August 2023

Parallelization of Dial-a-Ride Using Tabu Search

Raghuveer Naraharisetti
University of Wisconsin-Milwaukee

PARALLELIZATION OF DIAL-A-RIDE USING TABU SEARCH

by

Raghuveer Naraharisetti

A Thesis Submitted in
Partial Fulfilment of the
Requirements for the Degree of

Masters of Science
in Computer Science

at
The University of Wisconsin-Milwaukee
August 2023

ABSTRACT

PARALLELIZATION OF DIAL-A-RIDE USING TABU SEARCH

by

Raghuveer Naraharisetti

The University of Wisconsin-Milwaukee, 2023
Under the Supervision of Professor Tian Zhao

Dial-A-Ride is a transport system heavily constrained by following fleet size, vehicle capacity, and a fixed number of requests (pickup and drop-off points) with time windows. It is often modelled as Integer Programming, various solutions are proposed using heuristics. One such heuristic is "Tabu Search". Tabu Search is very CPU intensive with its process of search, therefore many modern computing techniques like using GPUs have been employed to make it efficient.

As with many other greedy algorithms, the local optima is not always the same as the global optima, so it is not possible to go past the local optima using greedy techniques for this problem. It is often modelled as Integer Programming, with the search space being very big, there are proven to not be so efficient. So, many heuristics have been proposed in the past, one such heuristic is "Tabu Search". The local search of this heuristic uses memory to keep track of recent moves made and tries to avoid them for specified iterations (marks as Tabu) and also continues to explore the neighbourhood search space even with the degradation optimization function value, thus helping the algorithm to go past the local optima towards global optima.

This thesis focuses on limitations of parallelizing DARP-TS for multi-core CPU, discussing major factors like (i) number of good moves in the neighbourhood and how we can estimate a value for N_SIZE (number of parallel moves to make in each iteration), (ii) difference between a CPU core and a GPU core in the context of thread scheduling, memory layout and memory limitations, (iii) proposes few data-structures to keep the memory allocations low thus reducing the time for garbage collection and (iv) proposes an

incremental way of calculating the value of optimization function in the local search phase which helps in mapping the execution and evaluation of N_SIZE moves in each iteration onto the multiple CPU cores.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Dial-A-Ride Problem (DARP) deals with moving passengers from one location to another with a fixed set of vehicles with a fixed capacity constraint on each vehicle. The customers/passengers submit the requests with pickup & drop-off locations along with pickup & drop-off time windows and max ride time. The objective of DARP is to provide transportation service to these requests under constraints defined in the system while optimization for the objective function of the system. A DAR where all the requests and constraints of the DAR are known in advance is classified as Static DAR. If the requests are allowed to arrive while the service is active, such DAR is classified as Dynamic DAR. This paper deals with Static DAR, so all requests and constraints of the system are known in advance. DARP is proven to be NP-Hard, it is generalized from the Pickup and Delivery Problem with Time Windows (PDPTW)[1]. The optimization function for the DAR is flexible in the sense that it can be defined as a combination of various factors and also possible to optimize for a subset or a specific case by modifying the constant coefficients for each or subset of factors. This paper closely follows the optimization function defined here[2], which calculates the total violations of the system with the constraints defined in the system. The lesser the optimization values, the less violation and the better the solution.

The literature contains numerous solutions proposed for addressing the problem, in-

cluding both exact solutions and heuristic-based approaches. Although exact solutions guarantee optimality, their runtimes often become impractical for real-time problems as the scale increases [3]. On the other hand, heuristics offer reasonable runtimes for real-world scenarios[1]. Continuous efforts have been made to enhance these heuristic methods [1][3][4]. In light of modern hardware advancements, such as GPUs, [5] presented an adapted version of heuristic-based algorithms tailored for GPU utilization. While GPUs offer faster computation, their architecture primarily targets graphics processing, posing challenges when attempting to apply general-purpose computing algorithms to them. Additionally, GPUs tend to be costly and not readily available off the shelf. In contrast, CPUs have become increasingly powerful, affordable, and commonly found in everyday computing devices. With the advancements in silicon chip research and the latest ARM processors, it has now become reasonable and feasible to implement these heuristics on multi-core CPUs, which are highly accessible and powerful.

This paper makes some observations on the calculation of optimization function ($f(s)$) value in the local search phase and formulates a way to incrementally calculate optimization function values for the solutions generated by applying a move on the existing solution using dynamic programming techniques to avoid re-calculation of parts of $f(s)$. This paper also proposes a few data structures to support the above technique for calculating $f(s')$. This paper also proposes the use of probability on vertices when picking for the random move generation in the local search phase with the probability value directly proportional to the absolute value of violation at the respective vertex, this helps the algorithm to make moves such that vertices with high violation have a high probability of removal from the respective route. Finally, we present the results of applying the proposed algorithm to benchmark datasets from [2] and demonstrate its speedup compared to different thread configurations, with the single-thread configuration as the baseline.

The previous research on the Dial-a-Ride Problem with Tabu Search (DARP-TS) has primarily focused on improving the heuristics and exploring GPU-based optimizations using frameworks like CUDA. These approaches have shown impressive speedups but are

limited by the availability and cost of GPUs.

In contrast, there is a lack of research specifically targeting general-purpose multi-core CPUs for modelling the DARP-TS algorithm. This paper aims to fill that gap by leveraging the widespread availability and relatively low cost of multi-core CPUs. By designing the algorithm to harness the computational power of multi-core CPUs, the paper seeks to make the DARP-TS approach more accessible and applicable in practical scenarios.

Furthermore, the paper takes a different approach by focusing on computer science techniques, such as the use of better data structures, to minimize the re-computation of intermediate results. By optimizing the data structures and employing incremental calculation techniques, the paper aims to improve the efficiency of the algorithm without relying solely on heuristic enhancements.

Overall, this paper attempts to address the limitations of previous research by targeting multi-core CPUs, which are more readily available and cost-effective, and by exploring computer science techniques to optimize the algorithm's performance.

The paper focuses on the calculation of the optimization function ($f(s)$) value in the local search phase of the algorithm and introduces an incremental approach to calculate the violation function values for solutions generated by applying moves to the existing solution. This approach utilizes dynamic programming techniques to avoid redundant calculations of parts of the $f(s)$. To support the incremental calculation of $f(s')$, the paper proposes the use of specific data structures. These structures facilitate efficient updates and retrieval of information related to the parts of the $f(s)$.

Additionally, the paper suggests incorporating probability values when generating random moves in the local search phase. The probability assigned to each vertex is proportional to the absolute violation at that vertex. This probabilistic approach allows the algorithm to prioritize moves that remove vertices with higher violations from their respective routes.

The proposed algorithm is evaluated using benchmark datasets from a previous study. The results demonstrate the speedup achieved by different thread configurations compared

to the baseline single-thread configuration.

Overall, the paper presents observations and techniques to improve the efficiency of the algorithm, specifically in the local search phase. The proposed approaches aim to reduce redundant calculations, prioritize moves based on violation levels, and leverage parallel computing to enhance the algorithm's performance.

The mentioned algorithm is implemented in Julia language here

Chapter 2

Related work

2.1 Related Work

The DARP was first introduced by [6] with applications from various cities are reported [3], [7], [8]. Early literature focused on exact methods for single vehicles [9]–[11] and then form multiple vehicles [12], [13]. With exact solutions taking longer times to complete research focused more on applications of various Meta-heuristics to Dial-a-Ride Problem (DARP) with improved computational efficiency compared to exact methods. Several meta-heuristics, such as Genetic Algorithm [14], Variable Neighborhood Search [15], Deterministic Annealing and Multi-Atomic Annealing have been employed to address the DARP. These approaches have shown improved computational efficiency compared to exact methods, making them attractive alternatives. The focus has shifted towards developing effective solutions for large-scale DARP instances.

Tabu Search (TS) is one of the meta-heuristics [16] that has proven to be efficient in many optimization problems. Cordeau and Colorni [2] adapted Tabu Search for DARP and also introduced benchmark dataset instances and their best-known solutions with absolute runtime. Kirchler and Calvo [1] proposed a better and adapted version of DARP-TS with the objective of considering the moves with a constraint of granularity threshold calling it Granular Tabu Search (GTS) and also reported absolute runtime for the benchmark instances from [2].

Kirchler and Calvo [1] developed a new way of generating an initial solution where the DARP instance is modelled into a graph and an assignment problem is employed on the graph to get the assignment of requests to vehicles which are called clusters, where a cluster is defined as a set of requests that are close to each other spatially and temporally. Recent research [5] proposed a generic GPU framework for DARP and reported speedups for DARP-TS with various other heuristics modelled to run GPU devices.

However, the existing research lacks a comprehensive analysis of how the number of moves made in each iteration (N_SIZE) impacts the convergence speed to the best-known solution, as well as its performance with different CPU thread/core configurations. This paper aims to address this gap by presenting an adapted version of the DARP-TS algorithm. We investigate the relationship between N_SIZE, thread configurations, and the runtime to reach the optimal solution. By conducting experiments and analyzing the results, the paper provides insights into the optimal settings for N_SIZE and thread/core configurations, contributing to a better understanding of the algorithm's performance in different computing environments.

2.2 Tabu Search

DARP can be classified as an Integer Programming and combinatorial optimization problem. The best value for the optimization function in the system is referred to as Global Optima and the best value for the optimization function in each iteration or each move is referred to as Local Optima. In greedy solutions, local optima often reach the global optima. This is not the case with DARP, it is not always possible to escape the local optima with a greedy approach of considering a local move if and only if it improves the optimization function. In DARP, local optima is not equal to global optima. So, local optima have to be escaped to reach global optima, which means the algorithm has to allow the local optima to degrade and consider moves which degrade the optimization function, this will let us escape the local optima and eventually reach the global optima. Because of this reason, many meta-heuristic solutions have been proposed such as tabu search[16],

variable neighbourhood search [15], and genetic algorithms [14]. Meta-Heuristic solutions don't guarantee global optima but often guarantee near-global optimal solutions in a reasonable time, which is often desired in most real-world scenarios. This is why the usage of meta-heuristics has seen a significant rise in optimization problems.

Approximate algorithms in the domain of combinatorial optimization problems are often called heuristics. Meta-heuristics are a class of approximate methods, that are designed to attack hard combinatorial optimization problems where classical heuristics have failed to be effective and efficient. [17] define meta-heuristics as

Definition: A meta-heuristic is an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space using learning strategies to structure information in order to find efficiently near-optimal solutions.

So it doesn't guarantee a globally optimal solution is found but provides a solution that is near the global optimal solution that is good enough for many use cases. Tabu Search is one such meta-heuristic which is widely used in solving combinatorial optimization problems where the search space of the solution is exponential. A local search in tabu search is defined a as guided search for a solution that is in the neighbourhood of the current solution. A neighbourhood is defined as a set of solutions that can be obtained from the current solution by applying or making a 'MOVE' to the current solution. It is important to determine the next neighbourhood to explore at each iteration, with problems like DARP, where there exponential number of possible solutions from any given current solution, it is not possible or viable to explore all the possible moves, so we should achieve an effective trade-off between quality of next move and effort to find it. So we need some procedures to evaluate the quality of the next move. Using memory for a guided search of search space to avoid cyclic moves is introduced by [16]

The notion of things that are considered "taboo" can change over time, as memory evolves and adapts. Two kinds of memories are often used in Tabu Search, one is called 'Short-term memory' and the other is called 'Long-term memory', both play the role of

making the search process escape local minima while making sure that the search process is not just confined to a neighbour but explores more diverse regions [16]. Tabu Search (TS) is based on the idea that making poor strategic choices can provide more useful information than making random ones. TS incorporates intelligent search principles by balancing exploiting good solution features and exploring new, promising regions. Two strategies, Intensification and Diversification, are employed to make the exploration process more effective. Intensification modifies choice rules to focus on examining neighbours of elite solutions, while Diversification encourages the search process to examine unvisited regions and generate solutions that differ significantly from those seen before. The Tabu Search focuses more on the local search phase. It starts by generating an initial solution and performs a search that is referred to as 'Local Search' by making moves that direct the system towards better optimization function value or exploring new regions. [16]

Chapter 3

Formal Definition

3.1 Formal definition of DARP

This paper follows the exact formulation of DARP defined in [2]. The DARP is defined on a complete graph set G=(V,A), where $V = \{v0, v1, ...v_{2n}\}$ is the vertex set and $A = \{(v_i, v_j) : v_i, v_j \in V, i \neq j\}$. Vertex v_0 represents the depot for the fleet of nV vehicles. All the vehicles start and end the trip at v_0. The remaining $2n$ vertices represent pickup and drop-off for the transportation requests. Each vertex pair (v_i, v_{i+2n}) represents the pickup and drop-off vertex for the transportation request i. Each request comes with a set of constraints with the same or different values for pickup and drop-off vertices. Below are these constraints

- Load: Each vertex $v_i \in V$ has a associated load denoted by q_i for request i. $q0 = 0$
- Time Window: Each vertex $v_i \in V$ has a time window $[e_i, l_i]$ for the requested service.
- Service Duration: Each vertex $v_i \in V$ has a non-negative service duration, which is the time duration reserved for service at this vertex

Let s be the solution for a given DARP instance, then is defined as a set of nV routes, one for each vehicle in the fleet $s = \{route_i : i \in N \,|\, 1 \leq x \leq nV\}$. Solving for s consists

of following

- Every route start and end at v_0
- Each route is defined as a set of arc (v_i, v_j) where i and j are consecutive vertices in the same route.
- For each request i, the requests v_i and v_{i+2n} belong to the same route and v_{i+2n} is visited later than vertex v_i
- The load of the vehicle k at any point in time doesn't exceed Q_k
- The service at vertex v_i begins in the interval $[e_i, l_i]$ and every vehicle leaves the depot and returns to the depot in the interval $[e_0, l_0]$
- The ride time of any user does not exceed L
- The total routing cost for all vehicles is minimized

Other variables are also associated with each vertex v_i

- Arrival Time - A_i, represents time of arrival of vehicle at vertex v_i
- Beginning of Service - B_i, represents time at which service is started at vertex v_i. $B_i = max(A_i, e_i)$
- Departure Time - D_i, represents time at which vehicle left the vertex v_i. $D_i = B_i + d_i$
- Wait Time - W_i, represents wait time incurred at vertex v_i. $W_i = B_i - A_i$
- Ride Time - L_i, represents total ride time of the passenger boarded at vertex v_i. $L_i = B_{i+n} - A_i$

3.2 Optimization function

The Optimization function is denoted by $f(s)$ and gives a quantified value of how optimized is given solution s for the given DARP system and its constraints. It can be seen as the

summation of total violations of the DARP constraints, so the lower the value of $f(s)$, the lesser the violations by s and the better the solution. So, the algorithm tries to get the optimization to a lower value. The $f(s)$ consists of 5 parts

- $c(s)$ - denotes total travel costs of all the routes
- $q(s)$ - denotes total load constraint violations on all the routes
- $d(s)$ - denotes total max route duration violations on all routes
- $w(s)$ - denotes total time window violations on all routes
- $t(s)$ - denotes total ride time violations on all routes

$$s = \{route_i : i \in N \mid 1 \leq x \leq nV\} \tag{3.1}$$

$$f(s) = c(s) + \alpha q(s) + \beta d(s) + \gamma w(s) + \Gamma t(s) \tag{3.2}$$

Here $\alpha, \beta, \gamma, \Gamma$ are self-adjusting positive parameters, by dynamically adjusting these parameters the algorithm directs the solution to explore solution space and is particularly useful for tightly constrained instances [2]

Chapter 4

CPU core vs GPU core

A GPU, which stands for Graphics Processing Unit, is a specialized processing unit primarily designed for rendering images and videos. It takes input data related to images, videos, or scenes and generates output that can be displayed on screens. GPU processing involves intensive vector math and matrix multiplications. GPUs are optimized for these operations with specific design choices, but they may have limitations when performing general computing tasks. Typically, GPUs have thousands of cores.

On the other hand, CPUs, or Central Processing Units, are designed for general-purpose computing. CPUs take instructions as input and produce output. The design choices in CPUs are geared towards a wide range of computing tasks. Modern CPUs usually have 8 to 32 cores. However, comparing GPUs and CPUs based solely on the number of cores would not be a fair comparison. For example, the latest NVIDIA Rx4090 GPU has 16,384 cores and offers 49 TFLOPS, while the latest Intel i9 CPU has 24 cores and provides 849 GFLOPS, which is significantly lower compared to the Rx4090. However, this does not mean that the Rx4090 GPU will outperform the i9 CPU in all scenarios. The design choices made around GPUs optimize them for graphics processing, and it would not be advisable to replace GPUs with CPUs in all cases.

To understand why a GPU core is different from a CPU core, we need to consider their respective designs and characteristics. GPU cores are designed for high throughput,

high bandwidth, and high-precision floating-point mathematical operations. They excel at fast mathematical operations but may not perform as well in general-purpose computing. At a high level, a GPU core consists of four components: a dispatch unit for receiving instructions, a Floating Point Unit (FPU) for floating-point operations, an Integer Unit (INTU) for integer operations, and a results queue for storing the results of instructions. When a GPU receives an instruction, it is handled by the scheduler, which assigns the instruction to a specific thread. Threads in a GPU are organized into WARPs, which are sets of 32 threads. WARPs work with a concept called "Single Instruction Multiple Data" (SIMD), meaning all threads in a WARP can only execute a single instruction per cycle but on different data if necessary. However, this comes with a limitation: if there is branch divergence in one or some threads, it blocks the progress of all threads within the WARP since the instruction is no longer uniform. Therefore, when designing programs for GPUs, it is important to minimize or avoid branch divergence to ensure optimal performance.

CPU cores are designed for general-purpose computing and are optimized to handle unpredictable branching, data access, and arbitrary instructions. Each core in a CPU can operate independently, which is known as MIMD (Multiple Instruction Multiple Data) architecture. This independence of cores makes CPUs more suitable for general-purpose computing tasks. Additionally, modern compilers have become intelligent enough to optimize programs for various CPU architectures. This implies that a single program can be targeted towards a wide range of machine architectures, eliminating the need for device-specific optimizations. This alleviates the requirement for modifying programs with each new iteration of device architecture.

Chapter 5

More Tabu Search

A solution in DARP is a set of routes, usually one for each vehicle in the fleet. In Tabu Search, the initial solution plays a crucial role in guiding the local search towards near-optimal solutions. A good initial solution can expedite the convergence to better solutions compared to a poor initial solution. [1], the authors propose using an assignment problem to generate an initial solution for the Dynamic Dial-a-Ride Problem (DARP).

To model DARP as an assignment problem, the goal is to construct clusters of requests that are spatially and temporally close together. Each cluster represents a vehicle route. Since each request has pickup coordinates and a pickup time window, the system can be represented as a graph with nodes in a two-dimensional space. However, considering only one dimension (e.g., pickup coordinates) would not be accurate since requests can have different or widely separated time windows. To address this, the paper introduces the concept of "Average Departure Time" to convert the two-dimensional values into a scalar value.

In the constructed graph, all requests and vehicles are considered nodes, and edges are assigned weights based on the following rules: 1. If both nodes A and B are requesting nodes, the weight of the edge connecting them is the Average Departure Time of A and B. 2. If A and B are the same requesting node, no edge is connected (no self-edges). 3. If A is a requesting node and B is a vehicle node (or vice versa), the weight of the edge

connecting them is the time taken for vehicle B to reach the pickup node of request A from the starting depot. 4. If both A and B are vehicle depots, they are not connected by an edge.

Once the nodes are connected and weights are assigned, the graph-building step is complete. The assignment problem algorithm is then applied to this graph, aiming to minimize the overall cost of the assignment. The output of this algorithm is a cluster of nodes representing the initial solution, which is subsequently used as the starting point for the Tabu Search.

[5] proposes an alternative approach to generating the initial solution by randomly generating multiple solutions and selecting the one with the lowest optimization function value. In this paper, the method defined in [5] is employed to generate the initial solution.

5.1 Issues with Assignment Problem

The worst-case run-time complexity of the assignment problem is $O(n^3)$ and it is not trivial to parallelize this to make sure multi-CPU, so we end up only using a single CPU even though we have multi-CPU. Another issue we found with the assignment problem is that, if the requests are close enough, the solution of the assignment problem often ends up with a single or few big cluster or clusters. This is not considered a good initial solution and would make the local search difficult to reach the near-optimal value of the optimization function, as the optimal solution divides the requests into multiple clusters so that each cluster can be assigned to a vehicle. For the above reason, we found that using the assignment problem is not viable when considering modelling the DARP to run on high parallel architectures, So we need to use other approaches for generating the initial solution.

5.2 Local Search

Once the initial solution is generated, the next step in TS is the search process, which is usually called Local Search. The local search has the following components to it (Figure 5.1).

- Current solution, which is the current solution of the system.
- Best solution, which is the best solution found so far
- A move, a move consists of 5 parts i, k1, k2, p1, p2, where request i from route k1 and inserted to the route k2, where pickup node for request i is inserted at position/index in the route k2 and drop-off node for request i is inserted at position/index in the route k2

Local search is performed in iterations. Each iteration requires an evaluation of the neighbourhood and executing exactly one move to generate a new solution. In each iteration, the 'N_SIZE' number of moves are randomly generated and each is applied to the current solution, each yielding a new solution. Then the best solution which gives the lowest value for the optimization function is picked among the new generated solution and made as the current solution for the next iteration, if the current solution is better than the best solution found so far, the best solution is updated with the current solution. This continues until a Stopping Criteria has been met. This paper follows two stopping criteria.

- Run until we reach the best-known solution
- Run for a given period of time

5.3 Diversification and Intensification

Diversification and Intensification are employed to improve the effectiveness of the Tabu Search method. Intensification directs the search towards more known elite regions which have a high probability to be part of an optimal solution. Diversification directs the search

towards more unexplored regions, this helps the search to escape the local minima and reach optimal solutions. [2]

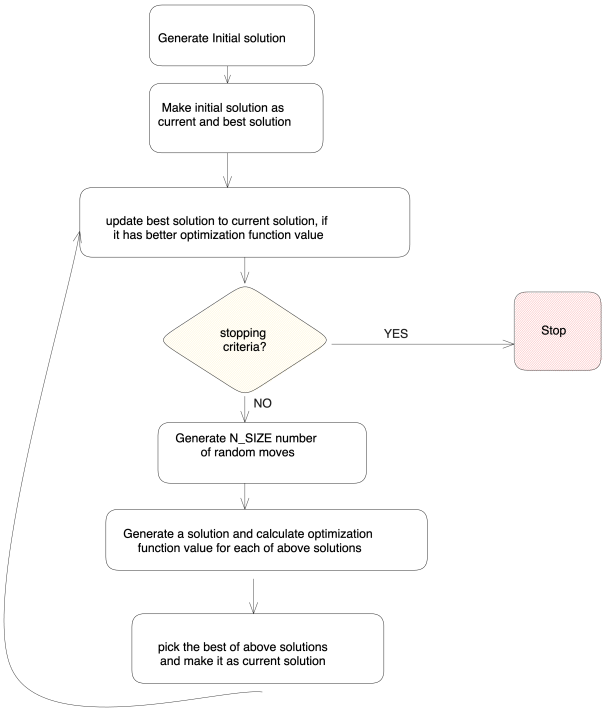

Figure 5.1: Tabu Search flow diagram

5.4 Tabu Memories

To escape the local minima and explore new promising regions, TS uses two types of memories.

- Short-Term Memory, also called 'Tabu list'
- Long-Term Memory

In DARP-TS, we store the arcs of moves used in both the memories, but for various reasons. An arc can be defined as a tuple $(move.i, move.k_2)$ referred as p_{ik}, which represents the request i being inserted or assigned to the vehicle of route k_2.

Short-term memory is used to mark the moves used as tabu for a certain number of iterations, this is used for intensification, and diversification of the search process (See

section 4). The value of the number of iterations to tabu a certain move depends on the value of θ at the time of operation. θ is defined in [2] as a variable to hold the value of $TabuTenure$, the number of iterations for which a move is forbidden from being reconsidered again.

Long-term memory is used to record the number of times a particular p_{ik} is considered in any of the current solutions in the search process. This is used to calculate the p_{ik} in the penalty formula, which is used to penalize a solution by a factor proportional to the frequency of addition of its distinguished attributes [2].

Aspiration Criteria

In Tabu Search, We need to override tabu rules in certain cases. These are often defined in terms of aspiration criteria. These allow the algorithm to reconsider moves that are marked tabu for certain iterations. [16]

Intra route Optimizations

[2] proposes executing intra-route optimizations in the local search algorithm after a certain number of iterations, denoted as κ. The optimization process involves sequentially removing each vertex (v_1, v_2, ..., v_{2n}) from its original route and reinserting it at a new position within the same route. The objective is to find the position that maximizes the value of the optimization function. This approach employs a brute force method, considering all possible combinations and selecting the best one for each route.

Chapter 6

DARP-TS on multi-core CPU

In the DARP-TS (Dial-a-Ride Problem with Tabu Search) context, the local search process is identified as the most time-consuming aspect [5]. Thus, it is sensible to focus on parallel execution for the local search to leverage the capabilities of multi-core CPUs. During each iteration, N_SIZE random moves are generated, and each move is executed to create a new solution. The $f(s)$ is then recalculated for each solution s', and the best one is chosen as the current solution for the next iteration. This process continues until a stopping criterion is met. It is important to note that recalculating the $f(s')$ is a computationally intensive task and this becomes much more impact on the performance if we increase the N_SIZE i.e. explore more of the neighbourhood in each iteration. Thus, it is important that we optimize the calculation of $f(s')$, where s' is the result of applying a move on s

To analyze the runtime complexity of calculating the $f(s')$ for a given solution s' in DARP which is the result of applying a move on s, let's consider an example where N_SIZE is the number of solutions generated per iteration, nR represents the number of requests, and nV denotes the number of vehicles. The runtime complexity of calculating the $f(s')$ value for a single route is O(nR) since the worst case could involve all requests being assigned to a single route. Consequently, the runtime complexity for all the routes in a solution (i.e., a single iteration) is O(nR * nV). If we extend this calculation to N_SIZE solutions per iteration, the runtime complexity becomes O(N_SIZE * nR * nV)

per iteration.

This paper makes a few observations from the above text to make some improvements to the runtime complexity defined above.

6.1 Observation 1: Summation of per route parts of each route, gives us f(s)

The Optimization function $f(s)$ used in the paper is defined in $eq2$. It is the summation of parts i.e. violations. Each part is the sum of violations on all the routes.

$$c(s) = \sum_{j=1}^{nV} c_r(route_j)$$

$$q(s) = \sum_{j=1}^{nV} q_r(route_j)$$

$$d(s) = \sum_{j=1}^{nV} d_r(route_j)$$

$$w(s) = \sum_{j=1}^{nV} w_r(route_j)$$

$$t(s) = \sum_{j=1}^{nV} t_r(route_j)$$

the expanded version of $f(s)$ can be written as

$$f(s) = \sum_{j=1}^{nV} (c_r(route_j) + \alpha q_r(route_j) + \beta d_r(route_j) + \gamma w_r(route_j) + \Gamma t_r(route_j)) \quad (6.1)$$

6.2 Observation 2: Not all requests need to be re-calculated in a local search iteration

In the local search, a randomly generated move is applied to the then current solution. A move is defined as picking a request i from route k_1 (both pickup and drop-off nodes) and inserting them in route k_2 at indices p_1, p_2. If we observe, not all the routes are impacted when applying a move [reference 1]. If a solution has route $k_1, k_2, k_3, k_4....k_{nV}$ and a move $(r, k_3, k_5, p_1', p_2')$ is applied which resulted in a solution $k_1, k_2, k_3', k_4, k_5', ...k_{nV}$, here only routes k_3, k_5 are modified to k_3', k_5'. So, we can calculate the $f(s_{new})$ value for the new solution obtained by using the $f(s_{cur})$ values of the current solution where the routes are included in the move and recomputing for the routes that are included in the move. This can be seen as an optimal substructure and able to sub-problem overlapping. Let s' be solution obtained by applying *move* on existing solution s, for which $c(route_j), q(route_j), d(route_j), w(route_j) \forall (j \in s)$ are already calculated when calculating $f(s)$

$$move = (i, k_1, k_2, p_1, p_2)$$

$$s' = \{x \in s \,|\, 1 \leq x \leq nV,\, x \neq k_1,\, x \neq k_2\} \cup \{k_1, k_2\}$$

$$c(s') = (\sum_{\substack{j=1 \\ j \notin (k1,k2)}}^{nV} c_r(route_j)) + c_r(route_{k1}) + c_r(route_{k2})$$

$$q(s') = (\sum_{\substack{j=1 \\ j \notin (k1,k2)}}^{nV} q_r(route_j)) + q_r(route_{k1}) + q_r(route_{k2})$$

$$d(s') = (\sum_{\substack{j=1 \\ j \notin (k1,k2)}}^{nV} d_r(route_j)) + d_r(route_{k1}) + d_r(route_{k2})$$

$$w(s') = (\sum_{\substack{j=1 \\ j \notin (k1,k2)}}^{nV} w_r(route_j)) + w_r(route_{k1}) + w_r(route_{k2})$$

$$t(s') = (\sum_{\substack{j=1 \\ j \notin (k1,k2)}}^{nV} t_r(route_j)) + t_r(route_{k1}) + t_r(route_{k2})$$

$$f(s') = c(s') + \alpha q(s') + \beta d(s') + \gamma w(s') + \Gamma t(s')$$

The above equations formulate the idea that when we generate a solution s' by applying a move on s, we can reuse the c, q, d, w, t violation function values for the unchanged routes ($route_j$ in above equations) and just recompute the violation function values for the changed routes ($k1, k2$) in above equations). Since the number of routes that are modified in each new solution is 2 which is constant. The runtime complexity of making N_SIZE moves and picking the best move that yields the best solution now becomes O(N_SIZE * nV * 2) which equals O(N_SIZE * nV). Using observation 2, we could improve the runtime complexity of local search iteration from O(N_SIZE * nR * nV) to O(N_SIZE * nV)

6.3 Using probability in picking nodes from the route for generating moves

In the step of randomly generating N_SIZE number of moves, below are a sequence of steps involved in this single step

- Make a list of all requests in the DAR system
- Loop below steps for N_SIZE times
  - Pick a route k_1 from list of routes in the current solution
  - Pick a request i from the above picked route k_1
  - Remove both pickup and drop-off nodes associated with this request from route k_1

- Randomly pick a route k_2
- Randomly pick two points or indices p_1, p_2 in route $k2$
- Build structure (i, k_1, k_2, p_1, p_2)
- Check if (i, k_2) is tabu, if yes mark the move as tabu
- Add the generated move to the list of N_SIZE moves

Instead of randomly picking requests from route k_1, this paper sets the probability proportional to the value of violations at each node for all the nodes. This makes those nodes highly likely to be removed and re-inserted into other routes. Our experiments show that this technique will direct the solution to not include nodes in routes which can result in high violations thus improving the value of $f(s)$.

6.4 Diversification and Intensification strategies

This paper uses diversification and intensification strategies from [2] and "Dynamic variation of Tabu tenure" from [1].

6.5 Aspiration Criteria

This paper uses aspiration criteria defined in [1], which allows move arcs which are tabu to be executed in local search only if they lead to a solution whose objective function value is better than the best solution found so far. [1]

Chapter 7

Scaling

7.1 Scaling local search for multiple CPU cores & role of N_SIZE

Modern programming languages offer high-level semantics to map compute functions onto available threads or cores in a compute instance. In the context of DARP-TS, we have identified the local search as the component that needs to be scaled to utilize multiple cores effectively.

The size of the entire neighbourhood in DARP-TS is defined as $nR * nV$, indicating that each vehicle can potentially be assigned to the route of any other vehicle. The number of moves that can be made is also limited by the size of the neighbourhood. Therefore, the value of N_SIZE should be constrained by the size of the neighbourhood. However, setting N_SIZE equal to $nR * nV$ is not advisable. In TS, moves are tabu for certain iterations to avoid repetitive searches in the same neighbourhoods. If N_SIZE is set too high, the algorithm may get stuck in move generation, and although this can be mitigated by using aspiration criteria, it does not lead to effective solutions escaping local minima or reaching the optimal solution in a reasonable amount of time. On the other hand, if N_SIZE is set too low, the algorithm may not take sufficient steps to reach an optimal solution within a reasonable time frame.

The value of N_SIZE should also be directly proportional to and bounded by the number of available cores. Setting N_SIZE larger than the available cores will result in increased iteration time since there are fewer cores than required to execute and evaluate the N_SIZE moves concurrently. Conversely, setting N_SIZE smaller than the available cores leads to under-utilization of cores, and due to the lower number of moves executed per iteration, it takes more time to reach an optimal solution.

In conclusion, the value of N_SIZE in DARP-TS should be directly proportional to the size of the neighbourhood and the number of available cores, ensuring a balance between utilizing the neighbourhood effectively and making efficient use of the available computational resources. The later section of the paper discusses a formula to effectively pick a N_SIZE for a given nR, nV and a number of available cores.

7.2 Scaling N_SIZE

The value of N_SIZE in the DARP-TS algorithm is indeed bounded by the size of the neighbourhood. If N_SIZE is set to a value larger than the size of the neighbourhood, the algorithm may get stuck in the random move generation phase as it won't be able to generate N_SIZE unique moves. On the other hand, if N_SIZE is set too small, it may not effectively utilize the computing resources, resulting in a long time to reach the optimal solution.

To address this, the paper proposes setting the value of N_SIZE as a percentage of the dataset size. This approach ensures that the N_SIZE value is proportional to the size of the neighbourhood, allowing the algorithm to explore a sufficient number of moves while avoiding getting stuck in the random move generation process. By relating N_SIZE to the dataset size, the paper aims to strike a balance between exploration and computational efficiency in the local search phase of the algorithm.

Therefore, it's important to consider the characteristics of the problem instance and the specific neighbourhood being explored when setting the value of N_SIZE. Adapting N_SIZE based on the problem instance, such as a percentage of the dataset size, can provide

a reasonable starting point. However, fine-tuning and experimentation may be necessary to determine the optimal value of N_SIZE that strikes a balance between exploration and computational efficiency in the search process.

In summary, the value of N_SIZE should be carefully chosen, taking into account the number of "good moves" available in each neighbourhood, to ensure the effective exploration of the search space and efficient convergence towards the optimal solution.

Chapter 8

Implementation issues

8.1 Memory Scaling

The fact that all the cores in a CPU share common memory has both advantages and disadvantages when designing algorithms. One advantage is that there is no need to move data into specific parts accessible only to the target core or close to it; instead, all the memory can be shared (except for a few core-specific caches). However, a disadvantage is that the algorithm becomes highly susceptible to data races. It becomes easy for a thread or core to overwrite a shared memory block while another thread is using it to store intermediate values or variables. This can lead to unexpected and erroneous behaviour in the algorithm. Most of the high-level languages provide some kind of semantics to partially alleviate this issue.

In programming languages with automatic garbage collection, the program lacks control over the memory release process, which is handled by the language runtime. This means that the runtime can interrupt the program at any time for garbage collection, leading to potential slowdowns in DARP-TS if memory allocation is not carefully managed.

For instance, if we employ a naive approach, each iteration requires the creation of nV arrays, each of which has a size of at most nR. In the worst case, the space complexity becomes $O(N\_SIZE * nR * nV)$. Considering the scale of the neighbourhood, garbage

collection may kick in at some iteration, causing the program to lose control. It's important to note that garbage collection algorithms are typically single-threaded, which can adversely impact the runtimes of DARP-TS.

To address this issue, one solution is to always use pre-allocated data structures in the local search process. By doing so, no new allocations are made during each local search iteration, resulting in fewer garbage collection interventions and reduced garbage collection runtimes. The paper under discussion utilizes pre-allocated arrays for all local search processes, which has demonstrated improved optimization runtimes in experiments conducted with the Julia implementation.

By employing pre-allocated data structures, the DARP-TS algorithm can effectively mitigate the impact of garbage collection on its performance, leading to more efficient execution.

8.2 Data structures

The paper discusses various implementation-specific data structures used to optimize memory usage and avoid dynamic memory allocations. These data structures contribute to achieving the desired objectives mentioned in the "Memory Scaling" section 8.1. Four key data structures introduced and discussed in the paper are the Route, RMap, OptRoute and OptRoutes.

Route

A Route is defined as an array that represents the pickup or drop-off nodes for vehicles in the fleet. Routes are extensively used throughout the algorithm, from creating initial solutions to applying moves in local search iterations. To avoid dynamic memory allocations, the paper pre-allocates three instances of routes:

- Initial Solutions: Since the number of initial solutions is known in advance, arrays required for all initial solutions are pre-allocated.

- Current Solution routes in local search: As these routes change with each local search iteration, they are pre-allocated.
- Best Solution routes: These routes are also pre-allocated.

By pre-allocating these instances, the paper minimizes dynamic memory allocations and enables efficient memory utilization. In local search iterations, when N_SIZE number of moves are made, only two routes in a solution are modified, allowing the re-use of the other nV-2 routes' metadata structures without cloning. This approach enables scaling of N_SIZE on devices with limited available memory. Changes to current or best solution routes are performed in place without dynamic allocations.

RMap

The algorithm requires hash maps to store information such as Arrival Times, Departure Times, and Wait Times for all nodes in a route and all routes in the system. However, pre-allocating and re-using hash maps is challenging. To address this, the paper utilizes pre-allocated arrays for these purposes. To map a particular node to an index in the respective pre-allocated arrays, the paper introduces a single hash map referred to as RMap. RMap stores the index at which information about a specific node is stored in all the pre-allocated arrays.

By using pre-allocated arrays and the RMap data structure, the paper achieves efficient memory management and avoids the need for dynamic memory allocations for hash map-related operations.

Figure 8.1 provides a visual representation of the RMap data structure and Pre-allocated array for Wait Times of a route as an example. As per the mappings in RMap, the associated value for the pickup vertex of request 20 can be found in index 2 of the pre-allocated array for wait times which is value "4". So for each solution, we now only need one hash map which we refer to as RMap to map which index in the pre-allocated array has the information for a respective vertex. This way, we can have an arbitrary number of pre-allocated arrays to store various kinds of intermediate and final results for

a solution and reuse these pre-allocated arrays in the next iterations.

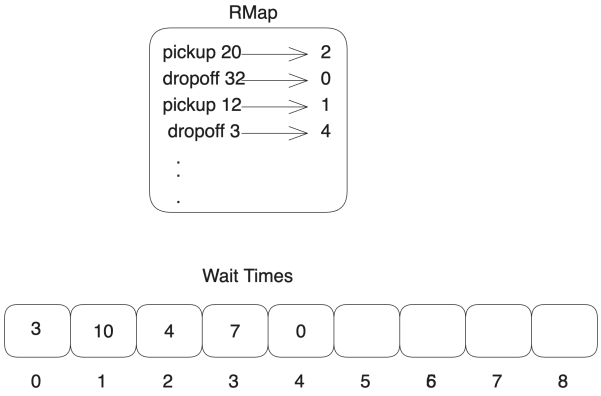

Figure 8.1: RMap data-structure

OptRoute

In the context of scaling local search for multiple CPU cores, storing violations per route allows for the incremental construction of the f(s) value for solutions generated by applying moves in the local search. OptRoute is a data structure that holds all the violations for a specific route. It also includes probability weights for each node in the route, which are used in the random generation of moves. Whenever possible, OptRoute is re-used for unchanged routes, reducing the need for additional memory allocations.

OptRoutes

OptRoutes is a data structure that holds instances of OptRoute for a given solution, which is a set of routes. It also stores various other metadata related to the solution, such as the f(s) value, cost penalty, and the move that resulted in the solution (if applicable). When constructing an OptRoutes instance for a new solution, components of OptRoutes are partially re-used, further optimizing memory usage.

By utilizing the OptRoute and OptRoutes data structures, the paper achieves efficient storage of violations, probability weights, and other metadata associated with routes and solutions. This helps in improving memory utilization and the overall performance of the algorithm.

Chapter 9

Performance Evaluation

The proposed Multi-CPU-based DARP-TS is implemented in Julia language. All the experiments were conducted on AMD Ryzen Threadripper PRO 3975WX 32-Core. We considered standard DARP benchmark instances from [2] to conduct the experiments. These instances have various datasets where the number of requests ranges from 24-144 and the fleet size from 3-13. The algorithm is run with different thread configurations by setting Julia's specific system environment variable $JULIA\_NUM\_THREADS$. Following guidelines of [2], these experiments use the same violation coefficient values or ranges.

Figs. 3 to 7 show the runtime of the algorithm to reach the best-known solution with various thread configurations. For N_SIZE for these experiments, we chose 10, 20, 30 and 35 % of the data-set size, where dataset size is defined by $nR * nV$.

Fig. 8 shows the convergence plot for the time taken to achieve the best-known solution for data-set pr10 with 1 and 16 thread configuration and N_SIZE values of 10,20, 30 and 35 % of the data-set size.

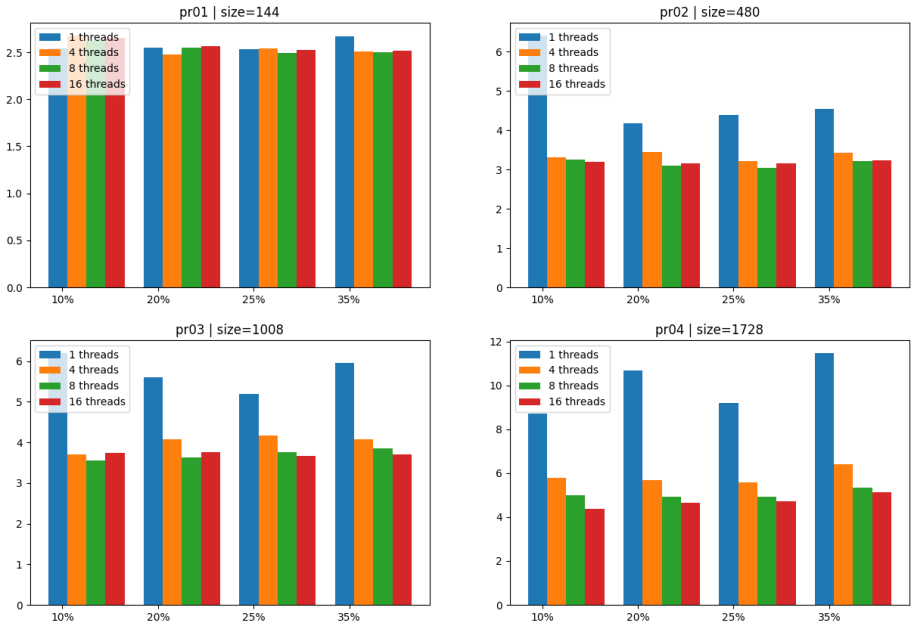

Figure 9.1: pr01-pr04

the impact of different N_SIZE percentages (10%, 20%, 25%, 35%) and thread configurations (1, 4, 8, 16 threads) on the algorithm's runtime for pr01, pr02, pr03 and pr04 datasets.

Based on the pr01 plot in 9.1, which represents the dataset pr01 with 24 requests, it can be concluded that there is not a significant improvement in runtimes when varying both N_SIZE and multi-CPU cores. This observation can be explained by the following factors:

- N_SIZE Value: The values of N_SIZE $20\%, 25\%, 35\%$ are too big for the dataset size which 144. The 10% N_SIZE is good enough to explore most of the "good moves" in the neighbourhood, increasing the N_SIZE beyond that is only exploring other bad moves that don't contribute to the optimization process. This also explains why the total runtime is very similar for all the percentages of N_SIZE.

- Multi-CPU Cores: Increasing the number of CPU cores may not lead to significant runtime improvements due to certain factors. One possible reason is that the computational power provided by a single thread configuration is already sufficient for the given problem size. Additional threads may introduce more context switching and thread overhead, which can negate any potential performance gains.

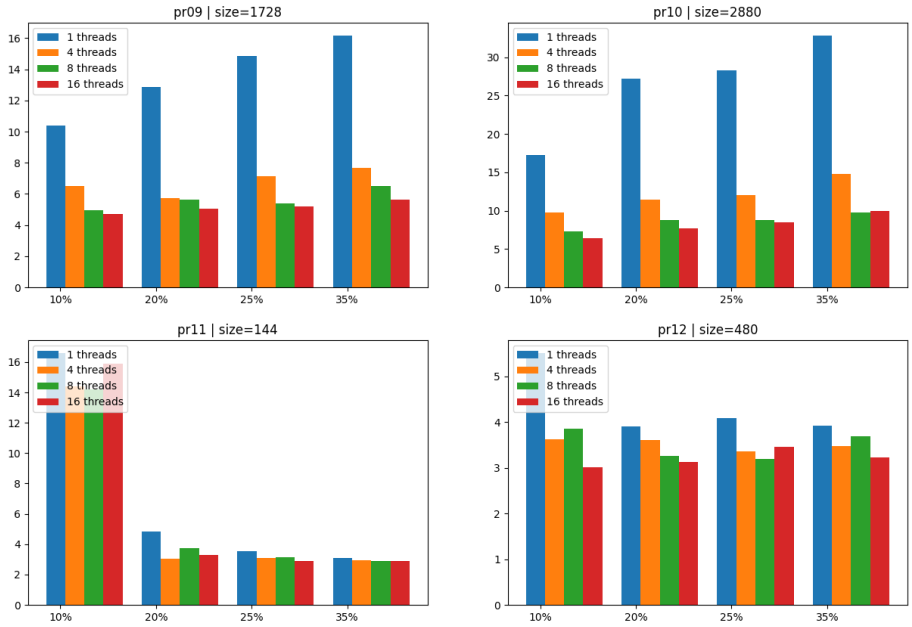

Figure 9.2: pr09-pr12

the impact of different N_SIZE percentages (10%, 20%, 25%, 35%) and thread configurations (1, 4, 8, 16 threads) on the algorithm's runtime for pr09, pr10, pr11 and pr12 datasets

Based on plots pr09 and pr10 in 9.2, which represent the datasets pr09 with 108 requests and 8 vehicles in the fleet, and pr10 with 144 requests and 10 vehicles in the fleet, the following observations can be made:

- Runtime Improvement with Number of Threads: The runtime of the algorithm demonstrates a significant improvement as the number of threads increases. This indicates that parallelizing the algorithm by utilizing multiple threads can effectively reduce the overall execution time. As more threads are employed, the algorithm completes faster. However, the difference in improvement between 8 and 16 threads is not as substantial, suggesting that there may be diminishing returns beyond a certain number of threads. Increasing the number of threads beyond a certain number may not provide significant additional runtime improvements.

- Limitations of N_SIZE: The observations indicate that there is not a significant improvement in runtime with an increase in N_SIZE. This is not due to the inability to scale the moves with available threads, but rather due to a limitation on the number of "good moves" that can be made in each iteration.

In Figure 9.6, the convergence to the best-known solution for different N_SIZE values in the pr10 dataset is plotted. The graph shows that for the 16-thread configuration, all N_SIZE values exhibit similar slopes. This can be attributed to the observation that the number of good moves within a neighbourhood is limited, and reaching the optimal solution entails navigating through these neighbourhoods. Despite parallelizing the execution and evaluating multiple neighbourhoods simultaneously, the algorithm is still constrained by the sequential aspect of navigating through neighbourhoods.

On the other hand, in the 1-thread configuration, the slope of the convergence line increases with higher N_SIZE values. This is because there is a limitation on the number of "good moves" available, and exceeding this limit results in making moves that do not contribute to the optimization process. Instead, it increases the computational time of each iteration, consequently slowing down the convergence process.

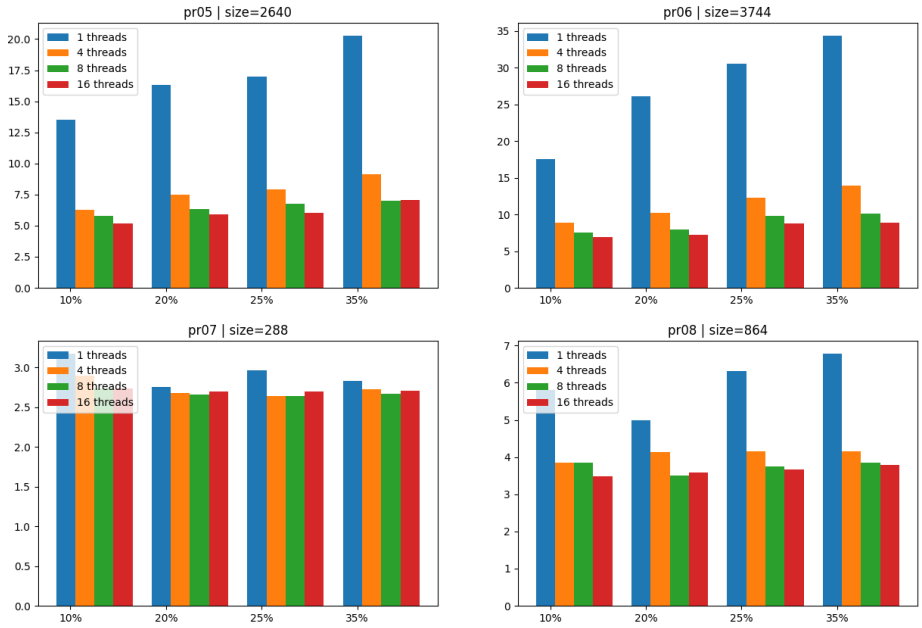

Figure 9.3: pr05-pr8

the impact of different N_SIZE percentages (10%, 20%, 25%, 35%) and thread configurations (1, 4, 8, 16 threads) on the algorithm's runtime for pr05, pr06, pr07, pr08 datasets

Overall, these observations highlight the impact of N_SIZE and thread configurations on the convergence speed of the algorithm, with the limitations of good moves and the interplay between parallel execution and sequential neighbourhood navigation influencing the results.

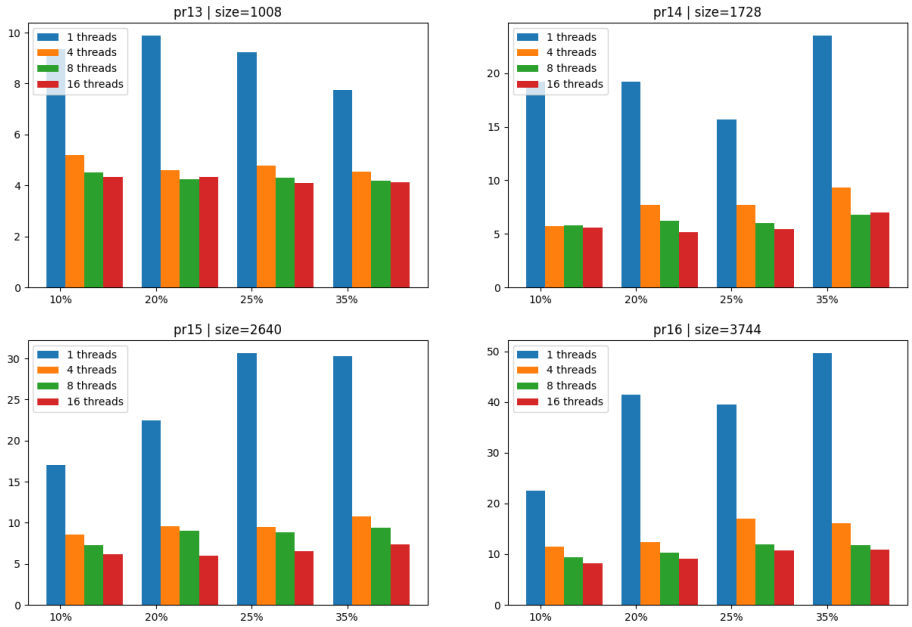

Figure 9.4: pr13-pr16

the impact of different N_SIZE percentages (10%, 20%, 25%, 35%) and thread configurations (1, 4, 8, 16 threads) on the algorithm's runtime for pr13, pr14, pr15, pr16 datasets

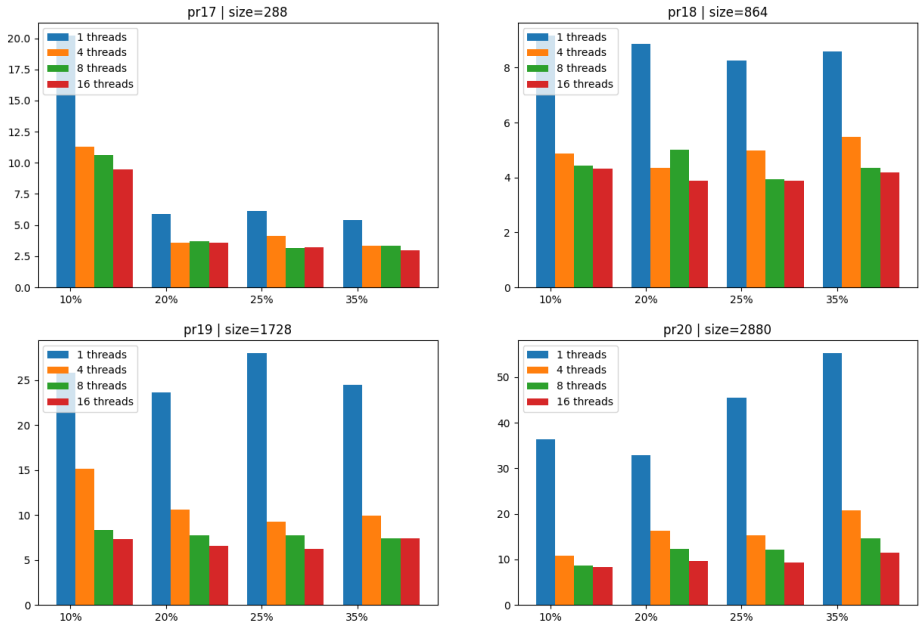

Figure 9.5: pr17-pr20

the impact of different N_SIZE percentages (10%, 20%, 25%, 35%) and thread configurations (1, 4, 8, 16 threads) on the algorithm's runtime for pr17, pr18, pr19, pr20 datasets

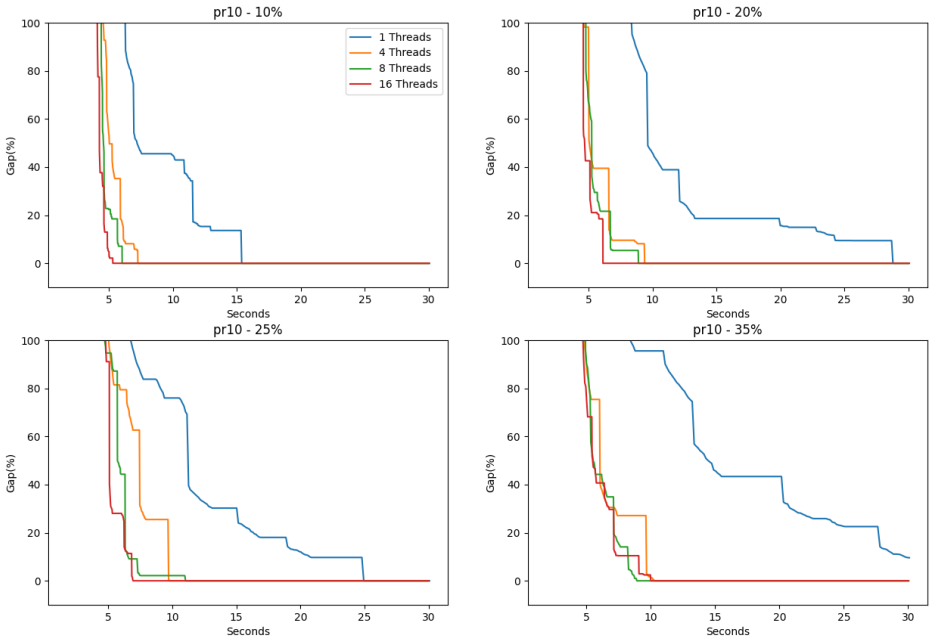

Figure 9.6: converging plot for pr10

the impact of different N_SIZE percentages (10%, 20%, 25%, 35%) and thread configurations (1, 4, 8, 16 threads) on the algorithm's convergence to the best-known solution.

Chapter 10

Conclusion

This research focuses on addressing key limitations of the DARP-TS algorithm when applied to multi-CPU cores. One major limiting factor is the number of "good moves" available within the neighbourhood, which directly affects the algorithm's ability to reach an optimal solution. To estimate this metric, the research proposes using a percentage of the dataset size, specifically the number of requests multiplied by the number of vehicles. This provides a rough approximation of the neighbourhood's size and helps in determining an appropriate value for N_SIZE.

Additionally, the study explores the impact of different thread configurations on the algorithm's performance. It observes a speedup of approximately three times when utilizing multiple threads compared to a single-threaded configuration. However, it is also noted that higher threads converge faster compared to lesser threads. This finding suggests that the algorithm's progress is limited by the serial nature of navigating through the various neighbourhoods, despite parallelizing other aspects of the execution such as execution and evaluation of N_SIZE neighbourhoods in each iteration.

The study proposes a way to incrementally calculate the optimization function for the intermediate solutions generated during the local search process for the evaluation of randomly generated moves. This combined with pre-allocated memory can attribute to decreasing computation time and scaling to more threads while keeping the memory

allocations low.

Furthermore, the research highlights the challenge of limited memory in multi-core CPU machines and the role of garbage collectors in dynamic memory-controlled languages. To mitigate the impact of memory allocation and garbage collection on runtime, the paper proposes pre-allocating arrays and introduces specific data structures. By avoiding dynamic memory allocations and minimizing garbage collection interventions, the algorithm's overall performance is improved.

Pandi et al. [5] reported an impressive average speedup of 55 times when applying the DARP-TS algorithm on GPUs. This substantial speedup can be attributed to the superior performance of GPU cores compared to CPU cores. GPUs are designed with a large number of cores, often numbering in the thousands, enabling them to handle massive parallelization of computations. This parallel processing capability allows GPUs to efficiently process the DARP-TS algorithm and achieve significant speed gains.

In contrast, CPUs typically have fewer cores compared to GPUs, which limits their parallel processing capabilities. Additionally, CPU architectures often abstract away the CPU scheduling process, which can result in unwanted preemption, context switching, and thread overheads. In comparison, GPUs provide more flexible and fine-grained control over low-level scheduling, allowing for optimizations specifically tailored to the problem at hand. The CPUs also abstract the memory allocation and memory block placement using virtual memory, which when dealing with algorithms that use a lot of memory can often result in page cache misses, fragmented memory, and non-sequential memory allocation often leading to additional CPU instructions. This optimized scheduling mechanism and fine-grained memory control contribute to the efficient execution of the DARP-TS algorithm on GPUs, leading to the observed speedup.

It is worth noting that the specific N_SIZE value used in the GPU-based implementation is not specified in the research paper. The N_SIZE parameter determines the number of random moves generated in each iteration to explore the neighbourhood. While the impact of N_SIZE on the speed of convergence is not explicitly discussed in the context

of the GPU implementation, it is an important factor to consider when optimizing the algorithm for different hardware architectures.

Moreover, the research acknowledges that it does not delve into low-level optimizations specific to the target device. The paper focuses more on high-level techniques and considerations to improve the performance of the DARP-TS algorithm on multi-CPU cores. Low-level optimizations, on the other hand, involve fine-tuning the algorithm implementation for a specific device, taking advantage of hardware-specific features, and exploiting optimizations at the instruction level. These low-level optimizations are often crucial in achieving maximum performance on a given device.

In summary, the difference in speedup between the GPU-based approach and the multi-CPU core approach can be attributed to the inherent hardware differences. GPUs offer superior parallel processing capabilities due to their large number of cores, resulting in significant speedups. Additionally, low-level optimizations specific to the target device and the ability to control the scheduling algorithm at the computational cores level play a crucial role in further optimizing the algorithm's performance. This research investigates the limitations of the DARP-TS algorithm on multi-CPU cores and proposes solutions to address these limitations. It emphasizes the importance of estimating the number of "good moves," explores the impact of thread configurations on convergence speed, suggests techniques to optimize memory usage, and garbage collection and proposes a way to incrementally calculate f(s) for intermediate solutions during the local search phase.

Bibliography

[1] D. Kirchler and R. Wolfler Calvo, "A granular tabu search algorithm for the dial-a-ride problem," *Transportation Research Part B: Methodological*, vol. 56, pp. 120–135, 2013, ISSN: 0191-2615. DOI: https://doi.org/10.1016/j.trb.2013.07.014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S019126151300129X.

[2] J.-F. Cordeau and G. Laporte, "A tabu search heuristic for the static multi-vehicle dial-a-ride problem," *Transportation Research Part B: Methodological*, vol. 37, no. 6, pp. 579–594, 2003.

[3] R. W. Calvo and A. Colorni, "An effective and fast heuristic for the dial-a-ride problem," *4or*, vol. 5, no. 1, pp. 61–73, 2007.

[4] S. Ho, S. C. Nagavarapu, R. R. Pandi, and J. Dauwels, "Improved tabu search heuristics for static dial-a-ride problem: Faster and better convergence," *CoRR*, vol. abs/1801.09547, 2018. arXiv: 1801.09547. [Online]. Available: http://arxiv.org/abs/1801.09547.

[5] R. R. Pandi, S. G. Ho, S. C. Nagavarapu, and J. Dauwels, "A generic gpu-accelerated framework for the dial-a-ride problem," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 10, pp. 6473–6488, 2020.

[6] N. H. Wilson, J. M. Sussman, H.-K. Wong, and T. Higonnet, *Scheduling algorithms for a dial-a-ride system*. Massachusetts Institute of Technology. Urban Systems Laboratory, 1971.

[7] H. N. Psaraftis, "An exact algorithm for the single vehicle many-to-many dial-a-ride problem with time windows," *Transportation science*, vol. 17, no. 3, pp. 351–357, 1983.

[8] J. Desrosiers, Y. Dumas, and F. Soumis, "A dynamic programming solution of the large-scale single-vehicle dial-a-ride problem with time windows," *American Journal of Mathematical and Management Sciences*, vol. 6, no. 3-4, pp. 301–325, 1986.

[9] T. R. Sexton and Y.-M. Choi, "Pickup and delivery of partial loads with "soft" time windows," *American Journal of Mathematical and Management Sciences*, vol. 6, no. 3-4, pp. 369–398, 1986.

[10] Y. Dumas, J. Desrosiers, and F. Soumis, "The pickup and delivery problem with time windows," *European journal of operational research*, vol. 54, no. 1, pp. 7–22, 1991.

[11] J.-F. Cordeau, "A branch-and-cut algorithm for the dial-a-ride problem," *Operations Research*, vol. 54, no. 3, pp. 573–586, 2006.

[12] J.-J. Jaw, A. R. Odoni, H. N. Psaraftis, and N. H. Wilson, "A heuristic algorithm for the multi-vehicle advance request dial-a-ride problem with time windows," *Transportation Research Part B: Methodological*, vol. 20, no. 3, pp. 243–257, 1986.

[13] I. Ioachim, J. Desrosiers, Y. Dumas, M. M. Solomon, and D. Villeneuve, "A request clustering algorithm for door-to-door handicapped transportation," *Transportation science*, vol. 29, no. 1, pp. 63–78, 1995.

[14] R. M. Jorgensen, J. Larsen, and K. B. Bergvinsdottir, "Solving the dial-a-ride problem using genetic algorithms," *Journal of the operational research society*, vol. 58, no. 10, pp. 1321–1331, 2007.

[15] S. N. Parragh, K. F. Doerner, and R. F. Hartl, "Variable neighborhood search for the dial-a-ride problem," *Computers & Operations Research*, vol. 37, no. 6, pp. 1129–1138, 2010.

[16] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Computers & Operations Research*, vol. 13, no. 5, pp. 533–549, 1986, Applications of Integer Programming, ISSN: 0305-0548. DOI: https://doi.org/10.1016/0305-0548(86)90048-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0305054886900481.

[17] I. H. Osman and J. P. Kelly, "Meta-heuristics: An overview," in *Meta-Heuristics: Theory and Applications*. Boston, MA: Springer US, 1996, ch. 1, pp. 2–3, ISBN: 978-1-4613-1361-8. DOI: 10.1007/978-1-4613-1361-8_1. [Online]. Available: https://doi.org/10.1007/978-1-4613-1361-8_1.

[18] O. B. Madsen, H. F. Ravn, and J. M. Rygaard, "A heuristic algorithm for a dial-a-ride problem with time windows, multiple capacities, and multiple objectives," *Annals of operations Research*, vol. 60, pp. 193–208, 1995.