CH or customizable CH [22]. Bucket-based CH (BCH) is an effective CH-based approach for the one-to-many shortest path problem, which combines the information during the forward and backward search effectively. Given a graph $G = (V, E)$, the problem is to find all shortest paths from one source node s to a set of target nodes T. Each node v in the input graph is associated with a bucket $B(v)$, which is empty at first. This bucket will be continuously filled during backward searches on subgraph $G^{\downarrow}$ from targets in T. A pair of $(t, dist(v, t))$ is stored in $B(v)$, when a path from v to t is found during the backward search from target t [22]. During the forward search afterwards, a tentative distance array D stores the distances between each pair (s, t). When the forward search settles a node v, it scans the node's bucket, and for each entry $e = (t, dist(v, t)) \in B(v)$ the tentative distance $D_s(t)$ is updated to $min\{D_s(t), dist(s, v) + dist(v, t)\}$. This bucket-based approach can be utilized analogously in the case of many-to-one shortest path problem, in which the forward and backward search exchange roles.

4.1.3 Bundling Searches

Bundled Searches. Dijkstra-based algorithms can search for the shortest path from multiple starting points simultaneously by bundling searches from K sources. In this method, each vertex keeps track of K tentative distance labels. When an edge is relaxed, the algorithm tries to update all K distance labels at the destination vertex. Bundled relaxation can be more memory-efficient because all K distances are stored consecutively in memory. However, it might do unnecessary work if not all K searches have reached the same vertex. Hence, bundling works best when all K searches mostly use the same edges. The value of K can be adjusted to tune the performance. These bundled searches were first introduced for Dijkstra searches as centralized searches [15].

SIMD Parallelism. To accelerate bundled searches, we can leverage single-instruction multiple-data (SIMD). This approach takes advantage of modern CPUs' capabilities, which include special vector registers and instructions designed to handle multiple data items at once. By vectorizing the necessary computations during edge relaxations, we can perform K computations simultaneously using a single vector instruction. This utilization of SIMD instructions has been shown to significantly enhance the speed of bundled searches [7].

4.2 Dynamic Taxi Sharing Algorithms

In this section, we will summarize and explain the details of the two dynamic taxi sharing algorithms that are used as the basis of our contributions. Our work is developed based on these algorithms, therefore the main definitions and the overall implementation structure remain unchanged.

4.2.1 LOUD

As previously mentioned, LOUD - for local bucket dispatching - is a dispatching algorithm for finding best assignments of requests to vehicles based on bucket-based contraction hierarchies [6]. Here, the best insertion of the incoming request into a vehicle's route must minimize a cost function.

BCH Queries. In order to calculate the distances needed for the cost function, LOUD utilizes BCH queries. For each vertex h in the graph, a source bucket $B_s(h)$ and a target bucket $B_t(h)$ are maintained. These two buckets are both initially empty but are filled gradually whenever a stop s_i is inserted into a vehicle's route. Then, a forward (reverse) BCH search is run from s_i, and for each vertex h settled, an entry $(s_i, d_s(h))$ $((s_i, d_t(h)))$ is added to the corresponding bucket $B_s(h)$ $(B_t(h))$.

Elliptic Pruning. The most essential contribution of LOUD is a technique for pruning these BCH searches called elliptic pruning. Elliptic pruning utilizes the fact that the leeway $\lambda(s_i, s_{i+1})$ for a detour between two consecutive stops s_i, s_{i+1} is bounded due to the hard constraints of already matched requests. For each additional stop v that may be inserted between s_i and s_{i+1} the stop must satisfy the following constraint: $dist(s_i, v) + dist(v, s_{i+1}) \leq \lambda(s_i, s_{i+1})$. The added stop must lie inside a shortest-path ellipse with the two given stops s_i and s_{i+1} as its foci. Consequently, while computing entries for target or source bucket from s_i, LOUD inserts an entry at vertex h if and only if h lies within the ellipse around s_i and s_{i+1}.

Elliptic pruning accelerates BCH searches as it accounts for smaller buckets to be scanned. Crucially, LOUD stores the corresponding vehicle identifier in every bucket entry. The algorithm can thus rule out the possibility of a feasible insertion for vehicles for which a bucket entry has never been scanned. Only a small set of candidate vehicles C remains, which reduces the number of insertions that need to be enumerated and enhances the dispatching performance.

Request Handling. LOUD handles an incoming request in four different phases subsequently: computing shortest-path distances, trying ordinary insertions, trying special case insertions, and finally updating the vehicle fleet. For an incoming request $r = (orig, dest, t_{req})$ and a vehicle ν with route $R(\nu) = \langle s_0, ..., s_k \rangle$, we consider the insertion $i = (r, p, d, \nu, i, j)$. In this case, the pickup location p coincides with the passenger's origin *orig* and is inserted right after stop s_i. Similarly, the dropoff location d coincides with the passenger's destination *dest* and is inserted right after stop s_j.

In the first phase, shortest path distances are computed with the help of CH and BCH queries. Initially, the shortest path from pickup p to dropoff d is calculated with a standard CH query. Afterwards, four BCH searches are performed, which include two forward and two reverse BCH searches from pickup p and dropoff d. These searches find the following distances needed for the computation of the cost function:

$$dist(s_i, p), dist(p, s_{i+1}), dist(d, s_j), dist(s_{j+1}, d) \tag{4.1}$$

These distances can only be calculated by BCH searches with elliptic pruning, if $0 < i \leq j < k$, then the corresponding insertions are called ordinary insertions, which will be tried in the upcoming phase. The remaining special cases where $i = 0, i = k$ or $j = k$ will be considered separately by the algorithm in a later phase.

Ordinary Insertions. LOUD tries the different feasible ordinary insertions in the second phase, in which it checks all required constraints, including time constraints and capacity constraints. LOUD applies similar constraints to our constraints, mentioned in Section 2.2. Here, the algorithm takes into consideration only the set C of candidate vehicles.

Special Case Insertions. The special cases of LOUD consist of pickup before next stop (PBNS), pickup after last stop (PALS) and dropoff after last stop (DALS) insertions. They are considered by LOUD in this same order. In these cases, the BCH queries with elliptic pruning cannot be applied to find the distances regarding the next stop and the last stop. Therefore, separate queries are applied to compute the required shortest path distances.

In the case of $i = 0$ or PBNS, the pickup is inserted right before the next stop in the vehicle's route. The vehicle v has to make a detour from its current location between stops s_0 and s_1 directly to the pickup location p. Consequently, the distance between the vehicle's current location and the pickup $dist(l_c(v), p)$ is needed. However, since the vehicle is constantly moving and changing its location, the BCH queries cannot calculate this distance. Instead, it can only give a lower bound on the pickup detour. Because the vehicle has to divert to the pickup spot while traveling from s_0 to s_1, the pickup detour is defined as:

$$\delta_p = dist(s_0, l_c(v)) + dist(l_c(v), p(r)) + dist(p(r), s_1)$$

Based on the triangle inequality, it is known that:

$$dist(s_0, p(r)) \leq dist(s_0, l_c(v)) + dist(l_c(v), p(r))$$

Therefore, we can obtain a lower bound on the pickup detour δ_p:

$$dist(s_0, p(r)) + dist(p(r), s_1) \leq \delta_p$$

Utilizing this lower bound on the detour, a lower bound on the insertion cost may be calculated, and only when this lower bound cost is better than the current best insertion cost is an exact shortest path distance between the current vehicle location $l_c(v)$ and the pickup position $p(r)$ required. LOUD computes this exact distance with a standard CH query.

The remaining two cases involve either $i = k$ in case of PALS or $j = k$ in case of DALS. Whereas in PALS insertions, both pickup and dropoff stops are inserted after the vehicle's last stop, in DALS insertions, the pickup is inserted before and the dropoff after the last stop. In these two cases, the distance between the last stop s_k and a pickup p or a dropoff d is required. As LOUD does not generate source bucket entries for the last stop, elliptic pruning is inapplicable here, where the leeway λ is unbounded. These two special cases are considered after all ordinary and PBNS insertions have been tried. Here, a reverse Dijkstra search from the pickup p or the dropoff d is run, and whenever the last stop s_k is settled, the algorithm checks whether the resulting PALS or DALS insertion is better than the current best insertion. This Dijkstra's search can be stopped early, as soon as the radius of the search no longer admits an insertion with a smaller cost than the best known insertion [6].

4.2.2 KaRRi

KaRRi (for Karlsruhe Rapid Ridesharing) is an extension of LOUD, with the introduction of meeting points. As previously described in the problem statement (see Chapter 2), this algorithm utilizes sets of potential pickup and dropoff locations instead of the origin and destination of the passenger as in the original dynamic taxi sharing problem setting. This has led to some conceptual changes that contribute to the cost function described in Equation (2.1).

Cost Function. Originally, if the rider is being picked up at their given location, the vehicle never has to wait and can depart as soon as it arrives at the origin, meaning $t^{min}_{arr}(s_i) = t^{min}_{dep}(s_i)$. However, now that both the rider and the vehicle have to transport to the pickup spot, it is possible that the vehicle arrives earlier and thus must wait for the rider. Formally, this happens when $t^{min}_{dep}(s_i) + dist_{veh}(s_i, p) < t_{req} + dist_{psg}(orig, p)$. The departure time at stop s_i of the insertion $t_{dep}(\iota)$ is then the time when both parties have arrived at the pickup. The wait time of the vehicle at stop s_i can be written as $w(s_i) = t^{min}_{arr}(s_i) - t^{min}_{dep}(s_i)$. This wait time can affect the vehicle's operation time and the trip times of other riders.

The detours regarding the pickup and the dropoff that a vehicle has to make due to a new insertion ι are denoted as:

$$\begin{aligned} t_{detour}(\iota) &:= \delta_p(\iota) + \delta_d(\iota) \\ \delta_p(\iota) &:= t_{dep}(\iota) - t^{min}_{dep}(s_i) + dist(p, s_{i+1}), -dist(s_i, s_{i+1}) \\ \delta_d(\iota) &:= dist(s_j, d) + dist(d, s_{j+1}) - dist(s_j, s_{j+1}) \end{aligned}$$

In KaRRi, there is a set of possible pickups and a set of possible dropoffs for each request instead of exactly one PD-pair as in LOUD. Thus, the number of potential insertions is much larger, which in turn demands the computation of many more shortest path distances. Therefore, in addition to the existing speedup techniques in LOUD, the authors of KaRRi introduced some many-to-many routing techniques in order to approach this problem.

Request Handling. Now that the most essential difference between KaRRi and LOUD is clarified, we will continue to explain the process of handling an incoming request in KaRRi. For each request, the possible meeting points for this particular request will first be found. This is done by running bounded Dijkstra searches within walking distance of ρ from the origin *orig* and destination *dest* of the request r. Afterwards, the algorithm evaluates different types of insertions: ordinary, ordinary paired, PBNS, PALS, and DALS. For each phase of an insertion type, the best insertion so far ι^* with the minimum cost is updated to be used in the next phases.

Elliptic BCH Searches. The distances between a vehicle's existing stops and the feasible PD-locations are required. Here, BCH queries are employed to find the distances with elliptic pruning similar to LOUD. In comparison to the distances required in LOUD for exactly one PD-pair as mentioned in Equation (4.1), these distances must be calculated for the two sets of possible pickups $P_\rho(r)$ and dropoffs $D_\rho(r)$.

BCH searches with elliptic pruning are called elliptic BCH searches in KaRRi, to which two new routing techniques are introduced [23]. The first technique is based on the fact that an insertion between two consecutive stops s_i and s_{i+1} can only hold all hard constraints of the existing assignments when the detour does not exceed a leeway of $\lambda(s_i, s_{i+1})$. As a result, while scanning the source bucket entries of a node $B_s(v)$, the entry $(s, \mathit{dist}(s_i, v))$ is only relevant if $\mathit{dist}(s_i, v) + \mathit{dist}(v, p) \leq \lambda(s_i, s_{i+1})$. Based on this inequation, the remaining leeway of an entry is defined as $\lambda_{res}(s_i, s_{i+1}) := \lambda(s_i, s_{i+1}) - \mathit{dist}(s_i, v)$. In an effort to reduce the number of bucket entry scans performed by each BCH search, KaRRi sorts the bucket entries based on their remaining leeway in descending order. In this case, a BCH search can stop scanning the entries of a bucket as soon as an entry $(s, \mathit{dist}(s, v))$ is found with $\mathit{dist}(v, p) > \lambda_{res}(s_i, s_{i+1})$. Secondly, a further technique is applied, which aims to exploit the locality of the pickups and dropoffs: bundling BCH searches. Running a bundled search means running K searches rooted at K pickups (dropoffs) simultaneously and updating K distances with each edge relaxation (see Section 4.1.3). Since the pickups (dropoffs) lie within the walking radius ρ of the rider's origin (destination), it is highly likely that the pickups (dropoffs) share similar search spaces. Thus, many edges have to be relaxed for all K searches, making bundled searches highly effective for this use case.

PD-Distances. For the case of paired insertions, where the pickup and dropoff points are both inserted between the same pair of existing stops, the shortest path distance between the pickup and the dropoff has to be known. Here, BCH searches are used instead of CH searches as in LOUD, along with further techniques to speed up the searches. KaRRi defines an upper bound on all PD-distances as follows:

$$\mathit{dist}_{PD}^{max} := \max_{p \in P_\rho} \mathit{dist}_{psg}(\mathit{orig}, p) + \mathit{dist}(\mathit{orig}, \mathit{dest}) + \max_{d \in D_\rho} \mathit{dist}_{psg}(d, \mathit{dest})$$

where the two maximum distances can be calculated by running two local Dijkstra searches at *orig* and *dest* and the distance between the origin and the destination can be calculated with a single direct CH query. This upper bound can be applied during BCH search scans to only generate and scan entries whose distance is less than or equal to dist_{PD}^{max}. Moreover, the PD searches can also be bundled as the set of sources and targets here is localized.

PBNS Insertions. In the case of a PBNS insertion, KaRRi makes use of the same lower bound cost used by LOUD (see Section 4.2.1), so that not all distances between the vehicle's current location and the pickup locations have to be calculated. KaRRi uses a BCH searche to compute the necessary exact distances from a vehicle's current location to pickups, which can also be bundled.

PALS and DALS Insertions. Similarly to LOUD, the two remaining cases, namely PALS and DALS, are considered separately. These two special cases take up the majority of the total runtime of LOUD, due to the slow Dijkstra searches applied here. Therefore, in order to speed up these searches for the sets of pickups and dropoffs, KaRRi utilizes BCH searches instead.

Even though elliptic pruning is not applicable, BCH searches can still be used to find the distance from the last stop to the pickup (dropoff). This implies that the algorithm must maintain last-stop buckets with the following entries: $\forall s_{k(v)} : (s_{k(v)}, \mathit{dist}(s_{k(v)}, v)) \in B(v)$ for each $v \in V$. Then, for each pickup $p \in P_\rho$ (dropoff $d \in D_\rho$), KaRRi runs an individual

BCH search to explore the reverse search space rooted at p (d). This would update the tentative distance to $dist(s_{k(\nu)}, v) + dist(v, p)$ ($dist(s_{k(\nu)}, v) + dist(v, d)$) and eventually find the shortest path distance between the last stop of vehicle ν and the PD-location $dist(s_{k(\nu)}, p)$ ($dist(s_{k(\nu)}, d)$). These individual BCH searches can be bundled in a similar manner as the elliptic BCH searches. In an attempt to reduce the number of bucket entry scans, KaRRi also uses sorted buckets and updates the upper bound cost to help stop the search earlier.

One of the major contributions of KaRRi is the collective last-stop BCH searches for PALS and DALS. This originates from the idea that we do not need to know the distance between every last stop and every pickup or dropoff. If the best PALS or DALS insertion is known beforehand, it is this one distance $dist(s_{k(\nu)}, p)$ or $dist(s_{k(\nu)}, d)$ that requires exact calculation. Knowing the best insertion beforehand is impossible. Nonetheless, the authors of KaRRi came up with a subroutine to prune the individual PD-pairs by comparing them to each other whenever they meet at a vertex [23]. The PD-pairs are pruned by comparing the progress of the searches. This involves labels that identify the PD-pair at a specific vertex $v \in V$. The algorithm settles the labels one-by-one and applies a pruning technique called *domination pruning*, before propagating the label to the neighboring vertices. This pruning technique compares labels of the same vertex and discards the label l' if it is dominated by another label l. Details of collective BCH searches involving labels and domination pruning are not further discussed, as our algorithm did not attempt to speed up this algorithm due to its already optimized runtime.

5 Parallel Algorithm

Even though LOUD and KaRRi have both significantly improved the runtime of the dynamic taxi sharing problem in comparison to previous algorithms, it is suggested by the authors to apply parallelism to further enhance the performance. We present our approach to parallelizing the different phases of KaRRi in an attempt to improve the algorithm's running time. The critical problem in KaRRi remains that the searches have to be run for a number of PD-locations instead of one single PD-pair as in LOUD. Therefore, we explore the idea of parallelizing the different searches over the PD-locations on multiple threads.

In the following sections, we will go through all the phases of KaRRi in the order in which we parallelized them and explain each phase individually. We present the parallel algorithm along with the problems and difficulties that we encountered. Here, we only describe the high-level algorithm, the implementation details are discussed in Chapter 6.

5.1 Elliptic BCH Searches

Initially, we started with the elliptic BCH searches, as these searches make up the largest part of the runtime in KaRRi. Elliptic BCH searches are well suited for parallelization, since the searches are entirely independent from one another for each PD-location. Moreover, the number of elliptic BCH searches scales linearly with the number of relevant PD-locations. Therefore, it is logical to implement these searches to run in parallel over the PD-locations.

For each PD-location, two elliptic BCH searches have to be run: a *to-search* which finds the distances from existing vehicle stops to the PD-location, and a *from-search* which finds the distances from the PD-location to existing vehicle stops. In KaRRi, the algorithm iterates through the PD-locations (or batches of PD-locations in the case of bundled searches) sequentially and performs to- and from-searches to find the required distances. The two searches are each run once for the pickups and once for the dropoffs.

Our parallel algorithm loops through the PD-locations in parallel, assigning one PD-location to each worker thread. Each thread then performs the BCH query for the assigned PD-location. If there are more PD-locations than available worker threads, a thread is repeatedly assigned a new PD-location upon finishing a query. So that the to- and from-search can be run by multiple threads simultaneously, the data structures associated with the searches are saved in the thread local storage. This means that each thread will obtain a thread-specific copy of the required data structures and can use this to individually run the searches within its local space. Details on thread local storage will be discussed in Section 6.1.

The threads compute the distances for different PD-locations in parallel. Each thread temporarily stores their found results in a thread local distance vector. In total, this would require $O(|V| \cdot T)$ with T as the number of threads. This local distance data structure is reused for each query that the thread processes. At the end of a query, the local results for the current PD-location are transferred to a shared global result. As the result distances are accessed by later phases for enumeration, it is important that the global results have a certain structure so that the enumeration process is fast. The problem of allocating entries for global results also occurs in other searches. Therefore, we consider this in more detail in Section 6.2, specifically for the case of elliptic BCH searches, as we tried out several approaches for this search type first before being able to apply them to other search types.

In order to prune the number of ordinary insertions that need to be enumerated after the searches, the elliptic BCH searches also compute some auxiliary information. The algorithm maintains a subset C of candidate vehicles, for which the elliptic searches found some relevant distance. Furthermore, for every existing vehicle stop s, the minimum distance from s to any PD-location and the minimum distance from any PD-location to s are computed and stored. The algorithm uses these minimum distances to calculate the minimum cost of all insertions, which can be used to check whether a specific insertion has to be enumerated. Both the vehicle subset as well as the minimum distances are made thread-safe in our algorithm, whose implementation will be discussed in Section 6.4 and Section 6.3, respectively.

The sequential approach first performs all from-searches and then all to-searches. In the parallel setting, this would incur a global synchronization point between the two parallelized loops. To avoid this synchronization point, we interleave the searches by having each thread perform both the from- and to-searches for its assigned PD-location. This is possible as the to- and from-searches are parallelized similarly over the PD-locations.

5.2 Pickup and Dropoff After Last Stop

The next insertion types that we considered are PALS and DALS. These are previously mentioned as the two special cases in the original LOUD algorithm that take up the majority of the runtime. Even though these searches are then significantly improved in KaRRi thanks to the introduction of collective BCH searches, we focus on speeding up the individual last stop BCH searches. Elliptic pruning cannot be applied here, as the leeway λ is unbounded. Nonetheless, it has been shown in KaRRi that last stop BCH searches may still be put into use [23]. We aim to run these BCH searches in parallel, analogously to the elliptic BCH searches in the previous section. Furthermore, we parallelize the process of enumerating the found assignments.

At first, we implemented parallel searches and parallel enumeration of assignments separately. This induces a global synchronization point between the searches and the enumerations. Since these two processes are both parallelized over the PD-locations, they may also be interleaved instead to remove this synchronization point. A thread then first runs the searches for a single PD-location and enumerates the assignments regarding this PD-location right after. In the upcoming subsections, we will describe the last stop BCH searches and the enumeration process more specifically.

5.2.1 Last Stop BCH Searches

The last stop BCH searches find the distance between a vehicle's last stop and the different PD-locations. As each vehicle v has a distinct last stop $s_{k(v)}$, these distances can be identified by the vehicle and the PD-location. The searches are parallelized over the pickups in the case of PALS, and over the dropoffs in the case of DALS.

Each thread searches for the distances from every vehicle's last stop to its dedicated PD-location. Here, the thread runs an upward reverse search from its PD-location to find the wanted distances from the vehicles' last stops. The data structures used by the search are stored in each thread's local storage to be utilized during the parallel runs. The distances found are then written into a global shared result, similarly to elliptic BCH searches in Section 5.1, after a thread has finished its searches for the current PD-location. We applied dynamic allocation for the search results here, which will be discussed in more detail in Section 6.2.3.

The last stop BCH searches also compute a shared subset C of vehicles for whom any relevant distance was found. This set is utilized to speed up the process of enumerating insertions. Analogously to the elliptic BCH searches (see Section 5.1), we construct this subset in a thread-safe manner.

5.2.2 Enumerating Assignments

After running the last stop BCH search for its dedicated PD-location, each thread then enumerates the different assignments associated with this PD-location. For each candidate vehicle $v \in C$ and for each relevant PD-pair, an assignment is enumerated. We must update the request globally if we find a new best cost. As multiple threads enumerate the assignments simultaneously, access and update to the global best known assignment and best known cost must be ensured thread-safe. The implementation details for this will be addressed in Section 6.5.

5.3 Pickup Before Next Stop

In the case of a PBNS assignment, the vehicle, which is currently located between stops s_0 and s_1, would redirect to the pickup location directly to pick up the new passenger before transporting to its next stop. Therefore, we need to calculate the distance from the current vehicle's location to the pickup location. We cannot pre-compute this distance because the vehicle is constantly moving on its route. Based on a lower bound cost as defined in LOUD (see Section 4.2.1), only certain combinations of vehicles and pickup locations have to be calculated. In the sequential algorithm, KaRRi iterates through the vehicles to find pickup locations that require the exact distance and adds them to a queue. KaRRi delays the actual calculations until it finds all feasible pickups for a given vehicle. The distances from a single vehicle to the relevant pickups will be calculated in a bundled fashion.

In a special case, a pickup may be inserted before the next stop of a vehicle, while the dropoff is inserted after its last stop. We call these the DALS + PBNS assignments, which will be considered separately from other PBNS assignments in Section 5.3.2.

In our parallel algorithm, we modified searches to find the distance from the current vehicle's location to the pickup to be able to run in parallel. There are two variations implemented, one search type that can calculate the distance between a vehicle's location and a pickup on-the-fly and one search type that calculates the distances in batches. These variations of the search are applied in different phases of the algorithm: the DALS + PBNS assignments require the individual thread to determine this distance on-the-fly, whereas for ordinary and paired PBNS assignments, these calculations can be delayed similarly to the sequential algorithm. The batched searches aim to improve performance by calculating distances between multiple vehicles and multiple pickup spots in a bundled fashion.

5.3.1 Finding Distances Between Current Vehicle's Location and Pickup

In the sequential algorithm, the distances from a single vehicle's location to all required pickups are computed at once. First, bucket entries are constructed in a forward CH search rooted at the vehicle's location. We then run bundled BCH queries from all pickups using these buckets. We have implemented two parallel variants of these searches, as detailed in the following paragraphs.

On-The-Fly Calculation. This variant enables a thread to calculate the required distance on-the-fly during enumeration. The distance from the vehicle's current location to a specific pickup location is calculated using a single point-to-point CH query, as a BCH query only makes sense when finding one-to-many shortest path distances. This distance is computed individually and immediately when required.

Batched Calculations. This variant calculates the distances in batches of vehicles and pickup locations. Therefore, we can apply BCH searches to find these distances more effectively. The algorithm calculates the distances from relevant vehicles to relevant pickups for PBNS assignments in two steps. Initially, we run forward searches to construct the buckets for designated vehicles sequentially. These searches are not parallelized due to the small number of searches in comparison to the parallel overhead. Then, we run reverse searches from the pickup locations in parallel over the relevant pickups to find the required distances.

A common issue encountered during these calculations is the location of the vehicle at the time. As the current position of the vehicle may be unknown, it must be located before running the searches to find the related distances. We keep track of a set of vehicles with known locations to be able to check whether or not we have to compute the current location of a vehicle. Since this is executed in parallel, multiple threads may want to locate the same vehicle at once. Thus, we must ensure that a vehicle's location is only computed once by a single thread and stored in a shared vector. Afterwards, other threads can access the found location without having to locate the vehicle again. The algorithm utilizes a direct CH query to determine the path that the vehicle is taking. We then reconstruct this path and compare the departure time at each edge to the current request's time to locate the edge that the vehicle is currently on.

Both variants of the search write their found distances into a shared data structure. This allows the two search types to access each other's found distances and avoid calculating the same distance twice.

5.3.2 Enumerating Assignments

DALS + PBNS. We calculate the required distances individually in this special case, without batching. The parallel enumeration of DALS insertions handles this case, filters, and calculates the distances from the current locations of the vehicles to the pickups. Here, the individual thread computes the required distance between the current vehicle's location and a pickup on-the-fly and enumerates the corresponding assignment directly after (see Section 5.3.1).

Ordinary and Paired PBNS. In other PBNS assignments, which include ordinary and paired assignments, we enumerate the assignments similarly to the sequential algorithm, delaying the calculation of the required distances. To begin with, we determine the relevant vehicles and pickups for the searches. The algorithm sequentially loops through all vehicles to find pairs of vehicles and pickup locations. Each pair then represents a task for our parallelization. The individual thread then determines whether a specific pair of vehicle and pickup requires the exact distance between them. This is done by using a lower bound cost for each assignment and checking whether this is better than the current global best cost, similarly to LOUD (see Section 4.2.1). If the distance is required, the vehicle and pickup are both marked as relevant. The calculation of the exact distances is delayed until all the required distances have been determined. Checking the corresponding assignments with exact distances is likewise deferred.

In this case, the searches to find the distances from relevant vehicles' last stops to relevant pickup locations are run in batches. After all distances have been calculated in a bundled fashion, the algorithm enumerates the deferred assignments in parallel. During the enumeration, the thread has to update the best cost and best assignment, similarly to the PALS and DALS enumerations (see Section 5.2.2).

Overall, we apply parallelization during the determination of relevant vehicles and pickups, the batched searches, and the enumeration of assignments. However, as these processes are each parallelized over different constructs, they cannot be interleaved and hence incur global synchronization points in between. We accept these synchronization points in order to make use of parallel batched BCH queries instead of many parallel individual CH queries.

5.4 Pickup-Dropoff Searches

The remaining searches to be parallelized are the PD-distance searches. PD-distances are distances between the relevant pickups and dropoffs. It is a many-to-many shortest path problem to calculate the distances from every relevant pickup to every relevant dropoff. These distances are required for paired insertion types, in which the pickup and the dropoff are both inserted between the same pair of existing stops. In KaRRi, we can specify whether to run these searches using BCH or CH searches, with the BCH strategy being the default option.

5.4.1 BCH Strategy

Pickup-dropoff BCH searches involve generating the bucket entries for every dropoff in their reverse CH search space and then running the queries in the forward CH search graph, starting from each pickup.

Fill Dropoff Bucket Searches We run parallel searches from every dropoff to construct the required bucket entries s.t. the BCH queries rooted at pickups can later compute the PD-distances. The significant difference from other searches is that the results found here are not distances but bucket entries of vertices. Since the number of vertices is large, the bucket entries belonging to a vertex must be stored in an effective way. Here, dynamic allocation is applied for allocating the result entries, but adapted to fit the use case (see Section 6.2.3).

Pickup Searches Subsequently, we run pickup searches in parallel over the pickups to find the necessary PD-distances. We need to find distances from every pickup to every dropoff. Therefore, we statically allocate global result entries to the required size before the searches, mirroring the original sequential algorithm. Static allocation of query results will be further discussed in Section 6.2.1.

5.4.2 CH Strategy

For the CH strategy, the algorithm runs a single direct CH query to find the distance between a pickup and a dropoff. Here, queries are run from all pickups to all dropoffs. We parallelized the CH searches over the combinations of pickups and dropoffs. Within an iteration, a thread works on a single pickup and dropoff pair. The thread finds the distance between its assigned PD-pair and updates this to the global shared result.

For both strategies, a global minimum PD-distance and the minimum PD-distance per pickup are kept track of throughout the searches. Since multiple threads may access these minimum values concurrently, we ensure thread-safe access by applying atomic operations, described in more detail in Section 6.3.

6 Implementation Details

In this chapter, we will discuss the fundamental components of our algorithm in more detail to provide a better understanding of the actual implementation. The following sections represent the most important constructs, which are applied in various phases of the parallel algorithm.

Our algorithm is written based on the source code of KaRRi and LOUD in C++ 17 [6, 23]. We chose Intel oneAPI Thread Building Block (`TBB`) as our main toolset to implement parallelism due to the ease of use and variety of templates given. `TBB` is a flexible library that simplifies the work of adding parallelism and maps the logical implementation to be put into execution on threads. We employed `TBB`'s `concurrent_vector` for data structures that require concurrent growth and `TBB`'s `enumerable_thread_specific` for the use of thread local storage. For synchronization of primitive types, the C++ standard template library `atomic` was utilized.

6.1 Thread Local Storage

A general similarity between the different search types while applying parallelism is that each thread has to individually run the searches for their dedicated PD-location. Therefore, the data structures used by searches must be stored in the thread local storage to be accessed independently by the different threads. These essentially include the distance arrays that the Dijkstra-based searches use. In total, each thread would have to save $O(|V|)$ values in its local storage. We used the template `enumerable_thread_specific` from `TBB` to implement this. This template ensures that each thread obtains a unique copy of the wrapped object to use within its local execution without interfering with another thread's copy.

6.2 Allocating Query Results

We encountered some fundamental difficulties while parallelizing the searches over the PD-locations. As noted by the authors of KaRRi [23], the pickups and dropoffs are localized and thus have largely similar search spaces in all shortest path queries. In KaRRi, bundled BCH searches exploit the localization of PD-locations since the searches relax similar edges at the same time due to their similar search spaces. In the case of parallel searches, this means that several threads would want to make updates to data structures for the search

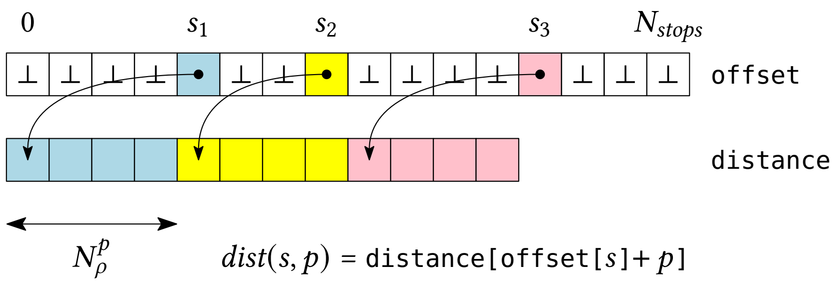

Figure 6.1: Example for dynamic allocation. s, s_1, s_2, s_3 are IDs of stops and p is the ID of a pickup. N_{stops} denotes the total number of stops currently scheduled for all vehicles. $\perp$ represents an invalid index value.

results simultaneously, which can lead to write conflicts and synchronization effort. In the following, we describe the details in the case of elliptic BCH searches. These concepts may be applied in other searches analogously. Recall that in this type of search, the algorithm searches for the distances between a vehicle's existing stops and PD-locations (see Section 5.1).

The sequential KaRRi algorithm allocates memory for the results of elliptic BCH searches dynamically. This means that memory for the distance between a stop s and a pickup p is only allocated once the respective elliptic BCH search finds any viable distance from s to p. This approach is not only memory-optimized but it is also cache-efficient, since only a few distances are updated repeatedly during the search while the majority of distances remain untouched.

To store the results of the elliptic BCH searches, KaRRi uses two vectors as visualized in Figure 6.1: a distance vector, which stores the actual distances, and an offset vector, which stores for each stop where in the distance vector the distances for that stop can be found. To start with, the offset of each stop is initialized to the invalid value $\perp$ and the distance vector is empty. The first time an elliptic BCH search finds a distance between a stop s and any pickup, KaRRi allocates entries for the stop. This is done by appending N_p^p many entries at the end of the distance vector and setting the offset for s to the beginning of this range. The index for the distance between stop s and a pickup p is then calculated from the offset of s and the ID of p. We described this for the case of pickups, in the case of dropoffs, it works analogously.

KaRRi performs batched allocations, allocating the memory for the distances between a stop s and every pickup once any search finds a viable distance between s and any single pickup. This batched approach makes sense since the pickups share similar search spaces, and thus it is highly likely that when a distance between a stop and a pickup is found, other distances between this stop and other pickups will also be found. Furthermore, it is beneficial to have the distances for a stop consecutively stored in the memory, because then the distances can be accessed more efficiently later when enumerating the insertions.

When executed in parallel, writing the results themselves is not a problem because the threads always work at different PD-locations and so never calculate the exact same

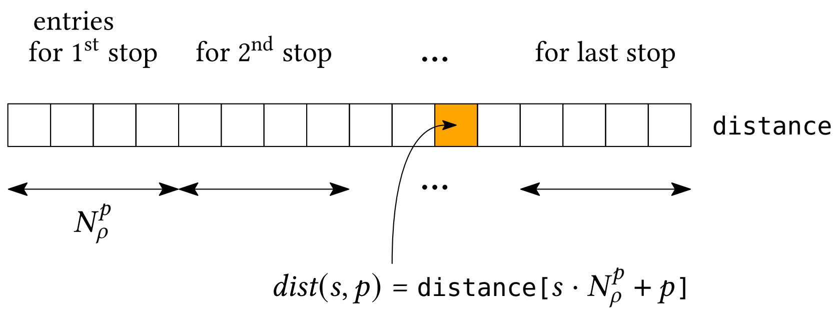

Figure 6.2: Example for static allocation. s is the ID of a stop and p is the ID of a pickup.

distance. However, it is the dynamic allocation of memory for the results that can lead to conflicts when executed in parallel. Multiple threads may simultaneously want to allocate entries for the same stop. This is a critical problem that lies at the center of our algorithm and has to be taken into consideration for all the different search types mentioned in Chapter 5. We tested out three alternatives to allocating query results in the following subsections: static allocation, semi-static allocation with the use of preliminary searches, and thread-safe dynamic allocation.

6.2.1 Static Allocation

First, we attempted a simple alternative, which is to statically allocate the required entries for all stops in advance. This way, while running the individual searches, each thread can assume that the shared distance vector has pre-allocated entries for the current stop and PD-location. Furthermore, since the threads each work on distances regarding a dedicated PD-location, it can be guaranteed that they never have to access the same entry in the distance vectors concurrently.

Since the entries in the distance vectors are statically allocated from the initialization of the searches, an offset vector for the indirection is no longer required. Instead, using a result entry's stop and PD-location, we can arithmetically calculate its position in the distance vector. As visualized in Figure 6.2, for distances between pickups and existing stops, the distance vector is initially allocated to the size of $N_\rho^p \cdot N_{stops}$, with N_ρ^p as the total number of relevant pickups and N_{stops} as the total number of stops. Then, the distance between the stop s and the pickup p can be found in the statically allocated vector at index $s \cdot N_\rho^p + p$.

Although this approach is straight-forward and easy to implement, there are numerous downsides to it. As the vectors are statically allocated to a certain length during the initialization, it is highly likely that many entries remain unset as distances between certain stops and PD-locations are not found. The static allocation of result vectors leads to a large initialization time as well as a large memory footprint and bad cache behavior since only a small number of entries at arbitrary offsets in the large distance vector are used.

6.2.2 Semi-Static Allocation

This alternative aims to reduce the number of unused statically allocated entries in the global shared results. Here, we initially search for stops that are considered relevant and pre-allocate the entries for those stops only before running the actual parallel searches. The preliminary searches filter out stops that do not have feasible PD-locations associated with them. For each stop s, we compute the smallest distance from s to any PD-location and the smallest distance from any PD-location to s. We then find out whether the stop s has any relevant PD-locations at all, by checking if the sum of the two minimum distances is less than or equal to the leeway $\lambda(s, s')$ between stop s and the following stop s' in the route. For stops that may have feasible PD-locations, we pre-allocate the required entries in the global distance vector before the parallel searches.

For the preliminary search, we run a BCH search rooted at all pickups (dropoffs). If a vertex v lies in the search space of multiple sources, we settle it only once, using the smallest distance among all sources. Thus, scanning a bucket entry also always results in the smallest possible distance to any of the sources, and we obtain the required minimum distances at the end of the search.

We sequentially run a single preliminary search that finds the required minimum distances over all pickups and a second preliminary search that does the same for the dropoffs. These searches are run during the initialization phase prior to the parallel elliptic BCH searches. The pre-allocation prevents us from having to allocate any memory during the actual elliptic BCH searches. Unlike static allocation, this alternative only pre-allocates entries for stops that preliminary searches prove to be relevant, rather than allocating entries for all stops. Hence, less memory is wasted on unused distances, resulting in improved cache behavior. The two additional preliminary searches contribute to the extended initialization time required for this type of allocation.

6.2.3 Dynamic Allocation

The most effective and closest approach to the original algorithm is to apply thread-safe dynamic allocation. We want to implement the same construct as in the original sequential algorithm, which consists of a shared distance vector and an offset vector. In order to allow concurrent growth on the global distance vector, we utilized the container `concurrent_vector` from `TBB`. This vector is concurrently growable and accessible by multiple threads. The starting index of the newly allocated entry range is calculated and saved into the offset vector. Nonetheless, it must still be ensured that only one thread allocates entries for a specific stop at once and is not interrupted by another thread. As a consequence, we ensure the thread safety of the dynamic allocation by using a lock for each stop.

For elliptic BCH searches, the query results include distances and meeting vertices. The sequential algorithm uses four vectors in total: two vectors for the distances to and from relevant PD-locations and two vectors for the corresponding meeting vertices. If we employ this structure, we would need four concurrently growable vectors and four corresponding offset vectors. We cannot use a single offset vector for the four concurrent vectors since the range for a stop in each of the vectors could be different, as we grow the vectors and allocate entries concurrently. In order to encapsulate the results and avoid the use of multiple vectors for different values, we created a data structure to represent the result entry of the searches. In this data structure, all distances and meeting vertices related to a specific pair of stop and PD-location are stored. By utilizing this data structure, the found distances and meeting vertices can all be stored into a single `concurrent_vector`, and only one offset vector is required to indicate the position of the entry in the result entry vector. This also reduces the time a thread spends allocating while holding the lock, which reduces lock contention.

The process of a dynamic allocation occurs as follows: every time a thread wants to allocate new entries for a stop, it checks whether this stop already has the required entries. We mark in a thread-safe flag array whether the entries for a stop have been allocated, so that the thread does not have to acquire the lock every time just to find out that the required entries already exist. In case the stop does not have entries yet, the thread fetches the dedicated lock for this stop. We then double check once again whether the offset for this stop is already set to a valid value. If this is the case, the lock is immediately released as dynamic allocation is no longer required. If not, the entries are allocated while holding the lock to ensure that no other thread can attempt to also allocate entries for this stop at the same time. Once the entries for the value vector have been successfully added, the starting index of the allocated entry range is saved to the offset vector, and the lock is released.

Local and Global Entries. However, this implies that all threads would constantly try to allocate entries for different stops at the beginning of the parallel searches. This means that the threads need to concurrently grow the shared `concurrent_vector`. This container from `TBB` must also synchronize the calls for concurrent growth internally, which leads to contention. We came up with the idea of splitting the results into local and global entries so that the threads can temporarily save the found entries in their thread local storage during a search. Then, when a thread has finished the searches for its assigned PD-location, including a to- and a from-search, it transfers the local results into the global entry vector. For this purpose, the thread iterates through all stops that it finds relevant for its designated PD-location l. For each relevant stop s, the thread allocates global entries if necessary and writes the distances between s and l to the corresponding spot in the global entry vector. Consequently, the concurrent grow calls to the `concurrent_vector` are spaced out more evenly.

On top of that, there are other positive effects of the thread local results. An advantage lies in the fact that each thread does not have to do random access writes to arbitrarily distributed offsets in the global result vector but only works on its small local result vector. This increases the cache efficiency. Moreover, since we have the to- and the from-search for a PD location run by the same thread (see Section 5.1), the thread can work exactly for this PD location. Accordingly, it can say specifically for the dedicated PD location whether a stop is relevant in both directions or not. When transferring to the global result, only stops that are relevant in both to- and from-searches need to be taken into account.

Thread Local Data Structure. Each thread has a thread local data structure to store its found entries. In the case of elliptic BCH searches, an index vector and a local distance vector similar to the sequential algorithm are in use. We initialize the local index vector to the number of stops before each search. Once a distance is found, the thread allocates a single entry in the local distance vector and saves the index of this in the local index vector. The thread first writes all its newly discovered entries in the local data structure, which lies entirely in its own thread local storage. Thus, during the searches, the thread does not have to worry about write conflicts with other threads that are working in parallel.

For other types of searches, the local entry data structure can be adjusted to fit the specific use case accordingly. When running searches to fill dropoff buckets in the case of PD-distance searches, we find bucket entries of vertices instead of distances (see Section 5.4.1). Due to the large number of vertices, it is infeasible to store an index for each vertex inside the thread local storage. To avoid using an index vector entirely, we instead store the bucket entries as pairs of vertex and distance in a single thread-local vector. In order to find and update a bucket entry at a specific vertex, the thread then uses a linear search through all bucket entries generated so far in the local search. This method is still reasonable, as any given search will only generate bucket entries at a small number of vertices. Moreover, we settle a vertex at most K times in the case of bundled searches, meaning we have to run maximum K linear searches to update the bucket entry. Without bundled searches, we do not have to do linear scans at all because each vertex is settled exactly once, and we can simply append the new bucket entry to the end of the thread-local vector.

Randomization of Write Orders. As the search spaces for each PD-location are similar, threads may encounter the same relevant stops in the same order during their local searches. This may lead to multiple threads trying to allocate entries for the same stop at the same time, which in turn results in contention on the lock of this particular stop. To reduce this contention, each thread transfers the results for its relevant stops in a randomized order.

6.3 Accessing and Updating Minimum Distances

In certain searches, it is necessary to store the minimum distance found over all PD-locations. These minimum distances are usually utilized during a latter phase for the calculation of cost lower bounds that serve to prune the solution space of potential best insertions. As every thread has to update these minimum distances during their searches, access to such values must remain thread-safe.

We utilized `atomic_int` from the C++ standard library to store the minimum distances. This allows us to perform thread-safe updates on the minimum values using a *compare-and-swap* loop. Assume a thread t found a new distance d that may be a candidate for a global minimum distance stored in the atomic A. Then, t atomically reads the current value d' at A and compares the value with d. If $d < d'$, a compare-and-swap operation on A allows t to atomically check if the value at A is still equal to d' and if so, replace the value at A with d. If a different thread has changed the value at A to some other value $d'' \neq d'$ in the meantime, t is informed about this instead. Then t compares d and d'' and potentially tries to update the value at A again by using another compare-and-swap operation. The thread iterates this approach until it either successfully stores d at A or it reads a value $d^* < d$ at A.

6.4 Thread-Safe Subset

Our algorithm keeps track of relevant vehicles or relevant vertices during different searches, for instance in elliptic BCH searches and last stop BCH searches (see Section 5.1 and Section 5.2). Various threads access and update this subset of vehicles or vertices concurrently. For this purpose, we design a thread-safe subset data structure that supports concurrent membership queries and insertions in constant time.

The data structure is implemented using `thread_safe_reset_flag_array` from Mt- KaHyPar [13] to mark membership of elements in the subset and `TBB`'s `concurrent_vector` to store an iterable list of members. When inserting an element e into the thread-safe subset, the current thread will call `compare_and_set_to_true` on the flag associated with e in the flag array. This method atomically attempts to set the flag to true and returns either true or false, depending on whether the calling thread successfully changed the value of the flag to true or not. Thus, even with concurrent calls to insert the same element e, there is exactly one thread that succeeds in setting the respective flag. Only this thread may insert e into the element vector, guaranteeing that each element in the subset appears in the vector exactly once. To allow concurrent growth on the elements of the subset, the container `concurrent_vector` is employed.

Thanks to the use of the `thread_safe_reset_flag_array`, we can insert an element, check whether an element is in the subset, and clear the subset all in constant time. The fast reset flag array that is used to mark membership of the subset allows random access to its elements in $O(1)$ and can reset all values to false by simply incrementing an internal threshold. The fact that we can clear the elements in $O(1)$ is important for our use since the thread-safe subset must be cleared before processing a new incoming request. To iterate through the elements, the `concurrent_vector` is used, which enables iteration in linear time.

6.5 Updating Global Best Assignment

The best known cost and the best known assignment are stored globally for each request. During parallel enumeration, as in the case of PALS, DALS, and PBNS (see Section 5.2.2 and Section 5.3.2), multiple threads may want to access and update these global best values concurrently. When a thread enumerates an assignment that satisfies all given constraints, it will try this assignment by comparing it to the current global best cost. Thus, access to the global best cost and assignment must occur thread-safely. We initially realized this by using a spin lock whenever a thread tries an assignment. If a thread wants to check whether the current assignment is better than the global best assignment, it must successfully acquire the lock. This could be solved with atomic operations if we were only looking for the lowest cost. But since we also have to update the entire assignment, we need a lock instead of atomics. Using a lock might cause a lot of contention and delay if the different threads constantly want to access the global best values.

Alternatively, each thread can store a local best cost and a local best assignment in its own local storage. These values are initialized to the global best known cost and best known assignment before the parallel execution. We compare the calculated cost of the current assignment with the local best cost and update it locally for each thread only. When all threads have concluded enumerating assignments, we can iterate through the threads' local best assignments and find the best assignment among them. As a result, the global best assignment is updated sequentially at the end of the enumeration process, and no lock has to be employed to ensure the thread-safety of these updates.

6.6 Collecting Statistics

Throughout the searches, statistics are collected for evaluation, these include, for example, the number of entries scanned, the number of edge relaxations, or the number of vertices visited. These statistics are originally stored in integer variables, which are accessed and incremented during the searches. Parallelism can lead to race conditions on these variables, resulting in one thread accessing an outdated value while another thread is incrementing it. Therefore, it is crucial to ensure thread-safe access to these variables.

For statistics that are only accessed once per search, the integer variable can be simply made atomic using the C++ standard library. For statistics that are updated more often during a single search, we transform them to be stored in thread local storage similarly to searches' data structures by using `TBB`'s `enumerable_thread_specific` as mentioned in Section 6.1. The statistics will be collected and added up individually within the thread local storage during the searches. At the end of all searches, we will combine these values across all participating threads to determine the total number. The `enumerable_thread_specific` template enables iterating through the local thread values and setting them, which is helpful for resetting the statistics during initialization of searches.

7 Evaluation

With the addition of our suggested parallel algorithm, we expand the codebase authored by Laupichler and Sanders for KaRRi [23]. Our code[1] is written in C++17 and compiled with GCC 9.4 at optimization level `-O3`. In this work, we utilize parallel primitives from the Intel oneAPI Thread Building Blocks (`TBB`) toolset (version 2021.11.0).

Hardware Environment. We use two computers for our experiments:

- `Machine A`: 2 x Intel Xeon E5-2683 v4 - 2 x 16 core processor @ 2.1GHz, 512GB DDR4-RAM @ 2133MHz, 40MB L3 Cache, AVX2 (256-bit SIMD instructions)
- `Machine B`: AMD EPYC Rome 7702P - 64 core processor @ 2.0-3.35GHz, 1024GB DDR4-RAM @ 2966MHz, 256MB L3 Cache, AVX2 (256-bit SIMD instructions)

We run experiments on two different computers in order to evaluate parallel execution with different numbers of physical CPU cores. The parameter tuning experiments are run on `Machine A` with a smaller number of cores, while the scalability experiments are run on `Machine B` in order to scale the level of parallelism. Note that on Machine A we only ever use a single processor node, meaning a maximum of 16 threads at a time. The maximum parallelism applied, i.e., the maximum number of threads used, is set by `TBB`'s `global_control` and `max_allowed_parallelism` accordingly. In this chapter, we denote the number of threads used as T.

Input Instances. Similar to KaRRi, we assess our algorithm using the following request sets: `Berlin-1pct` (`B-1%`), `Berlin-10pct` (`B-10%`), `Ruhr-1pct` (`R-1%`), and `Ruhr-10pct` (`R-10%`), which represent 1% and 10% of the weekday demand for taxi sharing in the Berlin and Rhein-Ruhr metropolitan regions, respectively [6]. The request sets for Berlin were artificially created with the help of the Open Berlin Scenario [48] for the MATSim transportation simulation [41]. We randomly select the request dates and dropoff locations for the Rhein-Ruhr request sets, utilizing distributions that provide comparable travel durations and request densities to those in the Berlin request sets. The underlying vehicle and pedestrian road networks are extracted from OpenStreetMap data[2]. For the vehicle network, we calculate the travel time of the corresponding edge using the speed restriction known for each route. For the passenger network, we assume that pedestrians walk at a steady pace of 4.5 km/h. It takes less than a minute to calculate the contraction hierarchies of the road networks for our instances using the open-source RoutingKit[3] software.

[1]Available at `https://github.com/molaupi/karri`

[2]`https://download.geofabrik.de/`

[3]`https://github.com/RoutingKit`

Table 7.1: Key figures of our benchmark instances. Shows number of vertices ($|V|$), edges ($|E|$), vehicles (#veh.), and requests (#req.). In addition, shows average number (rounded down) of pickups (N_ρ^p) and dropoffs (N_ρ^d) for walking radius $\rho \in \{300s, 600s, 900s\}$ on the `Berlin-1pct`, `Berlin-10pct`, `Ruhr-1pct` and `Ruhr-10pct` instances.

Instance	$\|V\|$	$\|E\|$	#veh.	#req.	$\rho = 300\text{s}$		$\rho = 600\text{s}$		$\rho = 900\text{s}$	
					N_ρ^p	N_ρ^d	N_ρ^p	N_ρ^d	N_ρ^p	N_ρ^d
`B-1%`	94422	193212	1000	16569	44	44	178	178	403	403
`B-10%`	94422	193212	10000	149185	46	46	183	186	415	421
`R-1%`	420700	887790	3000	49707	40	39	137	136	279	279
`R-10%`	420700	887790	30000	447555	40	39	137	136	279	280

As mentioned earlier, we assume all riders in our simulations walk to the meeting points. Thus, in order to scale the number of relevant pickups and dropoffs, we use different walking radii $\rho \in \{300s, 600s, 900s\}$. Key figures of the input instances are shown in Table 7.1. We conduct all experiments in five iterations and show the average runtimes of all five runs.

We use the same cost function as KaRRi (see Section 2.2) and also utilize the same parameters. A *time is money* approach is applied with $\tau = 1$ and $\omega = 0$. The parameter τ accounts for the equal weighting of a driver's and a rider's time, while ω makes sure that we do not penalize walking over driving. Based on the MATSim transport simulation, we select $\alpha = 1.7$ and $\beta = 2min$, indicating a maximum travel duration of $1.7 \cdot dist(orig, dest) + 2min$ for each journey. We select the wait time violation weight $\gamma_{wait} = 1$, the trip time violation weight $\gamma_{trip} = 10$, and the maximum wait time $t_{wait}^{max} = 300s$.

7.1 Parameter Tuning

In this section, we carry out specific experiments on `Machine A` in order to find the set of parameters that optimizes the performance of our parallel algorithm. We conduct our parameter tuning experiments on the `Berlin-1pct` instance with $\rho = 600s$ and run experiments with 8 and 16 threads. We consider each phase of the parallel algorithm separately in the subsequent subsections and test out the different parameters associated with the individual phases.

7.1.1 Elliptic BCH Searches

For elliptic BCH searches, there are three variants of query result allocation that could be applied: static allocation, semi-static allocation, and dynamic allocation (see Section 6.2).

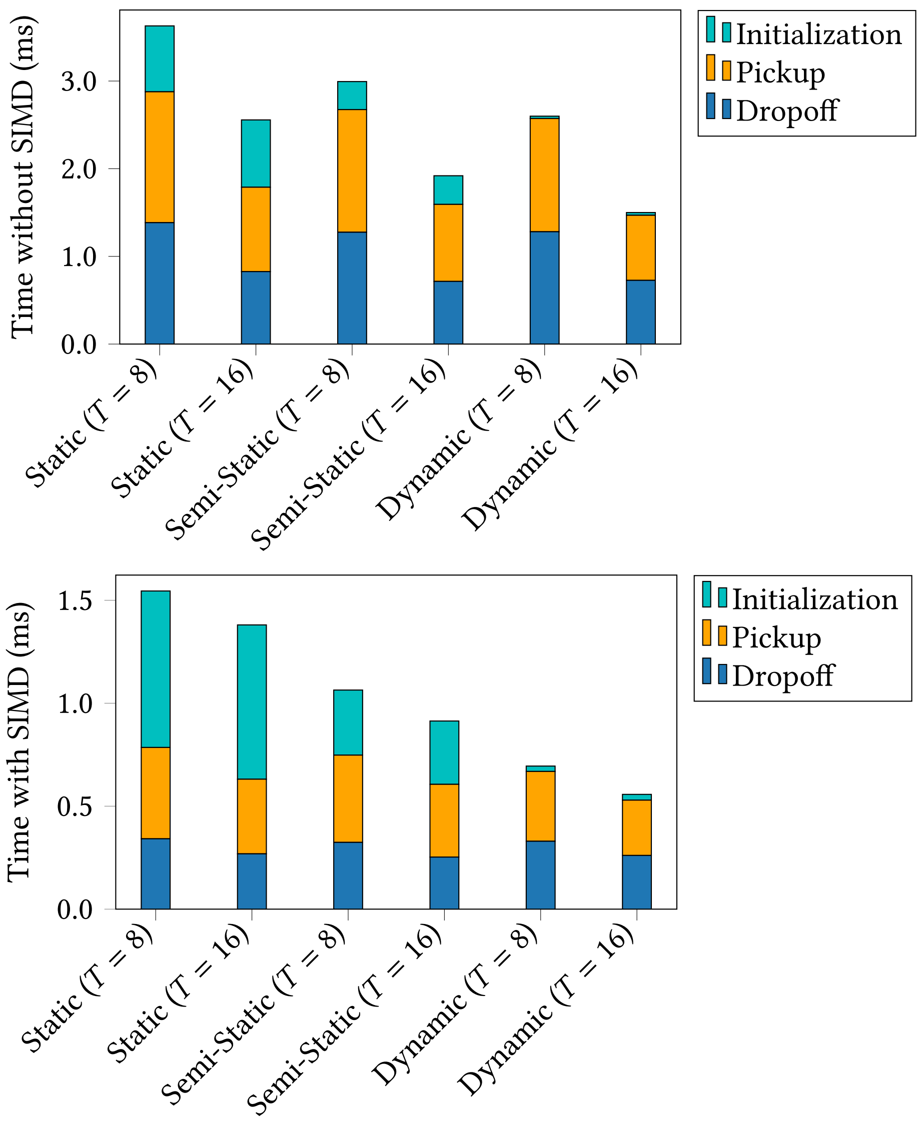

Figure 7.1: Runtimes for phases of elliptic BCH searches (initialization, pickup, dropoff) with different allocation types (static, semi-static, dynamic) and maximum number of threads ($T \in \{8, 16\}$). Runtime without SIMD instructions (top), runtime with SIMD instructions, $K = 16$ (bottom). Experimented on `Berlin-1pct` with $\rho = 600s$. Note the different y-axes.

We show experimental runtimes of elliptic BCH searches with and without SIMD instructions for the different allocation schemes in Figure 7.1. Overall, the bundled elliptic BCH searches with SIMD instructions ($K = 16$) perform significantly better than without bundling. A common pattern that can be seen in both variants is that dynamic allocation is the most effective among the three types of allocation.

For the static allocation, the initialization time takes up a large share of the total runtime. This can be reasonably explained, as the shared global results have to be cleared and statically allocated to the required size before the searches. The initialization time for

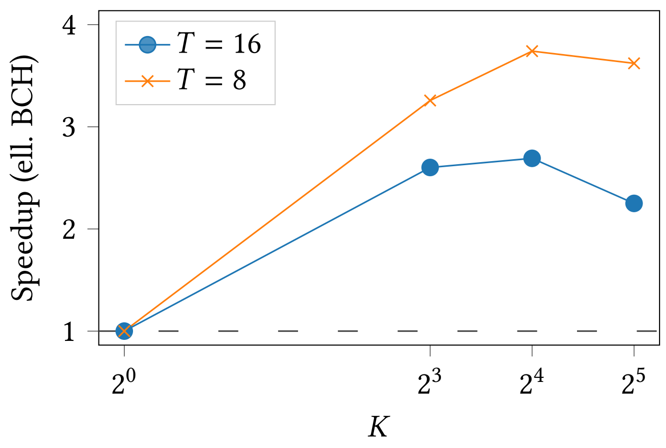

Figure 7.2: Speedup for elliptic BCH searches with dynamic allocation on different number of threads ($T \in \{8, 16\}$) when using SIMD instructions ($K \in \{1, 8, 16, 32\}$). Experimented on `Berlin-1pct` with $\rho = 600s$.

semi-static allocation is longer than that for dynamic allocation due to the need to run two preliminary searches in advance to pre-allocate relevant entries. The pickup and dropoff search times remain similar across the different allocation types. This indicates that even in scenarios with dynamic allocation, where potential contention between threads on parallel data structures may arise, the issue is effectively avoided. The improved cache efficiency achieved through dynamically allocated entries seems to outweigh the negative impact of contention, resulting in searches being equally fast or even slightly faster with dynamic allocation. Overall, dynamic allocation consistently yields the optimal runtime in all configurations. It can also be seen that a larger number of threads improves the runtime for all three allocation types.

Additionally, we show in Figure 7.2 how SIMD instructions can speed up bundled elliptic BCH searches with dynamic allocation over a range of $K \in \{1, 8, 16, 32\}$. The experiments are run on 8 and 16 threads. The best speedup can be observed with $K = 16$. For a larger K of 32, the speedup slightly decreases on both 8 and 16 threads. Elliptic BCH searches extensively examine the complete CH search space of each source, resulting in a significant amount of work being done in the peripheral regions of the sources. The sources' close proximity to one another makes effective bundling possible because it causes their search spaces to grow identical over greater distances. Larger values of K, however, result in overheads for edge relaxations and bucket entry scans that are closer to the sources and may not be bundled well. $K = 16$ strikes a balance between the two aspects, leading to the best speedup of 3.7 for 8 threads and 2.7 for 16 threads.

7.1.2 PALS and DALS

For PALS and DALS assignments, our algorithm runs individual BCH searches and the enumeration of assignments interleaved in parallel. The individual thread measures and sums up the local search and local try assignment times separately. Since threads work in

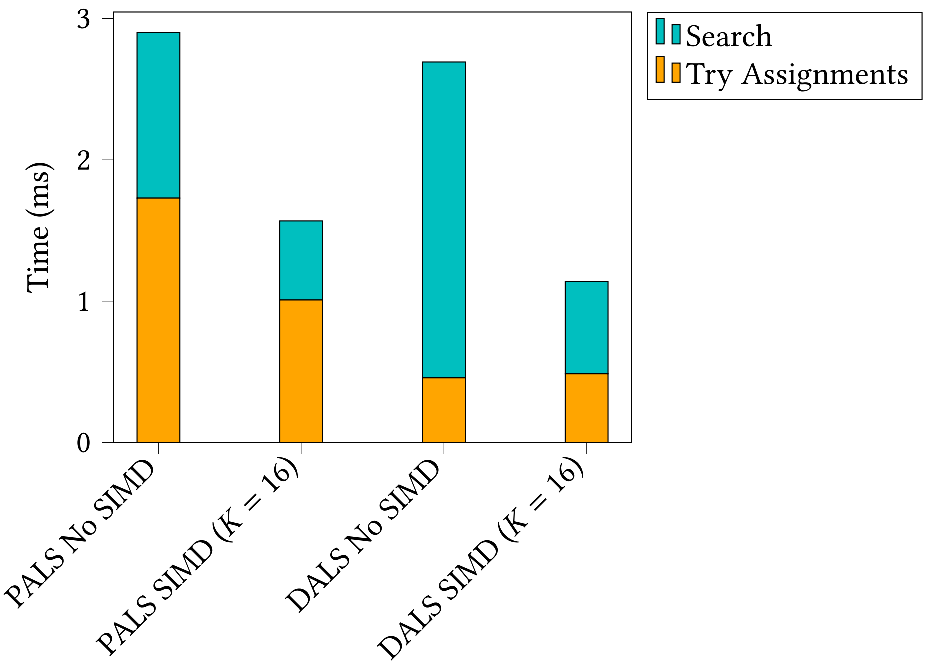

Figure 7.3: Sum of thread local runtimes (search time, try assignments time) for PALS and DALS assignments without SIMD instructions and with SIMD instructions ($K = 16$). Experimented on `Berlin-1pct` with $\rho = 600s$ and $T = 8$.

parallel, their running times overlap. The sum of local times is hence not comparable to the times measured in sequential. We compare these local times between PALS and DALS assignments with each other in Figure 7.3 with and without bundling searches.

It is noticeable that the local search time takes up more in the DALS case. This is due to the fact that the last stop BCH searches in the DALS case cannot include the PD-distance when calculating the lower bound cost as in the PALS case. Therefore, the searches must expand to a wider radius before the lower bound cost surpasses the best known cost, allowing for their termination. The local runtimes decrease when we additionally exploit SIMD parallelism with $K = 16$. Then, each single thread runs K searches and enumerates the results of these searches together in an iteration. The application of SIMD instructions during the searches significantly accelerates the thread's local search time. The individual thread enumerates the assignments of the K searches in sequential. However, we can still observe an improvement in the local try assignment times in the case of PALS. This may presumably be caused by the fact that the threads then enumerate more assignments in one run, which reduces the number of context switches between searches and enumerations.

We depict speedups for bundled individual BCH searches for the PALS and DALS assignments in Figure 7.4. Here, the speedups correspond approximately to the observations in the sequential algorithm KaRRi [23]. Bundling individual BCH searches does not yield significant improvements, as the majority of the work occurs near the sources. The KaRRi algorithm sorts last stop buckets, which results in most bucket entries being scanned at vertices near the source. Due to the usage of shortcut edges, the search spaces of different

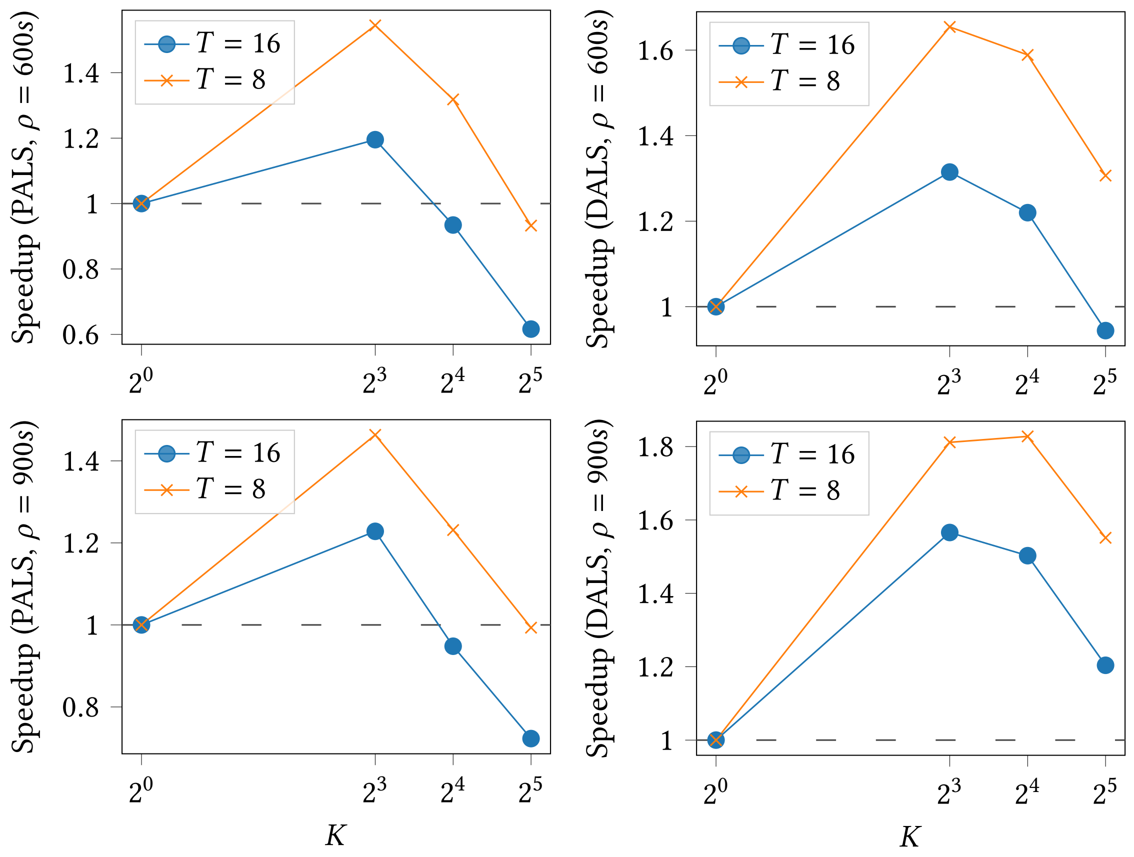

Figure 7.4: Speedup for PALS (left) and DALS (right) assignments on different number of threads ($T \in \{8, 16\}$) when using SIMD instructions ($K \in \{1, 8, 16, 32\}$). Experimented on `Berlin-1pct` with $\rho = 600s$ (top) and $\rho = 900s$ (bottom). Note the different y-axes.

individual BCH searches only overlap at further edges from the sources. Therefore, most tasks carried out by individual last stop searches cannot be bundled well. In the case of DALS, however, as the stopping criterion is worse, the BCH searches must expand a larger radius, which results in more work being done further away from the sources. This, in turn, enhances the bundling efficiency of DALS searches, accounting for the better speedup compared to PALS searches.

Furthermore, when performed in parallel with vector instructions, the searches become increasingly ineffective with larger Ks. With SIMD instructions, we can obtain instruction-level parallelism by using vector instructions, but only to a certain extent since the vector registers only have a limited bit width. For large values of K, it may occur that not all K instructions fit inside a single vector instruction, and we have to utilize multiple vector instructions instead of one. This leads to an increase in unnecessary work done near the sources with multiple vector instructions, which causes a waste of resources. Moreover, since the number of PD-locations is limited by the radius ρ, the parallelism over the PD-locations cannot fully utilize resources when using larger Ks.

Apart from the standard radius $\rho = 600s$ for parameter tuning, we experimented with radius $\rho = 900s$ to increase the number of PD-locations for better parallelism, as shown in Figure 7.4. With a walking radius of $\rho = 600s$, there are about 180 pickups and 180 dropoffs, whereas with $\rho = 900s$, there are roughly 400 of each. We observe similar trends with both radii, but the speedup is slightly better in experiments with $\rho = 900s$ in the lower two plots. The speedups of DALS assignments improved slightly with a larger number of PD-locations, while the speedups of PALS assignments remained relatively the same. It can be seen in the plots in Figure 7.4 that with $K = 8$, the bundled individual searches are the most effective. For PALS assignments, the speedup remains approximately 1.5 when run on 8 threads and 1.2 when run on 16 threads for both radii. For DALS assignments, with $\rho = 600s$, we achieve a speedup of 1.6 on 8 threads and 1.3 on 16 threads. This speedup is improved when run with $\rho = 900s$, namely 1.8 on 8 threads and 1.6 on 16 threads. In this case, $K = 8$ instructions still fit exactly into a single vector instruction. With larger Ks, the speedup decreases as an effect of the aforementioned problems when multiple vector instructions are used for K instructions, resulting in a waste of resources.

7.1.3 PBNS

In the case of PBNS, we search for the distances between the vehicle's current location and the relevant pickups (see Section 5.3). We utilize a lower bound cost to only calculate the distances that allow a feasible assignment. Thus, there are not as many pickup locations for parallelism as in the other cases. In other words, the workload is not enough for the application of thread parallelism together with instruction parallelism. We find that it is best to run single searches for the PBNS assignments instead of bundling them, which may result in extra unwanted work. Nonetheless, thread parallelism can still be beneficial if there are sufficient pickups for the threads to process.

7.1.4 PD-Distance Searches

For PD-distance searches, two search strategies may be applied, namely the BCH and the CH strategy (see Section 5.4).

We compare the runtime of PD-distance searches when run with BCH queries and CH queries with and without bundling in Table 7.2. In both bundled (with SIMD, $K = 16$) and non-bundled (without SIMD) searches, the CH strategy is remarkably slower than the BCH strategy. Even though employing more threads, as seen in the case of $T = 16$, can halve the runtimes of CH queries, they still remain considerably slower than BCH queries. For both the BCH and CH strategies, the initialization time is negligible in comparison to the search time. This phase involves clearing the distance results from the previous request. The CH queries only run pickup searches and therefore do not have the preprocessing time to generate dropoff bucket entries like BCH queries.

Table 7.2: Comparison of the PD-distance searches running times (in μs) with different maximum number of threads ($T \in \{8, 16\}$) using BCH searches (BCH) and CH searches (CH), with and without SIMD instructions. Shows average times for initialization phase (t_{init}), preprocessing phase (dropoff bucket generation in BCH searches) ($t_{preprocessing}$) and search phase (t_{search}).

T	Instructions Type	Search Type	t_{init}	$t_{preprocessing}$	t_{search}	t_{total}
8	with SIMD ($K = 16$)	BCH	20.5	383.0	288.4	691.9
		CH	20.5	0	10409.4	10429.9
	without SIMD	BCH	17.3	600.8	917.9	1535.9
		CH	17.5	0	48859.7	48877.3
16	with SIMD ($K = 16$)	BCH	20.4	385.2	226.2	631.7
		CH	20.9	0	5491.5	5512.5
	without SIMD	BCH	18.0	500.5	517.6	1036.2
		CH	18.0	0	24794.9	24812.9

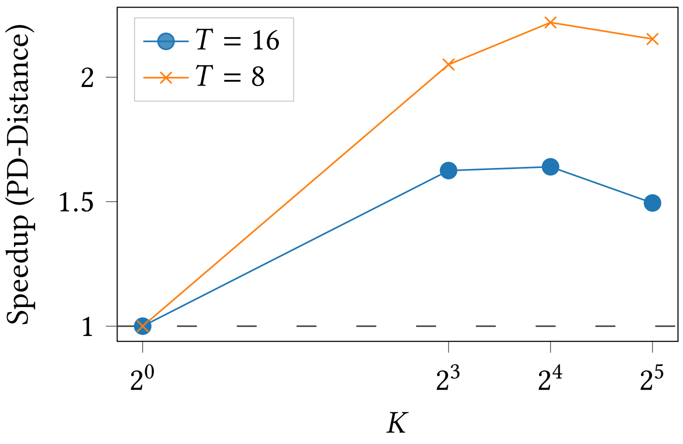

Figure 7.5: Speedup for PD-distances with BCH Searches on different number of threads ($T \in \{8, 16\}$) when using SIMD instructions ($K \in \{1, 8, 16, 32\}$). Experimented on `Berlin-1pct` with $\rho = 600s$.

We consider the BCH strategy for the next experiments, as it is the strategy with the noticeably better runtime. We show the experimental speedups of bundled BCH PD-distance searches with 8 and 16 threads over a number of K values in Figure 7.5. A similar behavior to bundled elliptic BCH searches can be observed here, where $K = 16$ produces the best speedup for both $T = 8$ and $T = 16$, with speedups of 2.2 and 1.6, respectively. Since the PD-distance queries search for the distances between the localized pickup and dropoff locations, bundling the searches benefits from similar arguments.

7.2 Scalability

In order to evaluate the scalability of our parallel algorithm, we run experiments with different numbers of maximum threads to compare the speedup achieved on `Machine B` - a computer with 64 physical cores. For scalability experiments, we run the algorithm on the `Berlin-10pct` instance with walking radius $\rho = 900s$ to ensure sufficient workload for the available number of threads. Here, we deliberately do not apply SIMD instructions in order to solely measure the effect of thread parallelism.

In this section, we explore the strong scalability of our algorithm over the increasing number of threads and the weak scalability on larger input instances – in our case, the number of PD-locations. The speedups computed here are measured against the baseline of the sequential KaRRi algorithm, employing individual BCH searches for last stop searches and BCH strategy for PD-distance searches, mirroring the configuration used in our parallel algorithm.

7.2.1 Strong Scaling

Our analysis focuses on comparing the runtimes of the parallel algorithm across varying numbers of threads $T \in \{8, 16, 32, 64\}$, in contrast to the sequential KaRRi algorithm with the same configuration. We depicted the total runtimes of the algorithm when run on different numbers of threads in a box plot in Figure 7.6. Here, we left out the outliers for better visualization and therefore included the maximum runtime of each number of threads on top of the box plot to provide the missing information. It can be seen that the runtime gradually decreases with an increasing number of threads. Nonetheless, the decreases in the higher number of threads are minimal and not as noticeable as the decreases in the lower number of threads. For instance, the difference between 32 and 64 threads is smaller than the difference between 8 and 16 threads. The arithmetic mean runtimes for $T \in \{1, 8, 16, 32, 64\}$ are 103, 32, 23, 18, 16 (ms), respectively.

Additionally, we show the geometric mean speedups of the different phases with parallelism and the total runtime in Table 7.3. The two phases with the best speedups are elliptic BCH searches and PD-distance searches, yielding both a speedup of roughly 10 on 64 threads. Here, the searches for each PD-location are completely independent from one another, which makes them suitable for our parallel implementation. In the case of PALS and DALS, the achieved speedups are not as significant, with a speedup of 3.55 for PALS and 2.93 for DALS on 64 threads. The parallel individual last stop BCH searches run best on requests with a larger number of PD-locations, which is not the average case in our dataset. In total, the parallel algorithm yields a speedup of approximately 5 when run on 64 threads in comparison to the sequential algorithm. Even though this speedup is far from the theoretical optimum, this can be presumably explained as the average request does not have a great number of PD-locations for parallelism. Our parallel algorithm works well on a larger number of PD-locations, which is shown in Section 7.2.2.

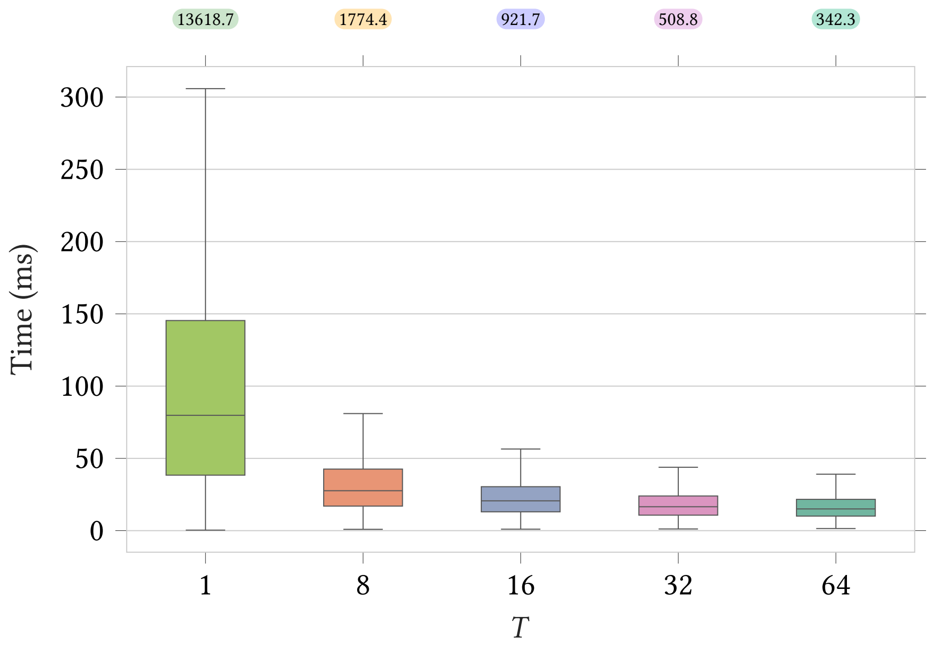

Figure 7.6: Total runtimes with different number of threads ($T \in \{1, 8, 16, 32, 64\}$). This plot does not include outliers, maximum runtime for each number of thread written at the top of the corresponding box. Experimented on `Berlin-10pct` with $\rho = 900s$ and without SIMD instructions.

Table 7.3: Geometric mean speedups on the `Berlin-10pct` instance with $\rho = 900s$ for the different phases, including elliptic BCH searches (ell. BCH), PD-distance searches (PD), PALS assignments (PALS), DALS asssignments (DALS) and for the total runtime of the algorithm (total).

T	ell. BCH	PD	PALS	DALS	total
8	3.52	4.20	1.90	1.23	2.74
16	5.49	6.02	2.64	1.99	3.69
32	8.05	8.26	3.37	2.83	4.57
64	9.96	9.68	3.55	2.93	4.95

7.2.2 Weak Scaling

Since our algorithm parallelizes the searches over the number of PD-locations, we evaluate the speedups in relation to the number of PD-locations in different phases. Each phase corresponds to an individual metric of PD-locations, as the searches are parallelized differently to find various distances. In Figure 7.7, the speedups of PALS and DALS assignments are plotted in relation to the number of pickups N_ρ^p and dropoffs N_ρ^d, respectively. In Figure 7.8, we show the speedup of elliptic BCH searches in relation to the total number of pickups and dropoffs together $N_\rho^p + N_\rho^d$, since these searches are run in parallel once over the pickups and once over the dropoffs. For PD-distance searches, we search for distances

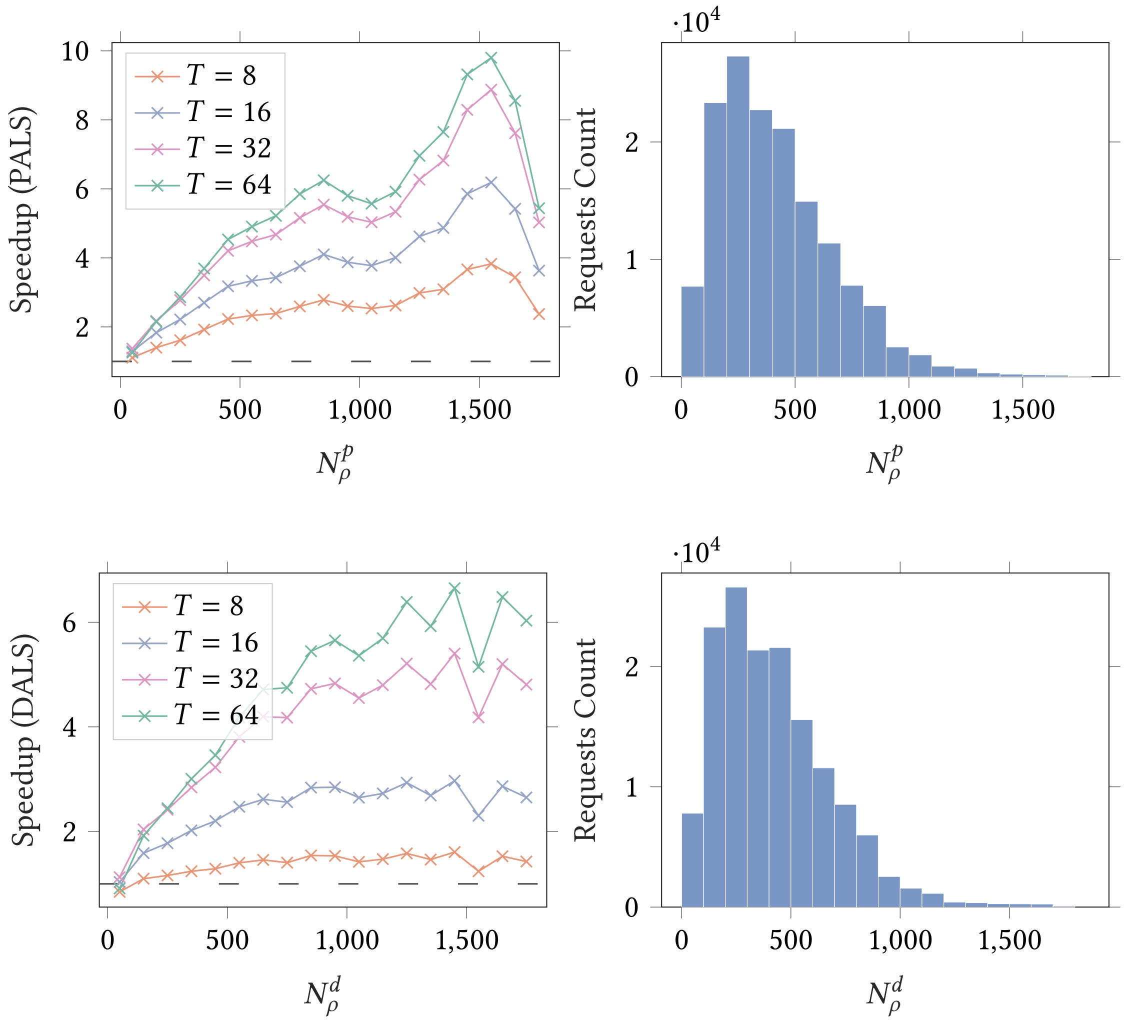

Figure 7.7: Speedup for PALS assignments in relation to the number of pickups (N_ρ^p) (top left). Speedup for DALS assignments in relation to the number of dropoffs (N_ρ^d) (bottom left). Distribution of requests over N_ρ^p (top right) and over N_ρ^d (bottom right), next to the corresponding speedup plot. Experimented on `Berlin-10pct` with $\rho = 900s$, without SIMD instructions and on different number of threads ($T \in \{8, 16, 32, 64\}$). Note the different y-axes.

between every pickup and every dropoff. Hence, the speedup is shown in relation to the product of the number of pickups and the number of dropoffs $N_\rho^p \cdot N_\rho^d$.

In all four phases, we divide the x-axis into 18 segments of the same size, depending on the metric range. Each point in the plot represents the geometric mean speedup among all requests, which fall into a specific segment. The histogram next to each speedup plot displays the distribution of the number of requests over the segments of the x-axis.

In the case of PALS and DALS, the distributions of the numbers of pickups and dropoffs per request are roughly equal, with the majority of the requests having less than 500 relevant

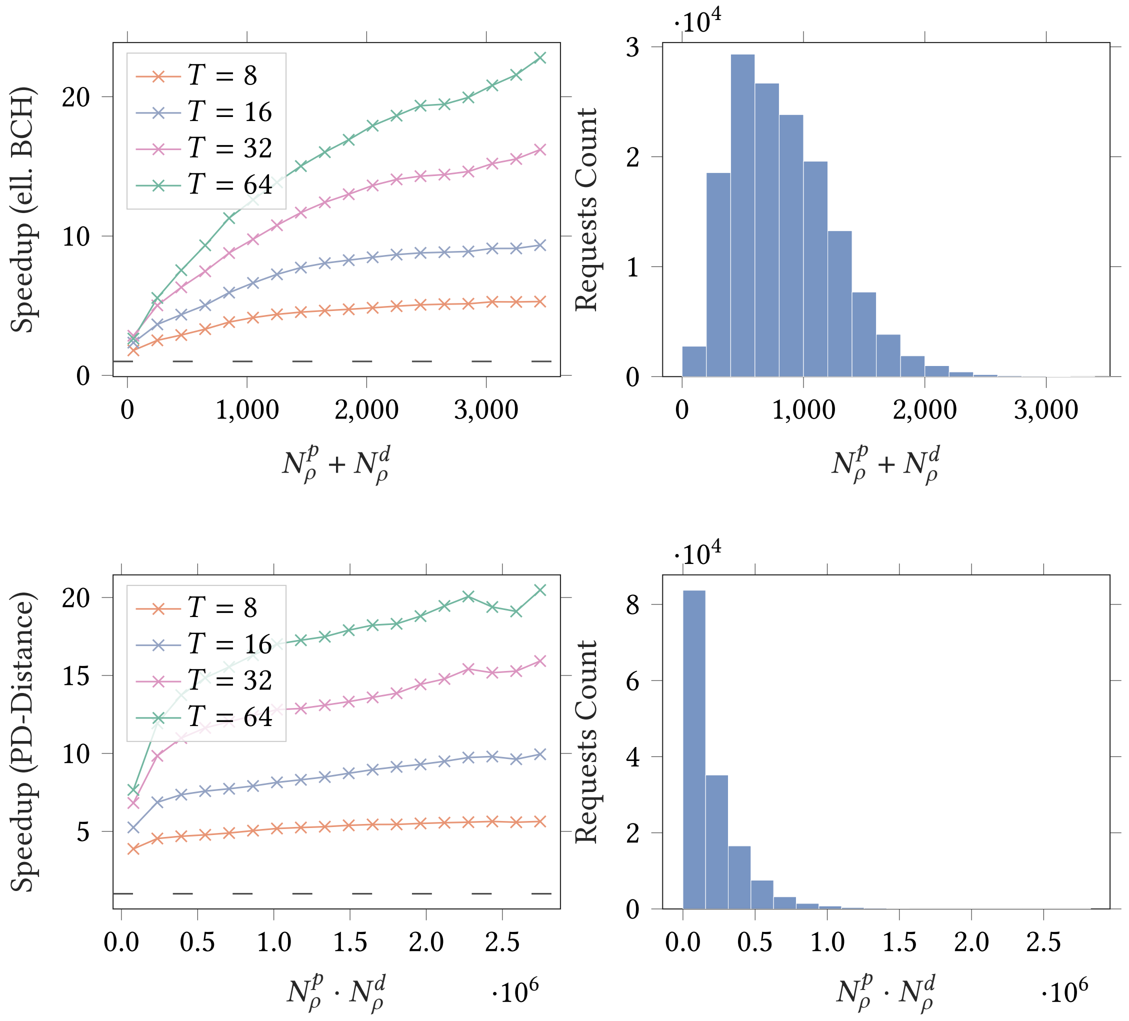

Figure 7.8: Speedup for elliptic BCH searches in relation to the sum of pickups and dropoffs ($N^p_\rho + N^d_\rho$) (top left). Speedup for PD-distance searches in relation to the product of pickups and dropoffs ($N^p_\rho \cdot N^d_\rho$) (bottom left). Distribution of requests over $N^p_\rho + N^d_\rho$ (top right) and over $N^p_\rho \cdot N^d_\rho$ (bottom right), next to the corresponding speedup plot. Experimented on `Berlin-10pct` with $\rho = 900s$, without SIMD instructions and on different number of threads ($T \in \{8, 16, 32, 64\}$). Note the different y-axes.

pickups and dropoffs. For these requests, however, the speedup remains rather low for both PALS and DALS assignments, with a gradual increase over the number of pickups and dropoffs. The small number of PD-locations here restricts the level of parallelism that may be applied.

For requests with a larger number of pickups and dropoffs, we observe considerable speedups, with a high of 10 in PALS assignments and a high of 6.6 in DALS assignments when run on 64 threads. Nonetheless, the number of requests with a larger number of relevant PD-locations is noticeably small. This indicates that the limited overall speedups are primarily a result of insufficient workload rather than the parallelization process itself. One can also observe that the increasing trend is similar over different numbers of threads. Due to the limited amount of data, the speedup values are generally less reliable with higher numbers of pickups (dropoffs). This explains the drop at the very end of PALS and the drop at around 1550 dropoffs of DALS.

In the case of elliptic BCH searches, the distribution of the total number of pickups and dropoffs is similar to that of the pickups and dropoffs separately. The speedups on the different number of threads $T \in \{8, 16, 32, 64\}$ steadily rise over the sum of pickup and dropoff locations, peaking at 23 on 64 threads.

For PD-distance searches, we plot the speedups against the product of the number of pickups and dropoffs $N^p_\rho \cdot N^d_\rho$. The first x-axis segment includes the majority of requests. The speedups here lie around 4 to 8 over the different number of threads. Then, the speedups increase marginally, or may even stop to increase at some point. This demonstrates the practical limit on the speedup that may be attained by the parallelization of PD distance searches, given sufficient work. With 8 threads, for example, it is impossible to attain a speedup greater than 5. On 64 threads, the speedup reaches a maximum of around 20 in the last segment.

All in all, the parallel execution has proven its effect, evident in both the total runtime and individual phases as described in this section. The speedup increases with the number of threads T and the number of PD-locations N^p_ρ and N^d_ρ.

7.3 Comparison to KaRRi

Based on the results of the previous sections on parameter tuning and scalability experiments, we found the best configuration for our parallel algorithm. This configuration includes dynamic allocation for the elliptic BCH searches, BCH strategy for the PD-distance searches, and parallel individual BCH searches for PALS and DALS. We run our parallel algorithm on 64 threads, as a result of Section 7.2, in order to optimize performance. In this final section, we compare our parallel algorithm to the sequential KaRRi algorithm and show the overall improvements.

We run the experiments on all instances in Table 7.1: `B-1%`, `B-10%`, `R-1%`, `R-10%`, in combination with different walking radii $\rho \in \{300s, 600s, 900s\}$. The sequential KaRRi is run with its best configuration, which utilizes collective BCH searches for PALS and DALS assignments (see Section 4.2.2), whereas our algorithm utilizes parallelized individual BCH searches. Based on Section 7.1, we apply SIMD instructions with $K = 8$, since this value of K produces a good speedup for all search types and strikes a balance for the smaller number of PD-locations. Furthermore, we experimented with a combined version of KaRRi and the parallel algorithm, in which the last stop BCH searches are run in sequential with KaRRi's collective searches and the remaining searches are run in parallel with our algorithm.

We give the running times for the different phases of the sequential KaRRi algorithm (Ka), our parallel algorithm (P), and the combined variant (Ka + P) on all instances in Table 7.4. For radius $\rho = 300s$, sequential KaRRi still remains the fastest for all instances. This can be addressed by the fact that the number of PD-locations is quite limited with radius $\rho = 300s$. Here, there are less than 50 pickups and less than 50 dropoffs, while we allow the maximum number of threads to be 64. As a consequence, the available resources cannot be utilized, and the parallel variant remains slower than the original sequential variant. For larger radius $\rho \in \{600s, 900s\}$, it can be seen that the parallel algorithm outperforms its sequential counterpart in most cases, with the exception of instance `Ruhr-10pct` with $\rho = 600s$. This can be explained, as the Ruhr instances are less dense and thus have a smaller number of PD-locations for parallelism. The parallel algorithm enhances significantly the runtime of PD-distance searches and elliptic BCH searches, as these searches are well suited for parallelizing over the PD-locations.

In the case of PBNS, however, since we have a restricted number of relevant pickups, there is usually not enough work to distribute for the individual threads (see Section 7.1.3). Moreover, we notice that PBNS does not scale well with the number of vehicles. It can be observed in Table 7.4 that the runtime for PBNS is only better in parallel on the `B-1%` instance with 1000 vehicles. In P, vehicles requiring bucket calculations are determined simultaneously in parallel, followed by sequential bucket construction for all vehicles and parallel queries for all pickups (see Section 5.3). In contrast, Ka iterates over vehicles, determining required pickups and calculating bucket entries for each vehicle sequentially, followed by queries for each vehicle and its pickups. The determination of which distances to calculate is based on the best known cost, meaning fewer distances need to be calculated with better known costs. The sequential algorithm updates the best cost after calculating each vehicle, thereby reducing the number of buckets generated for subsequent vehicles. Whereas in the parallel algorithm, the distances are determined simultaneously for multiple vehicles, and there is no opportunity to incorporate updated costs during the determination process. Consequently, parallel filtering may lead to generating bucket entries for more vehicles if no suitable insertion other than PBNS is found. Hence, when run in parallel on instances with a larger number of vehicles, the PBNS runtimes are generally larger.

We attempted to parallelize the individual last stop BCH searches for PALS and DALS assignments, which has proven an improvement when compared to the sequential individual last stop BCH searches in Section 7.2.2. Nonetheless, this type of search cannot beat the fast runtime of collective BCH searches introduced by KaRRi. This is the main reason why we additionally run experiments with a combined version of sequential KaRRi and our parallel algorithm, where the collective BCH last stop searches are in use instead of individual BCH last stop searches. The combined variant proves its enhancement, both to the sequential KaRRi algorithm and to our parallel algorithm.

In the combined algorithm Ka + P, the runtimes of PALS assignments still remain much higher than those in the original KaRRi algorithm, although we applied the same collective BCH searches here. The runtimes of DALS assignments, however, are quite similar and do not experience such a considerable increase when combined with the parallel algorithm. We assume that this is caused by `TBB` background tasks, as the PALS assignments are the first to be enumerated in the algorithm. It may be possible that `TBB`'s initialization overhead at the beginning of each request results in a noticeable increase in the runtime of this first phase.

7.4 Discussion

We conduct experiments to systematically investigate different parameters in the parallel algorithm and determine the best configuration for maximizing performance improvement (see Section 7.1). We explore the scalability of our algorithm with respect to the number of threads and the number of PD-locations (see Section 7.2). It is evident that the algorithm performs effectively in cases where there are numerous PD-locations, due to our parallel implementation over these locations. On average, our algorithm shows a modest speedup over the sequential counterpart, primarily because there is not enough work for the available resources.

When compared to the best configuration of KaRRi (see Section 7.3), which includes collective BCH searches, it can be seen in scenarios with a smaller number of PD-locations that the sequential KaRRi algorithm outperforms our parallel algorithm. Nevertheless, when dealing with a larger radius ρ and so a greater number of PD-locations, our parallel algorithm outperforms KaRRi, up to a factor of two. This overall speedup is mainly addressed to the significant speedups in elliptic BCH searches and PD-distance searches, whereas in special case insertions, our parallel algorithm falls short of the performance of the KaRRi algorithm.

Table 7.4: Running times (in μs) of different phases of the KaRRi algorithm (Ka), parallel algorithm (P), and combination of KaRRi and parallel algorithm (Ka + P) with different radii ($\rho \in \{300s, 600s, 900s\}$) on `B-1%`, `B-10%`, `R-1%`, and `R-10%`. Shows mean times for initialization, PD-distance searches, elliptic BCH searches, enumerating ordinary insertions, enumerating PBNS insertions, PALS and DALS searches as well as the mean total time per request. The smallest times per radius are marked in bold.

Inst.	ρ	Alg.	init.	PD	Ell.	Ord.	PBNS	PALS	DALS	total
`B-1%`	300	Ka	**213**	205	518	**12**	**73**	**1**	**38**	**1142**
		P	228	175	437	26	120	67	133	1287
		Ka+P	232	**174**	**432**	22	127	44	40	1165
	600	K	**609**	1192	2157	50	396	**5**	**66**	4563
		P	646	442	**560**	71	**335**	241	173	2575
		Ka+P	646	**439**	578	**66**	346	118	71	**2362**
	900	Ka	**1291**	4420	5223	**147**	1061	**18**	**101**	12358
		P	1330	1080	**756**	204	**735**	677	214	5108
		Ka+P	1332	**1079**	786	178	758	311	112	**4662**
`B-10%`	300	Ka	**241**	215	1323	**112**	**318**	**4**	**65**	**2399**
		P	268	**186**	**1706**	227	683	183	225	3632
		Ka+P	264	190	1742	196	657	31	68	3291
	600	Ka	**703**	1279	5329	**398**	**1388**	**39**	**106**	9380
		P	775	531	**2264**	579	2273	1011	334	7952
		Ka+P	760	**510**	2391	558	2238	111	118	**6859**
	900	Ka	**1449**	4818	13573	**1080**	**3687**	**142**	**154**	25057
		P	1558	1346	**2977**	1385	5084	3330	426	16312
		Ka+P	1523	**1337**	3179	1366	5061	375	179	**13213**

Continued on next page

Table 7.4: Running times (in μs) of different phases of the KaRRi algorithm (Ka), parallel algorithm (P), and combination of KaRRi and parallel algorithm (Ka + P) with different radii ($\rho \in \{300s, 600s, 900s\}$) on `B-1%`, `B-10%`, `R-1%`, and `R-10%`. Shows mean times for initialization, PD-distance searches, elliptic BCH searches, enumerating ordinary insertions, enumerating PBNS insertions, PALS and DALS searches as well as the mean total time per request. The smallest times per radius are marked in bold.

Inst.	ρ	Alg.	init.	PD	Ell.	Ord.	PBNS	PALS	DALS	total
`R-1%`	300	Ka	**162**	**143**	**527**	**21**	**81**	**1**	**31**	**1040**
		P	173	155	659	49	179	58	113	1476
		Ka+P	173	155	655	40	173	41	33	1353
	600	Ka	**422**	686	1733	**38**	**232**	**3**	**48**	3243
		P	450	**341**	**813**	68	391	153	155	2468
		Ka+P	449	344	845	60	388	115	53	**2344**
	900	Ka	**827**	2067	3612	**74**	**503**	**7**	**68**	7246
		P	880	700	**953**	109	739	325	192	4002
		K+P	874	**692**	1018	101	743	222	76	**3823**
`R-10%`	300	Ka	**188**	**148**	**2330**	**202**	**458**	**9**	**59**	**3537**
		P	209	173	4355	438	1628	208	283	7477
		Ka+P	206	173	4351	372	1537	34	65	6908
	600	Ka	**479**	704	7238	**313**	**1069**	**80**	**88**	**10132**
		P	520	440	**5547**	660	3862	555	389	12177
		Ka+P	507	**440**	5629	609	3780	95	100	11346
	900	Ka	**915**	2121	15135	**500**	**2174**	347	**118**	21481
		P	960	**898**	**6191**	842	6735	1263	455	17554
		Ka+P	939	911	6255	810	6594	**222**	135	**16057**

8 Conclusion

In this thesis, we introduce a parallel algorithm based on the dispatching algorithm KaRRi [23] for large-scale dynamic taxi sharing with meeting points. Employing fine-grained parallelism, we attempt to parallelize every phase of the original sequential algorithm. These phases include elliptic BCH searches, PD-distance searches, and special case insertions of PBNS, PALS, and DALS. We provide a high-level overview of our algorithm, describing each of the phases mentioned, and explain the implementation details, highlighting challenges encountered during parallelization.

Our evaluation reveals noticeable speed improvements compared to the sequential approach. Although the parallel individual last stop BCH searches do not match the effectiveness of KaRRi's collective last stop BCH searches, they still outperform the sequential last stop BCH searches considerably. Overall, combining our parallel algorithm with KaRRi's collective BCH searches produces a significant speedup, particularly evident in instances with a large number of meeting points.

While our parallel algorithm presents an initial step towards optimization, there exists potential for further enhancement through the integration of more advanced techniques. Our approach primarily focuses on parallelizing each phase of the sequential KaRRi algorithm using straightforward methods. Phases with subobtimal runtimes such as PBNS can be considered for further optimization. In addition, it is possible to enhance the algorithm's adaptability to real-time changes in traffic and road conditions by using customizable contraction hierarchies, similarly to KaRRi. Future work could explore adapting the algorithm to operate on distributed computing systems with distributed data, thereby extending its scalability and efficiency. There is also a promising opportunity for the parallelization of the dispatching algorithm across the numerous requests.

Bibliography

[1] Niels Agatz et al. “Optimization for Dynamic Ride-Sharing: A Review”. In: *European Journal of Operational Research* 223.2 (2012). DOI: 10.1016/j.ejor.2012.05.028.

[2] Niels A. H. Agatz et al. “Dynamic Ride-Sharing: A Simulation Study in Metro Atlanta”. In: *Transportation Research Part B: Methodological*. Select Papers from the 19th ISTTT 45.9 (2011). DOI: 10.1016/j.trb.2011.05.017.

[3] European Environment Agency. *Are We Moving in the Right Direction?: Indicators on Transport and Environment Integration in the EU; TERM 2000*. European Environment Agency, 2000.

[4] Kamel Aissat and Ammar Oulamara. “A Priori Approach Of Real-Time Ridesharing Problem With Intermediate Meeting Locations”. In: *Journal of Artificial Intelligence and Soft Computing Research* 4.4 (2014). DOI: 10.1515/jaiscr-2015-0015.

[5] Javier Alonso-Mora et al. “On-Demand High-Capacity Ride-Sharing via Dynamic Trip-Vehicle Assignment”. In: *Proceedings of the National Academy of Sciences* 114.3 (2017). DOI: 10.1073/pnas.1611675114.

[6] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. “Fast, Exact and Scalable Dynamic Ridesharing”. In: *2021 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. Proceedings. Society for Industrial and Applied Mathematics, 2021. DOI: 10.1137/1.9781611976472.8.

[7] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. “Real-Time Traffic Assignment Using Engineered Customizable Contraction Hierarchies”. In: *ACM Journal of Experimental Algorithmics* 24 (2019). DOI: 10.1145/3362693.

[8] Jean-François Cordeau and Gilbert Laporte. “The Dial-A-Ride Problem: Models and Algorithms”. In: *Annals of Operations Research* 153.1 (2007). DOI: 10.1007/s10479-007-0170-8.

[9] Marco Diana and Maged M. Dessouky. “A New Regret Insertion Heuristic for Solving Large-Scale Dial-A-Ride Problems with Time Windows”. In: *Transportation Research Part B: Methodological* 38.6 (2004). DOI: 10.1016/j.trb.2003.07.001.

[10] Dorian Dumez et al. “Hybridizing Large Neighborhood Search and Exact Methods for Generalized Vehicle Routing Problems with Time Windows”. In: *EURO Journal on Transportation and Logistics* 10 (2021). DOI: 10.1016/j.ejtl.2021.100040.

[11] Masabumi Furuhata et al. “Ridesharing: The State-of-the-Art and Future Directions”. In: *Transportation Research Part B: Methodological* 57 (2013). DOI: 10.1016/j.trb.2013.08.012.

[12] Robert Geisberger et al. “Exact Routing in Large Road Networks Using Contraction Hierarchies”. In: *Transportation Science* 46.3 (2012). DOI: 10.1287/trsc.1110.0401.

[13] Lars Gottesbüren et al. “Scalable High-Quality Hypergraph Partitioning”. In: (2023). DOI: 10.48550/arXiv.2303.17679.

[14] Carl H. Häll, Magdalena Högberg, and Jan T. Lundgren. “A Modeling System for Simulation of Dial-A-Ride Services”. In: *Public Transport* 4.1 (2012). DOI: 10.1007/s12469-012-0052-6.

[15] Moritz Hilger et al. “Fast Point-to-Point Shortest Path Computations with Arc-Flags”. In: *The Shortest Path Problem: Ninth DIMACS Implementation Challenge* 74.1 (2009). DOI: 10.1090/dimacs/074.

[16] Sin C. Ho et al. “A Survey of Dial-A-Ride Problems: Literature Review and Recent Developments”. In: *Transportation Research Part B: Methodological* 111 (2018). DOI: 10.1016/j.trb.2018.02.001.

[17] Mark E. T. Horn. “Fleet Scheduling and Dispatching for Demand-Responsive Passenger Services”. In: *Transportation Research Part C: Emerging Technologies* 10.1 (2002). DOI: 10.1016/S0968-090X(01)00003-1.

[18] Brady Hunsaker and Martin Savelsbergh. “Efficient Feasibility Testing for Dial-A-Ride Problems”. In: *Operations Research Letters* 30.3 (2002). DOI: 10.1016/S0167-6377(02)00120-7.

[19] Vipin Jain, Ashlesh Sharma, and Lakshminarayanan Subramanian. “Road Traffic Congestion in the Developing World”. In: *Proceedings of the 2nd ACM Symposium on Computing for Development.* ACM DEV ’12. Association for Computing Machinery, 2012. DOI: 10.1145/2160601.2160616.

[20] Nadira Jasika et al. “Dijkstra’s Shortest Path Algorithm Serial and Parallel Execution Performance Analysis”. In: *2012 Proceedings of the 35th International Convention MIPRO.* 2012.

[21] Jang-Jei Jaw et al. “A Heuristic Algorithm for the Multi-Vehicle Advance Request Dial-A-Ride Problem with Time Windows”. In: *Transportation Research Part B: Methodological* 20.3 (1986). DOI: 10.1016/0191-2615(86)90020-2.

[22] Sebastian Knopp et al. “Computing Many-to-Many Shortest Paths Using Highway Hierarchies”. In: *2007 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX).* Proceedings. Society for Industrial and Applied Mathematics, 2007. DOI: 10.1137/1.9781611972870.4.

[23] Moritz Laupichler and Peter Sanders. “Fast Many-to-Many Routing for Ridesharing with Multiple Pickup and Dropoff Locations”. In: arXiv:2305.05417 (2023). DOI: 10.48550/arXiv.2305.05417.

[24] Yeqian Lin et al. “Research on Optimization of Vehicle Routing Problem for Ride-Sharing Taxi”. In: *Procedia - Social and Behavioral Sciences.* 8th International Conference on Traffic and Transportation Studies (ICTTS 2012) 43 (2012). DOI: 10.1016/j.sbspro.2012.04.122.

[25] Charlotte Lotze et al. “Dynamic Stop Pooling for Flexible and Sustainable Ride Sharing”. In: *New Journal of Physics* 24.2 (2022). DOI: `10.1088/1367-2630/ac47c9`.

[26] Hui Luo et al. “Dynamic Ridesharing in Peak Travel Periods”. In: *IEEE Transactions on Knowledge and Data Engineering* 33.7 (2021). DOI: `10.1109/TKDE.2019.2961341`.

[27] Shuo Ma, Yu Zheng, and Ouri Wolfson. “T-share: A Large-Scale Dynamic Taxi Ridesharing Service”. In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 2013. DOI: `10.1109/ICDE.2013.6544843`.

[28] Tai-Yu Ma et al. “A Dynamic Ridesharing Dispatch and Idle Vehicle Repositioning Strategy with Integrated Transit Transfers”. In: *Transportation Research Part E: Logistics and Transportation Review* 128 (2019). DOI: `10.1016/j.tre.2019.07.002`.

[29] Kamesh Madduri et al. “An Experimental Study of a Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances”. In: *2007 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. Proceedings. Society for Industrial and Applied Mathematics, 2007. DOI: `10.1137/1.9781611972870.3`.

[30] Hiroshi Makino et al. “Solutions for Urban Traffic Issues by ITS Technologies”. In: *IATSS Research* 42.2 (2018). DOI: `10.1016/j.iatssr.2018.05.003`.

[31] Carlo Manna and Steve Prestwich. “Online Stochastic Planning for Taxi and Ridesharing”. In: *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*. Limassol, Cyprus: IEEE, 2014. DOI: `10.1109/ICTAI.2014.138`.

[32] Ulrich Meyer and Peter Sanders. “Δ-Stepping: A Parallelizable Shortest Path Algorithm”. In: *Journal of Algorithms*. 1998 European Symposium on Algorithms 49.1 (2003). DOI: `10.1016/S0196-6774(03)00076-2`.

[33] Motahare Mounesan et al. “Fleet Management for Ride-Pooling with Meeting Points at Scale: A Case Study in the Five Boroughs of New York City”. In: (2021). DOI: `10.48550/arXiv.2105.00994`.

[34] Masayo Ota et al. “A Scalable Approach for Data-Driven Taxi Ride-Sharing Simulation”. In: *2015 IEEE International Conference on Big Data (Big Data)*. Santa Clara, CA, USA: IEEE, 2015. DOI: `10.1109/BigData.2015.7363837`.

[35] Stefan Ropke, Jean-François Cordeau, and Gilbert Laporte. “Models and Branch-And-Cut Algorithms for Pickup and Delivery Problems with Time Windows”. In: *Networks* 49.4 (2007). DOI: `10.1002/net.20177`.

[36] Tobias Rupp. “Contraction Hierarchies: Theory and Applications”. PhD thesis. 2022. DOI: `10.18419/opus-12828`.

[37] Adella Santos et al. *Summary of Travel Trends: 2009 National Household Travel Survey*. FHWA-PL-11-022. 2011.

[38] David Schrank et al. *TTI’s 2010 Urban Mobility Report*. Texas Transportation Institute, 2010.

[39] Changle Song et al. “Incentives for Ridesharing: A Case Study of Welfare and Traffic Congestion”. In: *Journal of Advanced Transportation* 2021 (2021). DOI: `10.1155/2021/6627660`.

[40] Mitja Stiglic et al. "The Benefits of Meeting Points in Ride-Sharing Systems". In: *Transportation Research Part B: Methodological* 82 (2015). DOI: 10.1016/j.trb.2015.07.025.

[41] *The Multi-Agent Transport Simulation MATSim*. Ubiquity Press, 2016. DOI: 10.5334/baw.

[42] X. Wang, M. Dessouky, and F. Ordonez. "A Pickup and Delivery Problem for Ridesharing Considering Congestion". In: *Transportation Letters* 8.5 (2016). DOI: 10.1179/1942787515Y.0000000023.

[43] Yongjie Wang and Maolin Li. "Optimization Algorithm Design for the Taxi-Sharing Problem and Application". In: *Mathematical Problems in Engineering* 2021 (2021). DOI: 10.1155/2021/5572200.

[44] K. I. Wong and Michael G. H. Bell. "Solution of the Dial-A-Ride Problem with Multi-Dimensional Capacity Constraints". In: *International Transactions in Operational Research* 13.3 (2006). DOI: 10.1111/j.1475-3995.2006.00544.x.

[45] Shouqiang Xue et al. "Passenger-Perception Dynamic Ridesharing Service Based on Parallel Technology". In: *IET Intelligent Transport Systems* 17.9 (2023). DOI: 10.1049/itr2.12376.

[46] Biying Yu et al. "Environmental Benefits from Ridesharing: A Case of Beijing". In: *Applied Energy* 191 (2017). DOI: 10.1016/j.apenergy.2017.01.052.

[47] Lilhao Zhang et al. "A Parallel Simulated Annealing Enhancement of the Optimal-Matching Heuristic for Ridesharing". In: *2019 IEEE International Conference on Data Mining (ICDM)*. Beijing, China: IEEE, 2019. DOI: 10.1109/ICDM.2019.00101.

[48] Dominik Ziemke, Ihab Kaddoura, and Kai Nagel. "The MATSim Open Berlin Scenario: A multimodal Agent-Based Transport Simulation Scenario Based on Synthetic Demand Modeling and Open Data". In: *Procedia Computer Science* 151 (2019). DOI: 10.1016/j.procs.2019.04.120.