Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

The present work was submitted to the LuFG Theory of Hybrid Systems

MASTER OF SCIENCE THESIS

PARALLELIZATION OF A PROBABILISTIC RAILWAY TIMETABLE SIMULATION

Lukas Marian Assenmacher

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr. Nils Nießen

Additional Advisor:
Rebecca Haehn

Aachen, 18.05.2021

Abstract

Railway traffic plays an important role in the public transportation system not only for personal but also for freight transport. Hence it is important, that current and planned railway timetables are evaluated for their validity and compatibility for the given real-world infrastructure network. While several simulation approaches for this task already exist, these are either theoretical or only available commercially with only sparse information publicly available about the inner workings of the underlying algorithms. This is why, Haehn, Ábrahám and Niessen developed a novel, open probabilistic simulation model which offers examination of delay propagation within a reasonable computational complexity. Nevertheless, modern computer hardware has become very powerful, offering multiple CPU cores with potentially even more threads. Thus, by utilizing multi-threading, the aim is to leverage as much of the available hardware for the simulation as possible and possibly offering improvements in runtimes. Therefore, in this thesis I examine possible approaches to parallelization of the aforementioned simulation algorithm. The implemented approaches are then evaluated with regard to their advantages and disadvantages.

Acknowledgments

Firstly, I want to thank Prof. Dr. Erika Ábrahám for giving me the opportunity to work on this topic as well as for her continuous feedback and active input despite having a very busy schedule. I am grateful for her outstanding supervision as the first examiner.
I also want to thank Rebecca Haehn for her excellent supervision as my advisor. Her continuous feedback and input helped me tremendously in the course of this theses.
Furthermore, I want to thank Prof. Dr. Nils Nießen for accepting to be the co-examiner and reviewing this thesis.
Lastly, I want to thank my friends and family for their ongoing support, not only during this thesis but also during my whole studies.
I want to express my gratitude to everyone who helped and supported me especially during these times of a global pandemic.

Thank you all!

Contents

1 Introduction **1**

2 Preliminaries **5**
2.1 Simulations 5
2.2 Our Simulation Approach 7
2.3 Technical Background 11
2.4 Measuring Runtime Changes 16

3 Approach 1: Basic Multi-Threading **19**
3.1 Implementation 21
3.2 Analysis 24
3.3 Summary 27

4 Approach 2: Separated Statistics **29**
4.1 Implementation 30
4.2 Analysis 32
4.3 Summary 36

5 Approach 3: Sub-Graphs **37**
5.1 Implementation 41
5.2 Analysis 43
5.3 Summary 48

6 Approach 4: Parallel Time Processing **51**
6.1 Implementation 53
6.2 Analysis 54
6.3 Summary 59

7 Approach 5: On-Demand Scheduling **61**
7.1 Implementation 62
7.2 Analysis 63
7.3 Summary 69

8 Conclusion **71**
8.1 Summary 71
8.2 Learings 72
8.3 Future work 73

Bibliography **75**

Appendix **79**

A Additional Visualizations **79**

Chapter 1

Introduction

A 2019 report by the European Environment Agency (EEA) examining the means of transportation between 1995 and 2017 concluded, that not only the demand for passenger train travel has risen for the fifth year in a row but also the modal share in the said category increased slightly [10]. This coincides with a competitive metrics report by the Deutsche Bahn (DB), the largest German railway company, as published in late 2020 [9]. It notices, too, a slightly increased share of passenger train travel in 2019 and 2020. This highlights the growing interest and importance in recent years. This increased demand, if not properly accounted for, might put stress on the existing infrastructure and ultimately lead to delays and/or cancellations, possibly resulting in frustrated passengers and monetary losses. However, not only for passenger transport but also for freight, the railway infrastructure plays an important role, as according to the aforementioned EEA report, about 421 tonne-kilometres[1] have been transported in 2017, resulting in a share of about 17% [10]. A similar picture has been observed in 2019 by the DB in Germany, revealing an 18% market share of the German freight transportation sector [9]. While both studies observed a slight decline over the last years, one could argue that freight train travel still plays an important role in the overall logistics market. Like for passengers, delays or other problems in freight transportation may lead to unsatisfied customers or large financial fines. Hence, to reduce economical loss and increase passenger satisfaction, it is important that a train schedule is carefully planned and weaknesses in the current system are strategically analysed to reduce the probabilities of problems with delayed or cancelled trains.

To assess problems in a given railway network, one can run various types of studies and analyses. As one example, one should mention the data science approach which uses recorded real-world data of trains, such as actual arrival and departure times, possible travel speeds, capacities or events, to name a few. This data then can be processed together with a combination of mathematical, statistical and/or programmatic methods (e.g. regression, artificial intelligence, event-driven analysis) to find possible indications or relations on what may be an influencing factor for delays or other complications along a train's itinerary. This field of study has gained popularity in recent years. However, there are some limitations to this type of analysis: they require a large quantity of high-quality historic data to get usable results. If the data used for analyses is of poor quality, e.g. by inaccurate or missing events or if too narrowly/broadly sampled, the results may, too, be inaccurate or biased and may

[1] Freight in metric tonnes per travelled distance in kilometres.

lead to false conclusions. If the quality and quantity is sufficient, these methods may offer valuable insights into the past behaviour, for example in critical situations, from which predictions or assumptions for future behaviour can be made.

One downside of the various data science approaches is, that they mostly rely on already existing data. If this is not available, for example when a new route is formed, one can also make use of simulating the behaviour on the infrastructure. Simulations are independent of historical data and its quality as they primarily rely on the initial state from which the simulation shall start. When considering railways, the respective simulations will be performed on a graph-based structure modelling a real-world railway infrastructure network. On this graph, a given set of trains can then be simulated, for example, according to a timetable schedule, and possible delay factors such as delayed provisioning of trains, failure of a control point or prolonged waiting times at stations can be introduced. However, as it is extremely difficult, if not impossible, to consider every aspect and state within a simulation, a limited set of influencing factors as well as a certain level of detail needs to be used. This is why there are various types of simulations developed focussing on different specifics, e. g. how the trains can be simulated and what factors are considered to what extend. While this might lead to the assumption that simulations are less realistic or precise, the opposite is the case. Simulations can reliably model real-world behaviour without the need to experience it.

Furthermore, the hardware of modern computer systems has become very powerful. Most, if not all, of the currently available CPUs for consumer or professional use pack multiple cores which, in most cases, are also able to handle multiple threads at the same time. This offers a large source of computing power to leverage for simulation algorithms.

Therefore, in this thesis, we inspect the simulation approach as proposed by R. Haehn, E. Ábrahám and N. Nießen in [15] with regard to how parallelization can be implemented and its resulting impacts on runtime. This algorithm was developed by the aforementioned authors as there were no freely available simulation tools offering similar functionality and works by combining simulations and statistical methods to evaluate the propagation of initial delays, e.g. due to delayed provisioning. This is done by using a probability distribution for the trains' locations instead of discrete ones. With this probabilistic approach, the probabilities for a train arriving and residing at a certain infrastructure element at a given point in time can be measured, allowing for a deeper insight into the behaviour of the network and possible influencing factors for delays. The details of the underlying algorithm will be explained in more detail in Chapter 2.

While the simulation algorithm itself was already working, it was only operating in a single-threaded sequential and linear mode. This potentially leaves performance on the table, as modern computer hardware is oftentimes capable of multi-threaded parallel execution. Therefore, the goal of this thesis was the parallelization of the underlying algorithm and to allow for parallel execution leveraging the most potential of the available hardware. For that, I have iteratively parallelized different aspects of the simulation, analysed the effect concerning runtime and scalability and assessed the effectiveness.

The thesis will start with a brief introduction into background knowledge, related work and further information on the simulation in Chapter 2. Afterwards, the different approaches and their respective implementation details will be presented in the subsequent chapters. Each chapter will discuss the idea and details of the approach,

together with the important changes made within the code base as well as a discussion of the results. To conclude this thesis, we give a summary of the overall findings in the last chapter, Chapter 8, discuss possible strengths and weaknesses of the approaches, summarize the key learnings and lastly give an outlook to possible future work on this topic.

Chapter 2

Preliminaries

In this chapter, we want to give a brief overview of some of the currently existing train schedule and railway network simulation approaches and compare them with the probabilistic simulation used in this thesis. Furthermore, a short introduction to the technicalities of multi-threaded and parallel processing will be offered for a better understanding of the technical aspects.

2.1 Simulations

As seen in the introduction, railway traffic plays an important role in the current passenger and freight transportation sectors. Hence, the smooth and uninterrupted operation of train traffic is not only crucial from an organizational but also from a customer's point of view. For example, the planned and punctual operation is one key aspect of customer satisfaction [25] and thus may lead to increased utilization of trains as a mode of transportation forming a financial incentive for the train operators. Hence, to optimally plan a train schedule and assess it for its robustness and viability, oftentimes simulations are used. As the effective and robust railway timetable scheduling concerns any large network, this has been researched in many different ways scientifically and commercially alike. However, each simulation needs to find a balance between performance and detail, as it is impossible to compute a large scale simulation with fine granular detail within a reasonable time. Thus, one can divide the approaches into different categories: *microscopic* and *macroscopic* simulations. The former focuses on small parts of a network in high detail and the latter aiming for a broad, less detailed inspection of a large network. Furthermore, a division can be made between *synchronous* and *asynchronous* simulations, classifying whether all operations, e. g. movements are performed synchronously at the same time or if certain aspects are considered out of order.

Related Work on Railway Simulations. While it is easy to find scientific approaches or especially commercial or industrial simulations for railway networks, it is hard to compare them: each simulation focuses on varying levels of detail[1], prioritizes different key aspects[2] that need to be considered, uses different underlying simulation

[1] i. e. macro- or microscopic simulation

[2] for example capacities, weather, demand, pasenger flow, etc.

principles[1], or addresses completely different goals to achieve[2]. Therefore, it is hard to precisely pinpoint which approach to name and which to omit. Nevertheless, we want to give a brief overview of the most relevant works to put the work of this thesis into context.

Corman and Kecman [6] proposed a predictive approach for assessing train delays by using a Bayesian network. The idea of delay propagation in a train network is similar to how it is done in our simulation, however, the realization is quite different. Firstly, the focus lies on a real-time aspect and secondly Bayesian networks were used for the implementation. The method inspects the delay propagation concerning certain events, such as trains arriving or departing, however, it also takes other delaying events into account. As a result, the researchers found a strong correlation between a train's runtime and the experienced delays. Furthermore, reliable predictions for a trains delay for up to thirty minutes in the future could be made. This approach is not parallelized.

Li and Zhu [22] inspected the influence of passengers on the delay propagation in urban rail transit systems. They proposed a simulation algorithm to assess and predict the effect on the delay by various passenger-based factors. The study focussed on the passenger point of view and additionally examined the delay recovery based on possible passenger decisions with respect to the delay situation. From the description it appears that the simulation is nit processed in parallel.

Briggs and Beck [4] found in their 2006 study that the delay propagation in railway networks could be modelled by so-called *q-exponential functions*, a mathematical modelling function which in fact can be applied to an interestingly wide spectrum of complex, real-world systems.

Büchel, Spanninger and Corman [5] investigated the disruption of train traffic in the greater Raststadt region in the south of Germany after an accident in 2017. The researchers modelled the delay propagation of other trains in the region as an impact of the incident. Furthermore, a predictive model was developed to investigate the influence of said event on the train traffic in the nearby Swiss area. For their simulations, the researchers used the tool *OnTime*.

OnTime [32] is developed by *trafit* in conjunction with the VIA Consulting company and the VIA Chair at RWTH Aachen University. It was initially presented in 2013 in [13] and is a commercial, mesoscopic—between micro- and macroscopic—simulation based on the *Monte Carlo* principle. It combines simulation with analytical and statistical approaches in order to assess the quality of railway timetables concerning their robustness given the infrastructure it is planned for. The exact details on the inner workings are, as can be expected from a commercial application, not publicly shared in detail. According to the developers, it is already in effective use with several large railway companies.

Also developed by the VIA Consulting group is the LUKS tool suite [34]. The suite offers various tools from timetable planning and construction to validity checking and performance analysis of certain aspects of a railway network by the use of various modules, one of which is the LUKS-S simulation module. For this, a timetable is simulated microscopically on a given infrastructure network. Offering two modes, the simulation can either validate a timetable for its viability or test the robustness against delays. It too uses Monte Carlo simulation in multiple simulations runs [20]. According to the publishers, the tools are used by large rail companies, especially

[1] e.g. *Monte Carlo*, or *analytical*

[2] i.e. optimal planning, delay propagation, predictions, etc.

for construction and planning. However, as it too is commercial software, not much information on the precise algorithms or inner workings is provided.

Two other, already very old simulation software suites are RMCon RailSys [30] and OpenTrack [27]. Both tools are available for more than 20 years. Like with LUKS, RailSys consists of multiple management and planning tools, among which a simulation can be found. It is based on microscopic simulation of timetables [18]. Despite the age, to the best of my knowledge, almost no freely available information can be found regarding the internal algorithms. OpenTrack was originally developed by the Institute for Transportation Planning and Systems at the ETH Zürich and focusses on the aspect of timetable simulation using microscopic synchronous simulation [26]. Its main goal is to offer analytical tools to evaluate the feasibility of timetables and find critical properties or potential problems. As with RailSys and despite the name possibly indicating open-source software, next to no information about the actual simulation mechanism is publicly available.

Lastly, we want to mention TrenoLab trenissimo [33, 11]. It is a relatively recent tool using synchronous and microscopic simulation. It was developed to make use of the abundance of data that might be collected in recent years, providing detailed and precise simulations without overloading the available hardware and address issues and missing features of the competing simulations. Sadly, as with the other commercial software, no details on the implementation are available.

For all the commercial simulation algorithms mentioned above, it is not known if and to what extend parallelization techniques are used for the simulation.

To conclude the non-exhaustive literature review on simulations for railway systems and timetables, one can observe that while many tools are available, many of those are only available commercially and without free access. Furthermore, little is known about the actual methods or precise inner workings. Adding to this, many of the available simulations use or seem to use Monte Carlo simulation. Some researches suggest mathematical-statistical models which however is seldomly used in reality. This is why Haehn, Ábrahám and Niessen [15] proposed a probabilistic simulation approach which is used as the basis for this thesis.

2.2 Our Simulation Approach

The probabilistic railway timetable simulation used in this thesis was developed by Haehn, Ábrahám and Niessen and presented in [15]. It proposes a synchronous macroscopic and probabilistic approach by modelling the position of trains not by a fixed location but by a probability distribution for multiple locations. With this, for each train, the probability with which it resides at a certain infrastructure element at a given time is recorded. This allows to investigate multiple scenarios in a single run and also offers insights into the resilience of a railway timetable on a given infrastructure network. This probabilistic model has not been investigated or modelled before and provides a novel take on assessing and investigating delay propagation. As the algorithm is relatively new, currently only initial delays are investigated. Nevertheless, in the future other scenarios could be added to strengthen this approach. In this section, we want to explain the inner workings of the simulation algorithm in its current form in more detail.

Model. To start, we want to focus on the data model. All following definitions are, if not stated otherwise, as they are presented in the original article and will be summarized. The simulation is based on two fundamental models: an *infrastructure network* and a *railway timetable*.

The *infrastructure network* reflects the physical properties of the real-world railway network in question. It is modelled as a directed graph $G = (V, E)$ consisting of a set of vertices V connected by a set of edges $E \subseteq \{(u, v) \in V \times V | u \neq v\}$ without self-loops. The vertices represent so-called *operational control points*, or OCPs for short. These are important infrastructure nodes that a train passes on its itinerary such as stations, switches or more general waypoints. Each edge represents one or more physical *rail tracks* connecting the respective OCPs in the given direction. In our model, multiple tracks that serve the same OCPs will be represented by only one edge, however, to correctly reflect the available number of tracks, each infrastructure element has a *capacity* property. This capacity is then modelled as: $cap : (V \cup E) \to \mathbb{N}^+$. This stores the maximal number of trains that may reside at the given infrastructure element at a certain point in time.

To prevent an unintended blockage of infrastructure elements by trains that are waiting for their initial dispatch or at their final waypoint, two central source and target vertices $v_{\text{source}}, v_{\text{target}}$ are added to the infrastructure graph, so that $G' = (V', E')$ with $V' = V \cup \{v_{\text{source}}, v_{\text{target}}\}, E' = E \cup \{(v_{\text{source}}, v_i) \mid v_i \in V\} \cup \{(v_i, v_{\text{target}}) | v_i \in V\}$. Both nodes v_{source} and v_{target} as well as their respective out- and ingoing edges to the other vertices have an infinite capacity.

Time is modelled within a pre-defined interval $\mathbb{T} = [T_{\text{MIN}}, T_{\text{MAX}}] \subseteq \mathbb{N}_0$, the *time horizon*. Each point in time is discrete and is measured in minutes. However, later in this thesis this is changed to a decimal representation.

On this graph network, the movement of trains is to be simulated. For this, a *railway timetable* is used defining the finite set of trains $\tau_1, ..., \tau_m$, their properties (ID and type) as well as their planned itinerary. Each such train's finite itinerary is defined by an ordered set of vertices $\{v_1, ..., v_n\} \subseteq V'$ and arrival and departure times $a_i, d_i \in \mathbb{T}$ for each vertex v_i. This is called a *timed path* π_{id} for train with ID id with

$$\pi_{id} = \{v_{id,1}(a_{id,1} \mapsto d_{id,1}), ..., v_{id,n_{id}}(a_{id,n_{id}} \mapsto d_{id,n_{id}})\}.$$

For each train we also need to keep track of its current possible positions. For that, let $epdt \in \mathbb{T}$ mark the *earliest possible departure time* at which the train may leave a certain infrastructure element. Then we define a train's state to be $S_{id} = S^V_{id} \cup S^E_{id}$ whereas id is the train's ID and

$$S^V_{id} = \{(v_{id,j}, epdt) \mid 0 \leq j \leq n_{id}, epdt \in \mathbb{T}, d_{id,j} \leq epdt\} \quad \text{and}$$
$$S^E_{id} = \{((v_{id,j}, v_{id,j+1}), epdt) \mid 0 \leq j \leq n_{id}, epdt \in \mathbb{T}, a_{id,j+1} \leq epdt\}$$

Each such state model the trains state at the j-th element on its timed path π_{id}, vertex $v_{id,j}$, or the edge $(v_{id,j}$,$v_{id,j+1})$ respectively.

To accommodate the extension of the graph, as presented above, the itinerary of each train is altered by adding $v_{\text{source}}(a_{id,1} \mapsto a_{id,1})$ as first element of the path π_{id} and $v_{\text{target}}(d_{id,n_{id}} \mapsto d_{id,n_{id}})$ as its last. Therefore, the train starts at the source vertex v_{source} requesting to enter its first regular waypoint at the planned time $a_{id,1}$ and ultimately leaving the last waypoint ending at the target vertex v_{target} at the same time $d_{id,n_{id}}$ it would arrive at its regular final destination.

Lastly, let $E^+(v)$ be the set of all incoming edges of vertex v, or to be precise:

$$E^+ : V \to E, v \mapsto \{(u,v) \mid (u, v) \in E\}.$$

Algorithm. Now that we have achieved an overview of the underlying model, we can proceed with understanding the current implementation of the simulation algorithm. A simplified and abbreviated pseudo-code version can be found in Algorithm 1.

As first step, the algorithm starts with an initialization phase in which the infrastructure network and timetable is loaded. This is then followed by emplacing the trains at the edge from the source vertex s_{source} to their respective first waypoint. This is done according to a pre-defined delay distribution: a train is considered to start its itinerary with the probability for a certain delay. For instance, with a probability of $p_0 = 0.9$ the train starts with no delay and $p_1 = 0.1$ with one minute delay. Therefore, for each possible delay according to the distribution function, a state is inserted with the adjusted *epdt* times and the respective probability. The sum of all probabilities of each train is always 1. For testing the validity of a timetable, the probability of no delay is set to $p_0 = 1$.

After reading the input data and initializing required variables, the algorithm in its current form performs three separate scheduling phases for each point in time, which will be explained in more detail below: 1) scheduling of vertices, 2) edges and 3) passing requests.

In the first step, the *scheduling of vertices*, all trains that are planned to enter an OCP at the current time will be scheduled. For that, the simulation iterates over all vertices and checks the incoming edges for occupying trains that are ready and waiting to enter the current node. If the capacity of the respective vertex is sufficient, the trains are scheduled to leave the edge and enter the vertex with probability $p_{leave} = 1.0$. If the current demand is higher than the capacity, e.g. more trains want to enter than physically possible at that time, the simulation calculates a probability $p_{stay} \leq 1.0$ with which the train can't enter and thus has to remain at its current infrastructure element. The probability p' for entering the respective vertex is then determined by the complement of p_{stay}, ergo $p_{leave} = 1 - p_{stay}$. The total probability p' with which the train is at the next waypoint of its itinerary is therefore determined by the product of its probability p with which it currently resides at the given element and the probability with which it can leave p_{leave}: $p' = p \cdot p_{leave}$. The same holds for the probability that the train can't leave its current position and needs to be reconsidered at the next possible time $\overline{p'} = p \cdot p_{stay} = p \cdot (1 - p_{leave})$. As for *scheduling the edges*, this is done analogously. For each edge in the infrastructure graph, the source vertices are checked for trains scheduled to enter the edge at the given time and the probability for entering is again determined by the available capacity and overall demand.

After these two phases, we have to address an edge case: as we are currently inspecting discrete time steps, there is a fixed resolution (one minute at the moment) with which the scheduling is done. Therefore, it might occur that trains pass multiple infrastructure elements within the same time step, which refer to as *passing requests*. If such a case occurs at the given point in time, we have to schedule all remaining stops on the respective train's itinerary until the next point in time would be reached.

These three scheduling steps are repeated for all points within the defined time horizon $\mathbb{T}$. Once the computations are completed, the simulation performs some concluding analyses, e. g. the number of expected delayed trains.

Datasets. To run the simulation algorithms, different datasets were used. The original, raw real-world data was provided by the Deutsche Bahn to the supervisors of this thesis and was handed down to the author in a preprocessed way. For each dataset

Algorithm 1: Simplified and abbreviated simulation algorithm in pseudo-code.

```
/* Initialize variables, prepare data                          */
initializeSimulation();
/* Schedule Requests for each time step                        */
for Time t ∈ 𝕋 do
    passingRequests ← ∅;
    /* First, schedule the vertices                            */
    for Vertex v ∈ V do
        requests ← ∅;
        /* Gather all requests                                 */
        for Edge e ∈ E⁺(v) do
            for Train τ at e do
                if epdt(τ) ≤ t then
                    requests ← requests ∪ {τ};
        for Train τ ∈ requests do
            if cap(v) < |requests| then
                /* Schedule the request of train τ,
                   probability of entering p' = p · p_leave with
                   p_leave ≤ 1, remaining at the incoming edge
                   with p' = p · (1 − p_leave)                  */
            else
                /* Schedule the request of train t,
                   probability of entering p' = p. If arrival
                   at next waypoint is t: add τ to
                   passingRequests.                             */
    /* Second, schedule the edges                              */
    for Edge e = (u, v) ∈ E do
        requests ← ∅;
        /* Gather all requests                                 */
        for Train τ at u do
            if epdt(τ, v) ≤ t then
                requests ← requests ∪ τ;
        for Train τ ∈ requests do
            if cap(v) < |requests| then
                /* Schedule the request of train τ,
                   probability of entering p' = p · p_leave with
                   p_leave ≤ 1, remaining at the source vertex
                   with p' = p · (1 − p_leave)                  */
            else
                /* Schedule the request of train t,
                   probability of entering p' = p. If arrival
                   at next waypoint is t: add τ to
                   passingRequests.                             */
    /* Third, schedule passing requests, i.e. trains that
       pass multiple waypoints in the current timestep          */
    for Train τ ∈ passingRequests do
        /* Repeat scheduling of vertices and edges along
           the timed path for all waypoints where eptd ≤ t      */
/* Cleanup and evaluate                                        */
evaluate();
```

Dataset	Vertices	Edges
D	16306	35270
N	2646	5622
M + N	4534	9674
N + SO	5146	11028
M + W + SW	6635	14510

Table 2.1: Statistical information about the infrastructure size of the respective combinations of infrastructure graph datasets (North/East/etc.).

the time window was set between $T_{\text{MIN}} = 0$ (00:00h / 12:00am) and $T_{\text{MAX}} = 1620$ (03:00h / 03:00am of the next day) for the simulation. However, each timetable only contained a fixed cross-section with duration d within the given interval chosen by the supervisors. Furthermore, the complete dataset spans the entirety of Germany and is noted by either *D*. The Deutsche Bahn divides this network into sub-networks identified by the managing organizational unit. These are: North (N), East (O), South-East (SO), South (S), South-West (SW), West (W) and Center (M). For each of the above mentioned time windows, a selected combination of those subnets was chosen for inspection. A summary of the respective datasets can be found in Table 2.1 showing the size of the infrastructure networks and Table 2.2 depicting the respective timetable information.

As already mentioned in the introductory chapter, the algorithm has reached a working state, but only works in a single-threaded, sequential mode. As this potentially misses performance on modern computer hardware, the goal of this thesis was to parallelize the algorithm to possibly leverage the full potential of the simulation

2.3 Technical Background

After learning about related simulations and the specifics of the simulation used in this thesis, we want to focus on the technical background. In the following, a brief refresher on multi-threading, concurrency and parallelism in particular will be given.

2.3.1 Concurrency and Parallelism

Although the two words *concurrency* and *parallelism* are often mentioned in the same context and sometimes even used interchangeably, strictly speaking, those are two different things. While both concepts are considering performing tasks simultaneously the ideas are slightly different. Both, Williams [35] and Reinders et al. [28] stress this difference.

Concurrency, or *concurrent execution* describes the principle of two or more tasks being divided or separated and running alongside each other, hence concurrently. For this, only one executor, i.e. execution thread, is required. Concurrency can for example be achieved by letting tasks each run for a small amount of time, rapidly switching between them, hence often referred to as *task switching*. If this happens fast enough, it appears as if the operations are running at the exact same time, whilst they actually are performed sequentially. The term concurrency is often used to stress

Dataset	From	Until	Trains	Avg. Stops per Train
1h/D			5450	20.1
1h/N			702	18.8
1h/N_M	08:00	09:00	1450	17.9
1h/N_SO			1394	17.9
1h/W_M_SW			2629	18.9
2h/D			8020	26.7
2h/M_W_SW	08:00	10:00	3885	24.8
2h/N			1011	25.8
2h/N_M			2223	22.7
3h/D			10511	30.1
3h/M_W_SW	08:00	11:00	5103	27.8
3h/N			1402	28.5
3h/N_M			2956	25.6
4h/D			12993	32.1
4h/M_W_SW	08:00	12:00	6366	29.3
4h/N			1723	31.1
4h/N_M			3644	27.7
5d/D			16966	33.3
5d/N			2370	35.0
5d/N_M	12:00	17:00	4883	30.4
5d/N_SO			4599	31.3
5d/W_M_SW			8289	30.5
5h/D			15744	33.2
5h/M_W_SW	08:00	13:00	7726	30.4
5h/N			2021	33.4
5h/N_M			4366	29.1

Table 2.2: Statistical information about the various datasets used for testing the simulation. Dataset grouped by selected duration and divided into different sub-networks (North/East/etc.). Times in ISO8601 (24h) format.

the logical separation of certain aspects, e. g. the separation of the UI and the actual calculation tasks.

Meanwhile, *parallelism* refers to the concept that tasks may indeed be run at the same time. This can only be realized when multiple executors, i.e. threads, are available. To highlight the optimal utilization of the underlying hardware and thus achieving better performance in certain aspects of a process, e.g. faster processing of large amounts of data, the term parallelism or *parallel execution* is often used.

2.3.2 Technological Advancements

In recent years, computer processors have become very powerful, even in portable devices such as laptops and smartphones. Especially the processors in these consumer and professional devices nowadays seldomly contain only one *processing core*. The latest releases of AMD and Intel, the two biggest chip producers for desktop and server processors, are offering CPUs with core counts ranging from two up to 64 [16, 17, 2]. In professional equipment such as server-grade processors, these numbers may even be higher. Furthermore, for server hardware, it is common to host multiple CPUs on one board, offering an even higher number of combined usable cores. Besides that, most CPU manufacturers have additionally implemented technologies to not only allow the simultaneous execution of one single task, or *thread* to be more precise, per core but multiple. This is often achieved by utilizing task switching, i. e. rapidly switching between the active threads, letting them run in a quasi-parallel manner. These techniques combined offer a way to handle multiple operations at a time, which hence allow for *parallel concurrent execution* of tasks. A graphical example of these concepts can be found in Figure 2.1. This technological foundation offers significant potential for large-scale applications, such as simulations. If optimized for it, they might benefit from the parallel compute power and increase their performance and therewith also their runtime.

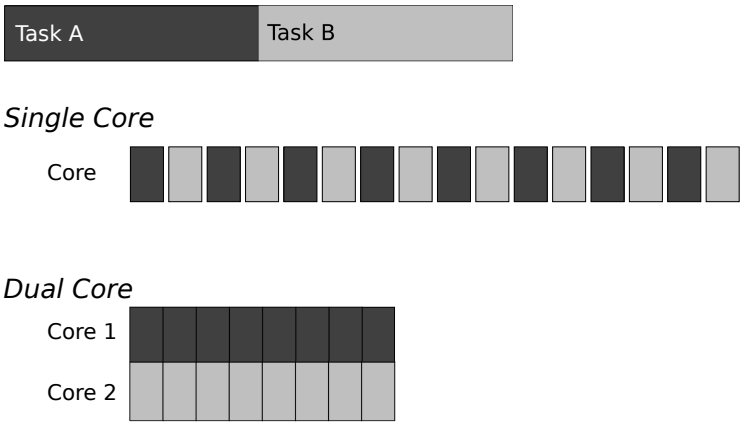


Figure 2.1: Example of concurrency via task switching on a single core and real parallelism using multiple cores. Adapted from [35].

2.3.3 Technical Details

To make use of the multiple cores and threads of modern CPUs, the respective application needs to be optimized for it. There are multiple ways to achieve this: using threads only or concurrent processes. Each one has its own advantages and disadvantages, which we want to briefly highlight alongside the functionality in the following paragraphs.

Threads

Threads can relatively easily be implemented and thus offer a quick way to get started with parallel computing leveraging the multi-threading capabilities of modern CPUs.

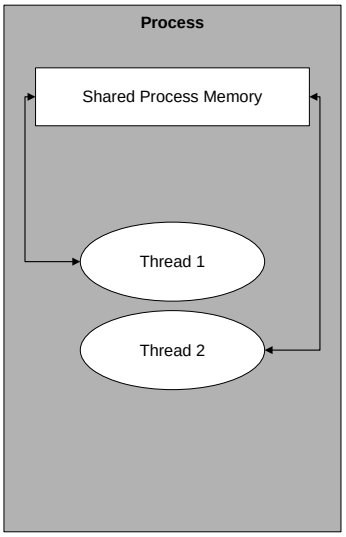
To visualize this concept, imagine a robot that has to perform a certain task, for example counting the number of marbles in a bowl. A thread can here be seen as an arm of the robot. Each arm or thread is capable of performing its task independently from the others but is controlled by the robot and all arms share its memory.

To formalize it: each thread is associated with exactly one parent process with which it shares its memory and I/O, i.e. it runs in the same context as its parent. This principle is depicted in Figure 2.2a. Therefore, the creation of threads is very fast, as no program data needs to be copied. Furthermore, operating on the same data is easily possible. This however means, strict access control for the shared data needs to be implemented, as data corruption may lead to unwanted results and behaviour or cause the entire application to crash. In the worst case, the corruption is not noticed and leads to corrupt results. To mitigate this, access synchronization can and should be used when simultaneously modifying the data. Typically, access synchronization for shared resources can be managed through *mutual exclusive* read/write access control. This principle is often referred to as *mutex*. A mutex can be seen as a lock that can be exclusively obtained by one single thread. If other active threads want to access this lock while it is held by another thread, they have to wait until it is released again. With this, it can be ensured that no two threads are simultaneously altering a critical data element. Computational results are being shared within the current application context. This, all in all, leads to a quick and efficient way for enabling parallel computing with only a small overhead and re-structuration of the code.

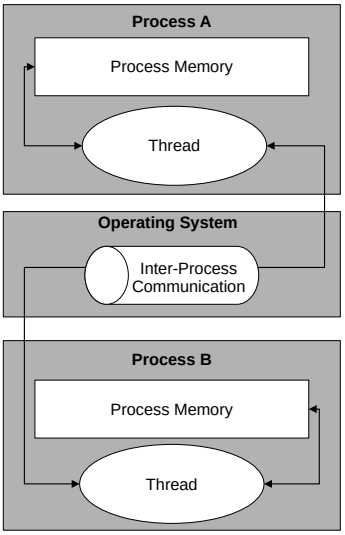
Multiple Processes

The second way to add parallelization to a program is through the use of multiple processes. To again visualize this concept, we return to the analogy from above: a process can be seen as the robot itself, with all arms and accessories attached. In order to help the initial robot, we can assign another robot the task of counting the marbles as well. Now both robots can work on this task. However, they do not share the same memory and work completely separate from each other. Thus, the marbles need to be shared and the total amount of marbles can for example be divided by two so that each robot counts its separate portion. In the end, both need to communicate their results to each other, so that the total amount is correct.

More formally, a process can create another process, a so-called child-process, to help with a certain task. The child-processes each have their own context, meaning they generally do not share the memory of the parent process. Hence it is crucial to plan in advance what process needs which data. One could simply hand over a complete copy of the parent process' data, but this might consume available RAM fast



(a) Threads are sub-tasks within the current process' context and share its memory. Adapted from [35].



(b) Processes are considered their own entities and do not share the same memory. Communication needs to be done through the host OS. Adapted from [35].

Figure 2.2: Comparison between threads and processes with regard to their runtime context.

or cause inconsistencies as possible changes need to be communicated to all processes, e g. to avoid double processing. Additionally, as memory on consumer machines is often limited, this may pose a problem, however on larger computing clusters, this is less of a concern. Nevertheless, the data to be processed by the single processes needs to be selected in advance, allocated and handed over to the respective processes. Due to this memory separation, the danger of data corruption is low to non-existing, which leads to only few synchronization code that needs to be accounted for. However, communication between processes, the *inter-process communication*, is very costly as it needs to be done through the operating system (OS). Due to the involvement of the OS, it generally takes more time to instantiate and communicate, introducing a significant overhead here. The concept of processes can be found in Figure 2.2b.

To conclude, the use of multiple processes may lead to better performance, especially on larger computing machines, such as clusters, with more than one CPU.

The probabilistic railway timetable simulation, that builds the foundation of this theses, is written in C++, which added native support for multi-threading with technical specifications released in the standard C++11 [36, 35] with further refinements in C++14 and C++17 [35]. However, there are also third-party solutions bringing other implementations of multi-threading to C++ , which were often used to add support for multi-threading and multiprocessing prior to the release of the aforementioned versions. When working with processes, however, the C++ standard so far does not define a way to natively manage the internal processes as it offers with threads [35].

This means, that one needs to either write the platform-specific management code on its own or rely on already existing implementations that are not part of the official standard. Due to the limited scope of this thesis, I focussed on the implementation of parallelism in the simulation algorithm with threads as the C++standards offer an comprehensive way for implementing parallelization on consumer machines.

2.3.4 Related Work on Parallelization

In the following, we want to address the literature on parallelization of algorithms or simulations in particular. After a literature review, it became obvious that the comparability and relevance to the topic of this thesis is hard to find, especially with the parallelization of this particular simulation in mind. This is mostly due to the fact that the way how an algorithm can be parallelized is heavily dependant on the algorithm itself and how it is designed on the one hand and on the other hand the programming language and partly also the operating system. There is no universal way, how parallelization has to be implemented. Furthermore, since the introduction of native multi-threading in the C++11 standard in 2011, the way this can be done in C++ , in particular, has changed drastically in recent years. While some approaches might be similar enough to take advantage of the methods or findings, in this particular case and to the best of my knowledge, this was not applicable and the relevance was found to be too minute. This too highlights the relevance of this thesis. The most helpful information, at least to me, was found in the literature on parallelization in general, for example in Williams [35] and Ljumovic [24]. Here, caveats and general advice were given to look out for when implementing parallelism. Furthermore, Reinders et al. [28] and Fernández-Villaverde and Valencia [12] also offered some insights but mainly focussed on specific implementations of parallel computing in C++.

For the interested reader, I recommend to start with the following literature:

- Danelutto et al. [7]
- REPARA Project [29]
- Sah and Vaidya [31]
- Jahr, Gerdes and Ungerer [19]
- Liang, Humos and Pei [23]
- HaCon [14]

2.4 Measuring Runtime Changes

As the goal of this thesis is to bring parallelization to the existing simulation algorithm, this will change the runtimes of the algorithm inevitably. Hopefully, these changes will provide some sort of speed up. To reliably compare the measured runtimes with each other and put them into context, we define the following metrics. These are adapted from [1].

Definition 2.4.1 (Measured Speed-Up using Threads)**.** *Let N be the number of threads being used, t_B the average runtime for the single-threaded baseline and t_N the average runtime using N threads, then*

$$S_N = t_B / t_N$$

the speed-up gained from executing the current program concurrently with N threads. The value can be interpreted as "How many multi-threaded runs can be made in the time the baseline run takes?". Hence, the value should be $S_N > 1$ and optimally $S_N \gg 1$. If $S_N = 1$ or falls below 1, it indicates that the multi-threaded variant is slower.

The gain in runtime speed-up is limited through an upper bound. Obviously, there is no magic way to increase the speed-up indefinitely. As already verbally described by Amdahl [3][1] in 1967, the maximum achievable speed-up of a program that can execute p % of its code in parallel is bounded by

$$S_{max} = \frac{1}{1-p}.$$

A second aspect to consider is the difference in runtime between the not explicitly threaded baseline version and a version explicitly using multi-threading but only one thread. This allows measuring the overhead introduced in bookkeeping to explicitly allow multi-threading to be used. This can be seen as the *work efficiency*, as defined by [1].

Definition 2.4.2 (Work Efficiency)**.** *Let t_B be the average runtime for the single-threaded baseline and t_1 the average runtime using the threaded version with only one thread, then*

$$r = t_1 / t_B$$

is the efficiency of the threaded version compared to the baseline. The value can be interpreted as "How much overhead is introduced when using the current threaded variant?". Hence, the value should be optimally $r = 1$, meaning no overhead is introduced. The larger the value gets, the more overhead is introduced.

[1] Amdahl only verbally mentioned this limit. In the following, we will use the generally accepted mathematical definition as it for example presented in Reinders et al. [28]. A visual explanation can be found by Kipfer in [21].

Chapter 3

Approach 1: Basic Multi-Threading

In the following chapters, various approaches for parallelizing the Probabilistic Railway Timetable Simulation by Haehn et al. [15] that have been explored will be discussed. To start, in this chapter we will explain the initial approach, show its implementation details and analyze the impact on the simulation runtime. From the results, further possible improvements will be deducted that potentially further boost the performance of the simulation, which subsequently will then be elucidated in the next chapters.

Like it was mentioned in the previous chapter, *threads* offer a lightweight way to make use of parallel computing on modern computer hardware. As the implementation of threads was facilitated in C++ in the recent versions 11 to 17 of the C++ standard by the introduction of a standardized programming interface, it seemed a good point to start. Therefore, firstly this initial and very basic approach will be explained, which will then later be improved according to the respective results and insights gained from the implementation process and subsequent analysis.

As a starting point, a basic extension of the given simulation algorithm was chosen as a proof-of-concept. As mentioned in the algorithm description in Chapter 2.2, the simulation follows a fixed and rather intuitive order of operations: each of the vertices needs to be scheduled first, then edges and lastly passing requests can be scheduled. The scheduling of vertices moves trains from the incoming edges to the vertices and scheduling of vertices then transports the trains from the vertices to the corresponding edges. Through this order of operations, it is guaranteed that all trains are considered correctly as trains are always considered in the correct way and cannot get stuck or misplaced. Therefore, the integrity of all states is given. Therefore, changing this order would require rewriting some major parts of the underlying structure and careful planning and consideration as well to further ensure a valid simulation. Hence, to start, we wanted to investigate the possibilities within the current implementation and began the parallelization of these three scheduling loops. That is, threads were added to handle the scheduling of vertices and edges in the main iteration through the respective time steps. For this, the vertices were divided in up to n equal-sized chunks, where n is the number of simultaneous threads to be used. The number can be adjusted by the user, but is limited to a maximum by the system: a system can only execute a certain number of threads in parallel. While it is usually possible

(a) Initial exemplary infrastructure graph.

(b) Exemplary infrastructure graph divided into two chunks for vertices and edges each as indicated by the differing colours/markings.

Figure 3.1: Visualization of infrastructure network split into two chunks to be used with two threads.

to define more threads than this number, no more threads can be executed by the system parallelly which reduces the efficiency with regard to the runtime. Hence, the system maximum was set as a possible maximum for the simulation. This maximal number was obtained through the `std::thread::hardware_concurrency()` [1] directive. The division of infrastructure elements, i. e. the vertices and edges of the underlying graph, into the aforementioned chunks did not follow any prioritization in this attempt and was handled in the order of the respective IDs. Formally speaking, let $V = \{v_1, ..., v_i\}$ and $E = \{e_1, ..., e_j\}$ be the set of vertices and edges respectively. Furthermore, let n be the number of chosen chunks. Then V and E are split into n equal-sized subsets each: $V_k = \{v_{1+(k-1)\lceil i/n \rceil}, ..., v_{k\lceil i/n \rceil}\}, k \in [1, n], i \in \mathbb{N}, v \in V$ or $E_{k'} = \{e_{1+(k'-1)\lceil j/n \rceil}, ..., v_{k'\lceil j/n \rceil}\}, k' \in [1, n], j \in \mathbb{N}], e \in E$. It can be seen that the resulting chunks are pairwise disjunct, i. e. $V_k \cap V_l = \emptyset$ for all $k, l \in [1, i], k \neq l$ and analogously $E_{k'} \cap E_{l'} = \emptyset$ for all $k', l' \in [1, j], k' \neq l'$. This splitting is visualized in an exemplary manner in Figure 3.1. In the given example, two chunks are used ($n = 2$). The six vertices ($i = 6$) are ordered by their ID (here 1-6) and split into two groups of three, where the first three vertices (1-3) and last three (4-6) are each grouped together. Similarly, from the eight edges ($j = 8$), which here are labelled from A-H for a better distinction to the vertices, the first four edges (A-D) and last four edges (E-F) are each grouped in the same chunk.

As for the changes in the algorithm, little was modified in this first approach. When threading was enabled by the user, the loops scheduling the vertices and edges in each time step were substituted by a threaded variant. This variant assigned the previously created threads to the respective chunks to work on in parallel. Each stage, i. e. scheduling of edges is then executed subsequently by the respective chunk's thread. Ergo, only the scheduling within a stage at a given time step is done in parallel.

As threads work on the exact same shared data as the parent process, measures against simultaneous access and modification need to be put into place, preventing data corruption, inconsistencies or erroneous results. This was realized with the help of C++ `std::mutex` [2] and `std::unique_lock` [3]. The latter is a standard wrapper implementation to facilitate the management of the underlying mutexes. A central mutex was created and whenever a variable that is used by multiple threads

[1] see `https://en.cppreference.com/w/cpp/thread/thread/hardware_concurrency` (visited last on 16. April 2021)

[2] see `https://en.cppreference.com/w/cpp/thread/mutex` (last accessed on 16. April 2021)

[3] see `https://en.cppreference.com/w/cpp/thread/unique_lock` (last accessed on 16. April 2021)

and therefore might be modified simultaneously was accessed, That is, when reading or modifying it, the mutex was locked using the unique_lock. This ensures the consistency of the data along all threads.

As can easily be seen, due to the waiting for each stage to finish before continuing and the use of only one mutex, this approach by far does not nearly exhaust the full potential of the available hardware. However, as we approached the problem in an iterative way, the goal in this first attempt was to achieve a first working parallelized prototype which then can be improved on in further iterations. The implementation-specific details will now be explained in the following section.

3.1 Implementation

As already indicated above, there is no real limit to how many threads can be created by a program, given enough physical resources are available. Nevertheless, due to hardware limitations, a system is only capable of handling a certain amount of threads at a time in parallel. If more threads are created than the system can execute concurrently, other threads need to be halted or extensive task switching needs to be applied which can diminish any improvements or even worsen the overall runtime. Depending on various circumstances, it may also be beneficial to not use the maximal number of allowed concurrent threads but only a restricted subset as it may yield even better performance, e. g. due to better load distribution. For one, especially on low power hardware such as laptops, but also on some desktop CPUs the speed with which a core works, depends on the current load. If only a few cores, and thus threads, are used, the CPU can focus all the available power to only those two cores, allowing for higher boost frequencies. If more cores are used or the entire CPU is stressed, the total power needs to be distributed to all cores and often the maximal frequency is lower. This may affect the overall performance. Additionally, it requires optimization to benefit from more cores as the overhead in preparing the data, assigning it to threads, merging the results, etc. may prevail the benefits from running the tasks in parallel. The maximal number of available concurrent threads is limited by the CPU and the number of its cores in particular. Each core can process at least one thread concurrently, however, most processor manufacturers have implemented further improvements so that, in most cases, up to two threads can be executed on a single processor core. To make optimal use of threads on a given machine, the maximal number of concurrent threads should be dynamically received. In C++, this can be done through the `std::thread::hardware_concurrency()` directive. The result is set as the maximal possible threads for the simulation. Furthermore, an argument switch was implemented to easily specify with how many threads the simulation shall be run. This was especially done to ease testing and evaluation of the parallelized application in different configurations. One can pass `-t 0` to the compiled simulation binary to use the non-threaded base variant, `-t n` to use n threads whereas n will automatically be limited to the previously determined maximal possible number of threads on the current system. To automatically enable the maximally supported count of threads, n can be set to $n = -1$.

After knowing how many threads are maximally possible and how many of those shall be used, the infrastructure network can be divided accordingly into subsets of infrastructure elements for each thread to handle. In this initial approach, this was done by dividing the set of vertices and edges into equally-sized *chunks*. It should be

noted, that no prioritization or special grouping was applied. The corresponding code can be seen in Listing 3.1. An exemplary visualization of the splitting was presented in Figure 3.1.

The chunks were modelled as pairs of `Graph::VertexIterator`[1] or `Graph::EdgeIterator` objects respectively, which represent the start and end of the chunk. While the chunks could also be stored as lists of vertices or chunks, I found it simpler, faster and more efficient to save it that way. This is, however, only possible as the chunks are assigned by the order of the respective element's ID. As can be seen in the referred code listing, the division of vertices was relatively straight forward, whereas the assignment of edges is slightly less elegant. This is due to the fact that the *boost* library, which is internally used to model the graph, allows VertexIterators to be modified by the addition of multiple steps at once whereas EdgeIterators can only be advanced by one step at a time through addition. Therefore, the edge iterators are not advanced by addition but by using the C++ `advace` method for pointers. It can be seen that the chunks are divided into sized of `vertex_chunk_size` for vertices and or `edge_chunk_size` for the edges, which correspond to the number of vertices ($\lceil i/n \rceil$) or edges ($\lceil j/n \rceil$) per chunk rounded to the next higher integer value.

Listing 3.1: Division of vertices and edges into the chunks to be utilized in threads.

```
1  Graph::VertexIterator vertexIterator, vertexEnd;
2  std::tie(vertexIterator, vertexEnd) = vertices(g);
3
4  Graph::EdgeIterator edgeIterator, edgeEnd, edgeEndChunk;
5  std::tie(edgeIterator, edgeEnd) = edges(g);
6
7
8  for (int thread = 0; thread < threads; thread++) {
9    vertexChunks[thread].first = (thread * vertexChunkSize) + vertexIterator;
10   vertexChunks[thread].second = std::min((((thread + 1) * vertexChunkSize) +
         vertexIterator), vertexEnd);
11
12   edgeChunks[thread].first = edgeIterator;
13   advance(edgeIterator, edgeChunkSize);
14
15   // Advance does not check boundaries, use edgeEnd as end if last chunk not
         completely full.
16   edgeEndChunk = (thread == (threads - 1)) ? edgeEnd : edgeIterator;
17   edgeChunks[thread].second = edgeEndChunk;
18 }
```

As the necessary preprocessing and preparations are completed with the chunk generation, we can now proceed with adjusting the scheduling parts. For each time step and each stage, the algorithm now assigns each thread the respective chunk. Within this thread, the iteration over the respective elements, i. e. vertices or edges is done. Hence, each thread works on only the respective limited subset of the infrastructure elements. This implementation can exemplarily be seen for the scheduling of edges in Listing 3.2, whereas the scheduling of vertices is done analogously and will be omitted here.

[1] Iterators are pointer objects, which are used by C++to quickly iterate over a collection. Vertex- or EdgeIterators, respectively, are special instances of iterators for traversing through the graph structure used to represent the infrastructure network.

Listing 3.2: Adjusted edge scheduling for multi-threading.

```
if (using_threads) {
  for (int thread = 0; thread < threads; thread++) {
    auto edgeIteratorChunk = edgeChunks[thread].first;  // Start of chunk
    auto edgeEndChunk      = edgeChunks[thread].second; // End of chunk

    // Run thread with edges from resp. chunk
    results.emplace_back(thpool.enqueue(
      // Lambda function as task description for thread
      [](auto sim, auto eit_s, auto eit_end, auto timestep) {
        // For each edge in the chunk,
        // perfrom scheduling for current time step
        for (; eit_s != eit_end; ++eit_s) { sim->schedule_edge(*eit_s,
            timestep); }
      },
      this, (edgeIteratorChunk), (edgeEndChunk), t));
  }

  // Wait for the threads to finish
  for (auto && threadResult: results) {
    threadResult.get();
  }
  results.clear();
} else {
  // Schedule the edges conventionally

  Graph::EdgeIterator eit, eend;

  for (std::tie(eit, eend) = edges(g); eit != eend; ++eit) {
    schedule_edge((*eit), t);
  }
}
```

To avoid conflicts with simultaneous access and thus possibly corrupting data, synchronization needs to be put in place. This can be achieved in C++ by using mutex locks implemented in the standard library `std::mutex`. Mutexes, an acronym for *mutual exclusion*, can be used as locks and thus limiting the access to certain parts of the data for, as the name suggests, mutual exclusion. This ensures that two threads that want to access a variable that is used in common cannot do this at the same time. In the first parallelization attempt, one mutex lock was created and locked, whenever a write operation to a common, shared variable in the code should be executed.

In an early stage of this initial approach, the threads were created whenever needed. This was found to introduce an overhead and prolong the runtime due to the creation and destruction of each thread requiring some system resources. Even-though the resources needed for the creation are very few, this all in all summed up to a noticeable difference. Hence, to reduce this overhead to a minimum, a thread pool was used. The C++ standard does not specify a direct solution for this, which is why we chose a lightweight external implementation[1] for this. This creates a pool with a fixed number of threads, to which work can be assigned. As long as no work is assigned, the threads are in a sleeping state and do not consume any resources. Only then the respective threads start working.

As a further note: the scheduling behaviour of *passing requests* was not altered at this point, as it requires more consideration and care to be executed in parallel. This is due to the fact that passing requests possibly can span a large part of the infrastruc-

[1] `https://github.com/progschj/ThreadPool/blob/master/ThreadPool.h`

ture network, meaning that chunk borders could be crossed. This, therefore, implies that the pre-calculated chunks cannot be used in their current form. Additionally, the passing requests are stored centrally as a vector and the modification or simultaneous access would be needed to be taken care of. While these issues can be mitigated, this was not in the scope of this first approach.

3.2 Analysis

After the changes were made, their effect needed to be evaluated in order to see if they yielded any improvements. To reliably test the algorithm and see differences in performance when using different amounts of threads, a benchmark tool was written. This tool can be configured with an input directory containing the datasets to be tested and the number of runs that should be executed per thread-dataset combination. For each dataset, all possible number of threads are tested, i. e. starting with the baseline version then increasing the thread count to be used until the system maximum is reached. For each run, the runtime of the simulation algorithm is measured and noted in a CSV file for further analysis. As for the evaluation, a small Python[1], script was written which loaded the recorded data into a Pandas[2] DataFrame for further analysis. Additionally the run times were plotted using the Seaborn library[3] for better visual representation.

For the evaluation, the simulation was run on two distinct machines which represent relatively recent consumer hardware: a laptop and a regular desktop computer, the specifications of which can be found in Table 3.1.

Property	**Laptop**	**Desktop PC**
CPU	Intel Core i7 8550U	AMD Ryzen 5 3400G
Frequency	1.9-3.0 GHz	2.2-4.0 GHz
Cores/Threads	4/8	4/8
RAM	16 GB	16 GB
Operating System	Ubuntu 20.04 via WSL	Fedora 33
C++ compiler	gcc/g++ 9.3	gcc/g++ 10.2

Table 3.1: System specifications used for testing.

The benchmark then was used to test all the available datasets as presented in Table 2.2 with thread count combinations, respectively. Each combination was run at least three times to account for varying system load and to get a relatively constant and representative measurement. While more samples would be better, due to the long runtime for all test combinations (over four hours) this was not feasible. However, for the desktop, it was observed that the variation in runtime typically was very low supporting the decision. Furthermore, due to the steady results, the desktop computer will be used as the main reference in the further analysis.

[1] `https://www.python.org/`, version 3.9
[2] `https://pandas.pydata.org/`, version 1.2.2
[3] `https://seaborn.pydata.org/`, version 0.11.1

Visual Results. The measured runtimes from the benchmark were used also to generate a plot for visual representation. For that, we used Python and Pandas for reading and managing the recorded runtimes in CSV format and plotted these results using the scientific plotting library Seaborn. The respective results can be seen in Figure 3.2. A detailed graph of the *5h* datasets is attached in the appendix in Figure A.1a.

As it can be seen, the improvements gained on the desktop with this approach are only marginal. In some cases, especially when considering larger and thus longer running datasets, the runs with two to three threads performed slightly better than their baseline counterparts "0 Threads"). However, the improvements diminish with more than three threads and the performance rapidly decreases. This indicates that a large communication overhead between the threads, extended waiting and/or poor task distribution is present when more threads are being used.

It can be additionally noted that the test results from the laptop computer have a much more pronounced curve. Here, no improvements could be found compared to the baseline runs. This is probably due to the small "slim and light" form factor of the given device: On the one hand, the device is relatively small and only offers a very compact cooling solution limiting the overall possible dissipated power. On the other hand, the specific CPU is a low power U model and furthermore can only make use of higher boost clocks when one to two cores are utilized. If more cores are stressed, the total maximal frequency is much lower, which can impact the overall performance.

Figure 3.2a shows the results for the runs on the desktop PC whereas Figure 3.2b depicts the results for the mobile laptop.

Analytical Results. For the evaluation, we defined the *speed-up* (see Definition 2.4.1) and *efficiency* (Definition 2.4.2) in Section 2.4. To measure the efficiency, we need the runtime of the baseline (original simulation with non-threaded scheduling) as well as the runtime with only one thread. This allows us to determine the overall overhead that is introduced by the parallelization, e. g. thread creation, assigning the chunks and running in threaded mode. Additionally, the overall speed-up shall be evaluated. For this, we decided to compare the runtime of the respective fastest variant to the baseline. As we have run each configuration multiple times, we applied the average to the runtime and used the resulting values for our evaluation. A summary of these results on the desktop computer can be found in Table 3.2.

As we introduced parallelization, we would hope to find some sort of improvement with regard to the runtime. However, as we already highlighted previously, in this approach, we do not expect the runtimes to be significantly faster. This is due to the rudimentary division in chunks and non-optimal usage of mutexes.

This is reflected in the actual results. As can be seen in the above-mentioned table and the visual results, the yielded improvements are rather minute: in most of the cases, the baseline run was the fastest. Only the 2h, 3h and 4h partially recorded some speed-ups.

Speed-Up. As already visible from the graphs, the speed-up in some cases is a slowdown. To put it into numbers, the best performing run from the *1h/D* dataset for example is used for the formula defined in Definition 2.4.1. Or to be precise: the baseline run was the fastest with an average runtime of 85.6 seconds, the second-fastest was with two threads and a runtime of 87.6 seconds. The runs with one thread averaged 98 seconds.

Dataset	Baseline [s]	1-Thread [s]	Fastest [s]	Speed-up	Efficiency	Best Thread
1h/D	85.558	98.069	87.617	0.976	1.146	2
1h/N	12.845	16.217	13.452	0.955	1.263	2
1h/N_M	26.762	28.318	27.148	0.986	1.058	5
1h/N_SO	30.332	31.903	30.268	1.002	1.052	5
1h/W_M_SW	38.144	40.871	39.571	0.964	1.071	5
2h/D	87.281	105.007	84.867	1.028	1.203	2
2h/M_W_SW	35.128	42.625	34.283	1.025	1.213	2
2h/N	13.085	15.533	12.643	1.035	1.187	2
2h/N_M	23.276	27.373	22.793	1.021	1.176	2
3h/D	93.496	99.491	90.178	1.037	1.064	2
3h/M_W_SW	39.644	42.509	36.991	1.072	1.072	2
4h/D	96.851	114.677	90.811	1.067	1.184	2
4h/M_W_SW	38.241	45.059	36.292	1.054	1.178	2
4h/N	14.927	16.025	15.369	0.971	1.074	5
4h/N_M	26.977	27.778	27.778	0.971	1.030	1
5h/D	106.378	116.535	104.272	1.02	1.095	2
5h/M_W_SW	43.081	46.591	44.689	0.964	1.081	4
5h/N	14.588	15.902	14.117	1.033	1.090	2
5h/N_M	26.369	28.41	28.026	0.941	1.077	2
Max	106.378	116.535	104.272	1.072	1.263	5.0
Min	12.845	15.533	12.643	0.941	1.030	1.0
Mean	44.893	50.468	44.272	1.006	1.122	2.684

Table 3.2: Benchmark results on the desktop PC for the various datasets. Average values, runtimes in seconds.

$$
\begin{aligned}
t_B &= 85.6s && \text{(average baseline)} \\
t_2 &= 87.6s && \text{(average runtime with } N = 2 \text{ threads)} \\
S_2 &= t_B/t_2 \approx 0.977\%
\end{aligned}
$$

This indicates that the fastest runs of the threaded variant on this particular dataset on average are about 2.3% slower for this particular dataset. This general behaviour seems to be overall present in the other datasets, as can be seen in the *Speed-up* column of Table 3.2. While some are faster, others are slower like the example above. This reflects in the mean value of approximately 1.0 and indicates that the best-threaded approaches are, on average, as fast as the baseline. Furthermore, the maximally achieved speed-up was with a value of 1.07 relatively insignificant. This demonstrates that improvements and optimizations are needed.

Efficiency. A similar behaviour is noticeable when inspecting efficiency as defined in Definition 2.4.2. Here too, this approach fared relatively poorly. To take the *1h/D* dataset as example again: the execution of the threaded variant with only one thread took 98.0 seconds whereas the baseline was completed in 85.6 seconds on average. This results in an efficiency value of:

$$t_B = 85.6s \qquad \text{(average baseline)}$$
$$t_1 = 98.0s \qquad \text{(average of parallel w/ 1 thread)}$$
$$S_N = t_1/t_B \approx 1.145$$

This means that about 15% overhead was introduced in the threaded variant. This is quite a lot and also needs to be improved on.

When comparing this to the other values observed and shown in column *Efficiency* of Table 3.2, for every dataset this value is greater than 1. The average value for all datasets is 1.122 and approximately aligns with the example calculation resulting in a general overhead of roughly 12%.

Profiling. To further assess possible bottlenecks and find long-lasting operations are used within the application, a profiler was used. For this, the included profiler of the selected IDE (CLion) was chosen, which generated and interactively processed the results of the Linux tool *Perf*.

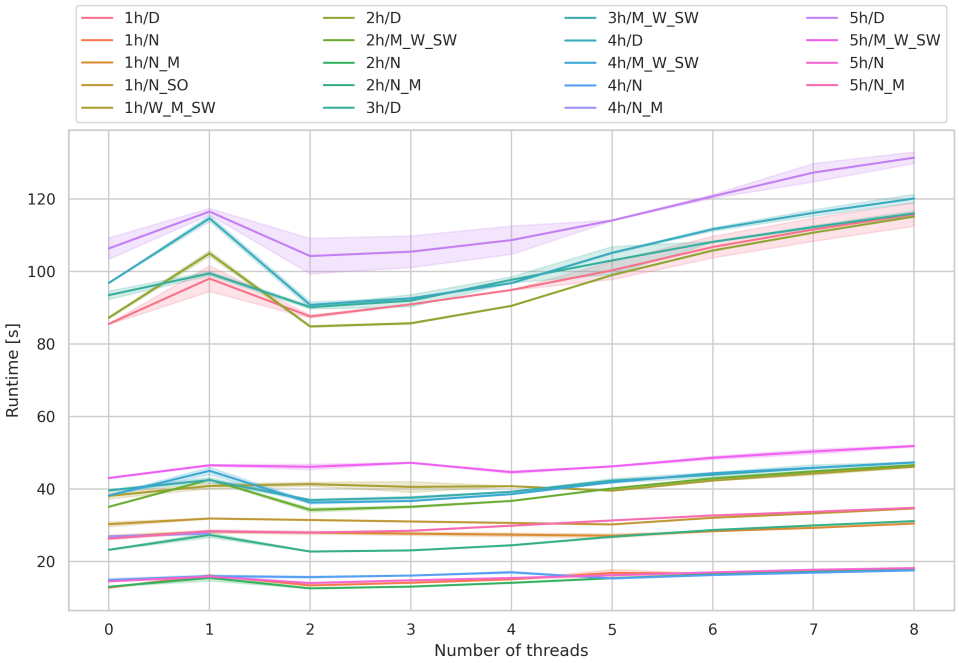
After running the profiler on the *5h/D* dataset, which was chosen for its size and performance results, it can be seen that about 22% of the runtime was spent locking and unlocking the mutex locks. This is a relatively large share and indicates why the performance was sub-par. It was additionally noted, that variables containing statistical information were frequently accessed. As these necessarily need to be guarded for simultaneous access to ensure correctness, these variable changes are commutative meaning that the order in which the changes are applied does not matter. This, therefore, is to be investigated for further improvements.

Furthermore, about 30% of the time was spend merging and 20% unblocking. The former is especially interesting as only a few merges are actually made, indicating possible improvements for the actual simulation algorithm, which however is not within the scope of this thesis.

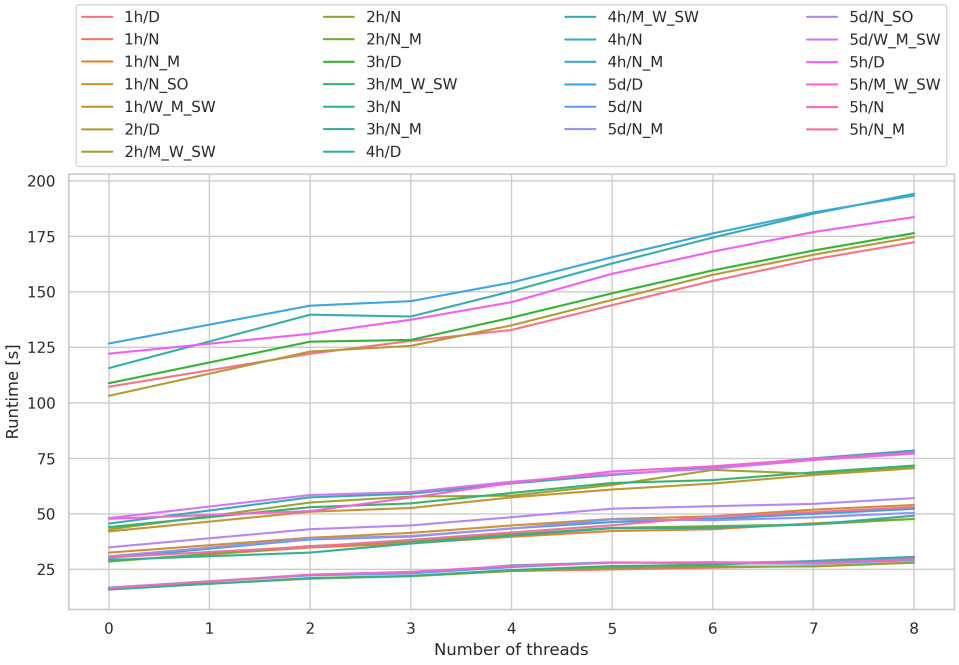
Learnings. In the results and evaluation presented above, it was shown, that the algorithm indeed can be parallelized as envisioned. However, it is obvious that the current implementation is far from efficient or fast and currently introduces more overhead as it provides speed-up. From the profiling results, I concluded that at the moment much time is spend locking and waiting for the write-lock to be obtained.

3.3 Summary

In this first approach, the simulation algorithm modified to support basic multithreading. This allowed the simulation to schedule part of its code in parallel. However, the analysis has revealed some discrepancies between the laptop and desktop computer. While the desktop noticed some minor improvements, the laptop could not improve the runtime at all. In this first iteration, nevertheless, we were able to show that the simulation indeed can be parallelized. The parallelization in fact showed already some minor improvement with much headroom in future iterations.



(a) Runtimes on the desktop computer.



(b) Runtimes on the laptop computer. Note: Due to poor performance, only one run for each thread/dataset run.

Figure 3.2: Runtimes of the initial approach with various datasets for differing thread counts.

Chapter 4

Approach 2: Separated Statistics

The evaluation of the initial parallelization attempt revealed one main issue that influenced the runtimes: long waiting for the mutual exclusive access when entering critical sections, which was aggravated by a large amount of statistic write operations requiring said exclusive access. The first part of this issue, i. e. the long waiting times to obtain a lock is not particularly striking as all write requests were locked by only one available lock regardless of what was written to. This can be prevented or at least reduced by locking logically independent variables separately. That is, if two variables do not share any logical connection and are independent of each other, these can be locked by separate mutexes to indeed further guarantee single simultaneous access but also allow for access of the separate variables. This is implemented in this approach.

Furthermore, in the initial approach, we noticed that there was a large number of write operations to statistics variables. While these are not specific to the parallelization, they nevertheless are important for the simulation algorithm itself. These variables contain, as the name suggests, statistical information about the current state of the simulation such as a number of scheduled vertices and edges or waiting trains. So as these statistics are mostly counters or variables that are computed by just simple addition or multiplication. This can be leveraged to an advantage as these mathematical operations are commutative, i. e. the order does not matter provided operators are not mixed together. As the latter was not the case, we decided to move the modification of the statistics variables into the context of the thread. That is, each thread has a separate copy of the statistics variables and only modifies this particular instance. Once the simulation is finished, the separate individual instances can simply be merged together. This concept was realized by creating a `Statistics` class, which takes care of all modification operations. It also handles the final merge with the other instances. Each thread is associated with one instance of said class.

This change is depicted in Figure 4.1. Each thread uses a separate Statistics class instance to keep track of changes done by local scheduling operations, here shown as *Task*. Once the simulation concludes, the separated statistics are merged into a central one, which then is used for further evaluation.

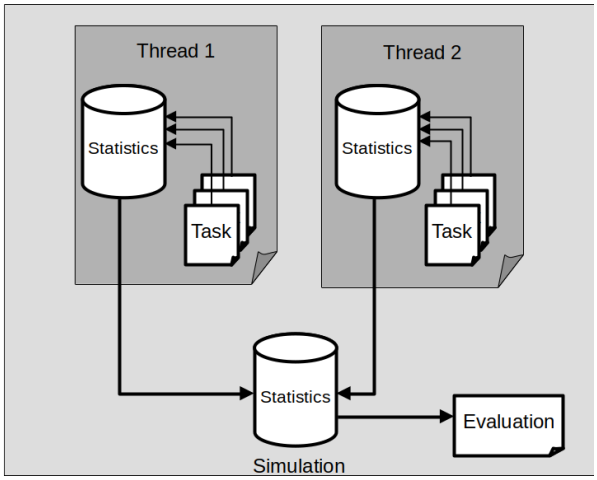


Figure 4.1: Separated Statistics. Each thread uses a separte instance for scheduling (*Tasks*). Once completed, merged into central object for evaluation.

4.1 Implementation

The first major implementation change was the introduction of the `Statistics` class. To realize this, a class was created and all variables that contained statistical information—variables with the `statistics_`-prefix—were migrated into this class as public members. From here, two separate experiments were made: one where the `std::vector` data types were kept the same and another where some of these were exchanged with `std::map` types. The later experiment was conducted with the reasoning that a thread would not visit all available vertices and edges as the graph was split into the previously introduced chunks. Furthermore, some updates of the statistic variables are made when an actual train change occurs. Hence when in a chunk only a part of the vertices observe changes, as some tracks or vertices are not used in the current timetable, the resulting vector would be filled only sparsely. Therefore, to save some memory, maps were used to only save the relevant information on the elements that the thread visits. However, accessing and modifying these maps introduces an overhead as the internal structure is very different. This means, in some circumstances, this model change may not be beneficial.

After running some tests with both experimental versions implemented, it was noted, that the differences for small datasets, i. e. small graph and short timetable were not significant. However, for larger datasets with a lot of trains and a large network, the differences were very pronounced: for example, when comparing both implementations on the *5h/D* dataset, the vector-approach used up *9.3GiB* of memory, whereas the map-approach only used *350-470 MiB* of memory while performing the simulation with 8 threads. This is a difference of about 20% to 27%. While running this was not a problem on the present hardware yet, for an even larger dataset this might quickly become an issue. Furthermore, the runtimes of the vector approach were slightly longer. This was due to a significantly prolonged merging time (0.03s for map, 13s for vector). This is probably due to a non-optimized merge for the vector variant and may be improved. On the other hand, it needs to be noted, that the map variant introduced slightly longer runtimes for the evaluation part. However,

this can also probably be optimized; for example, by converting back the map to a vector based data-structure.

Because of the significantly lower memory usage and no significant changes in runtime, I decided to pursue the map variant in the further implementation.

As the separate instances need to be joined in the end to result in conclusive overall numbers, a class method was implemented that takes care of this task. Said join method works by adding the respective values from one instance to another one.

The statistics instances are generated during the initialization of the simulation. Each thread is then associated with one of those instances to not mix them up. Further, all locations in the code where the former statistics variables were accessed needed to be changed. Therefore, most methods received an additional function parameter to hand over the correct instance/object. Every former access to the variables was then changed to access to the respective variables within the instances. Once the simulation steps concluded, the individual thread objects were merged to a central one with the implemented joining method for further processing and evaluation.

The second change was the introduction of separate locks for all other, non-statistical variables. With this, the individual access can be controlled precisely and threads that want to read or write to two independent variables do not need to wait for each other. The declarations can be seen in Listing 4.1. As the inclusion of entire code segments would exceed the scope and readability of this thesis, only some small exemplary uses are shown in Listing 4.2. It can be seen, that the mutex locks were implemented using `std::shared_mutex` objects, which also allow being used for synchronized read access. That is, the lock can be used in a *multiple-read, single-write* configuration, meaning that before reading a non-blocking lock can be acquired by multiple instances, however, when write access is needed, all other read and write locks need to be returned before the continuation. This ensures that data stays consistent if read and write operations would occur at the same time.

It should be noted that the way the tasks are scheduled has not changed compared to the approach described in Chapter 3.

Listing 4.1: Summary of used shared mutex locks.

```
1 std::shared_mutex mutex_shared_general;     // For general access
2 std::shared_mutex mutex_shared_reqs;        // For request handling
3 std::shared_mutex mutex_shared_caps;        // For capacity calculations
4 std::shared_mutex mutex_shared_blocked;     // For blocking trains
5 std::shared_mutex mutex_shared_at_element; //  For residing trains at element
6 std::shared_mutex mutex_shared_element_to_consider; // For mods on which
      train to consider next
```

Listing 4.2: Unique write lock to specifically reserve a variable for write access.

```
1  {
2    // Get unique access to shared lock for exclusive write
3    std::unique_lock uniqueLock(mutex_shared_at_element);
4
5    // the respective Occupier at the current infrastructure element
6    auto it = std::find(atCurrent.begin(), atCurrent.end(), Occupier(r.type, r.
         trainID, r.j, r.epdt, r.p));
7    assert(it != atCurrent.end());
8
9    // Delete current state of blocker
10   atCurrent.erase(it);
11 }
```

4.2 Analysis

Like in the previous chapter, this revision was tested on the same two devices as before with the same benchmark file. This time, mostly only two runs were made as the results appeared to be already adequately stable. However, similarly as in the previous approach the results from the mobile computer are overall less consistent. We will therefore focus on the results of the desktop PC in favour of compactness and consistency.

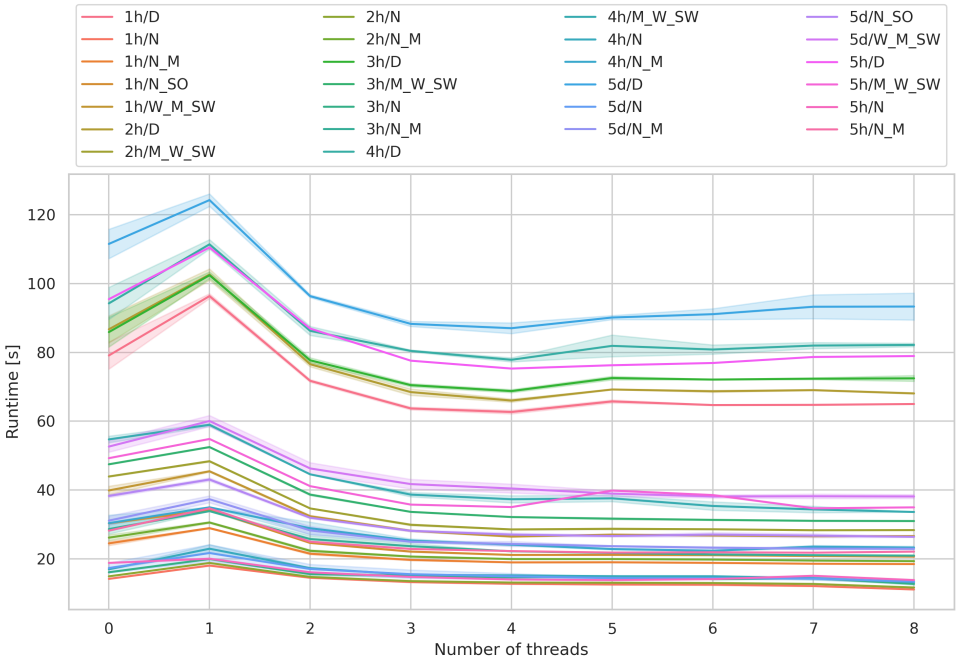
Visual Results. Like it was done in the previous chapter, the data was processed with the Python script using Pandas and Seaborn. The results can be found in Figure 4.2. Like before, a detailed graph of the *5h* datasets can be found in the appendix in Figure A.1b.

When inspecting the results of the desktop PC, one can see a clear but small improvement in runtime with more than two threads. However, the overhead introduced by this approach can be seen by the large bump when one thread is used, compared to the baseline results. It can additionally be noted that the threaded runtimes after the initial overhead are relatively constant and do not change much. This is where the differences of the mobile CPU become clear. While the initial behaviour is similar and also shows some speed-up compared to the baseline results, this advantage quickly deteriorates and with an increasing thread count, so does the overall runtime. This phenomenon is more pronounced with the larger datasets when compared to the smaller ones. These results highlight the differences of the respective CPUs and show that the laptop seems to be strongly limited in multi-threaded performance, probably due to its low power target. Furthermore, there are multiple outliers in the graphs of the laptop. This might be due to temporary power or thermal constraints or other applications that performed some tasks in the background, which impact the performance on the laptop severely.

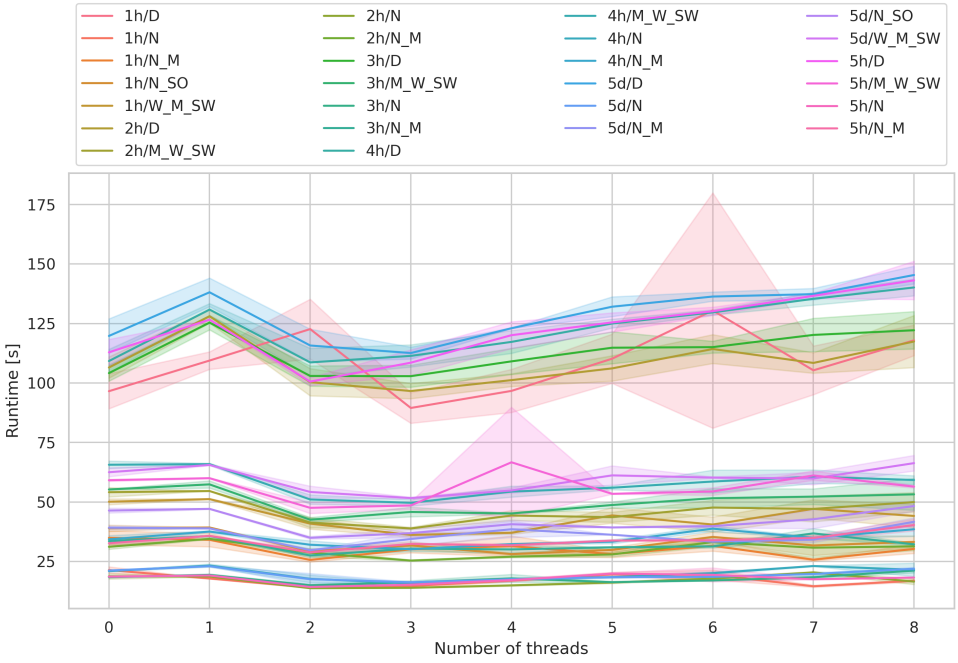
Analytical Results. After the visual representation already provided some initial clues, we want to inspect the speed-up and efficiency of the new approach. As we have already observed in the previous attempt and the graphical analysis above, the laptop computer provides much less stable results. Therefore, we focussed on the results of the desktop PC for which the resulting runtimes can be found in an averaged form in the following Table 4.1 grouped by dataset.

When inspecting the measured runtimes above, the observations of the visual analysis can be confirmed. We see, that in many cases, the run using all available eight threads was the fastest one (see column *Best Thread*) and for every dataset, a threaded variant performed better than the non-threaded baseline. This is especially the case for the datasets with lower train count (as defined by the observed duration, see Table 2.2). This resulted in an overall speed-up.

Speed-Up. As can be seen in Table 4.1 and the visual evaluation, the runtimes of the larger datasets with regard to timetable size and hence train count improved with each new thread until at total of four threads were used. Afterwards, no further significant gain could be observed. For the exemplary speed-up calculation, we want to compare the best result to the baseline of the *1h/D* dataset as done for the previous attempt in order to receive a best-case value for this approach. Hence, we take the



(a) Runtimes on the desktop computer.



(b) Runtimes on the laptop computer.

Figure 4.2: Runtimes of the Separated Statistics approach with various datasets for differing thread counts.

Dataset	Baseline [s]	1-Thread [s]	Fastest [s]	Speed-up	Efficiency	Best Thread
1h/D	79.090	96.341	62.669	1.262	1.218	4
1h/N	14.208	18.076	11.123	1.277	1.272	8
1h/N_M	24.520	28.941	18.506	1.325	1.180	8
1h/N_SO	30.343	34.172	20.654	1.469	1.126	8
1h/W_M_SW	39.889	45.453	26.488	1.506	1.139	4
2h/D	86.690	102.599	65.998	1.314	1.184	4
2h/M_W_SW	43.939	48.37	28.331	1.551	1.101	7
2h/N	14.947	18.828	11.696	1.278	1.260	8
2h/N_M	26.178	30.581	19.354	1.353	1.168	8
3h/D	85.883	102.388	68.753	1.249	1.192	4
3h/M_W_SW	47.485	52.487	31.002	1.532	1.105	8
3h/N	16.133	19.97	12.743	1.266	1.238	8
3h/N_M	28.663	33.879	20.963	1.367	1.182	8
4h/D	94.225	111.371	77.838	1.211	1.182	4
4h/M_W_SW	54.711	58.956	33.635	1.627	1.078	8
4h/N	16.982	22.964	13.254	1.281	1.352	8
4h/N_M	30.395	34.984	22.373	1.359	1.151	6
5d/D	111.492	124.227	87.013	1.281	1.114	4
5d/N	17.357	21.716	13.412	1.294	1.251	8
5d/N_M	31.127	37.338	22.934	1.357	1.200	8
5d/N_SO	38.296	43.066	26.412	1.450	1.125	8
5d/W_M_SW	52.629	60.06	38.137	1.380	1.141	8
5h/D	106.155	122.58	75.286	1.410	1.155	4
5h/M_W_SW	49.263	54.838	34.702	1.420	1.113	7
5h/N	18.839	20.068	13.852	1.360	1.065	5
5h/N_M	28.223	34.687	21.816	1.294	1.229	7
Max	111.492	124.227	87.013	1.627	1.352	8.0
Min	14.208	18.076	11.123	1.211	1.065	4.0
Mean	45.679	53.036	33.805	1.364	1.174	6.615

Table 4.1: Runtimes for the Separated Statistics approach of the simulation on the desktop computer with the given dataset and threads. Runtimes in seconds.

values for the fastest variant of dataset *1h/D* and calculate the speed-up, again with the formula introduced in Definition 2.4.1.

$$
\begin{aligned}
t_B &= 79.090s && \text{(average baseline)} \\
t_4 &= 62.669s && \text{(average runtime with } N = 4 \text{ threads)} \\
S_4 &= t_B / t_N \approx 1.262
\end{aligned}
$$

This indicates that the parallelized code with four threads is 1.26x or 26% faster than the baseline. This result is consistent with the other datasets yielding approximately the same relative numbers. For some smaller datasets, the improvements are noticeably better. Speed-ups of 30-40% were common, one even achieving 67%. When inspecting the overall result, an average speed-up of 1.364 is observed.

While this is already a significant improvement, especially compared to the results from the prototype, in comparison to the theoretically available computation power of four threads this gain is relatively small.

Efficiency. To assess the efficiency of this revision's changes, again the formula of Definition 2.4.2 is used. Like in the evaluation of the speed-up, the results from the desktop will be used due to the more consistent and steady results. For the overall assessment, the inputs *1h/D* and *1h/N* will be used in an exemplary manner, however, the results from the other data sets yield similar numbers as can be seen in the aforementioned Table 4.1. As for the *1h/D* data the efficiency is calculated as

$$
\begin{aligned}
t_B &= 79.090s && \text{(average baseline)} \\
t_1 &= 96.341s && \text{(average of parallel w/ 1 thread)} \\
r &= t_1/t_B \approx 1.218
\end{aligned}
$$

For the *1h/N* set, the result is $r \approx 1.27$, and shows a similar picture. When inspecting the averaged efficiency values from Table 4.1, the overhead introduced ranges from 1.065 to 1.352 with an average of 1.174. This means that the overhead in this approach is slightly worse with about 17% instead of the previously measured 12%. However, this is also a bit expected as the introduction of the Statistics class requires additional handling that was previously not there.

Profiling. Again, the developed approach was tested with the *Perf* profiler and reviewed in *CLion*. For the evaluation, the large *5h/D* dataset was chosen with 8 cores as it represents a large graph with a lot of trains.

From the results, it could be noticed that about 38.3% of the total time was spent locking or unlocking the mutexes. The careful reader might have noticed that this is more than found in the initial approach. While we are not quite sure why that is in particular, it is still a bottleneck and reducing these locking times would potentially result in better runtimes.

Inspecting and looking out for other culprits slowing down the simulation, it was found that the boost library currently seems the only major contributor to have an impact on the runtime besides the aforementioned handling of mutexes. Most of its share comes from iterator dereferencing or map accesses. However, this is nothing we can improve on within the scope of this thesis with the exception of minimizing its usage.

Learnings. During the evaluation of this approach, we have noticed that using a separated class handling the statistic values helped to improve the overall runtime of the simulation. Furthermore, while a vector-based statistics class is simpler, for datasets with both, a large graph and large timetable, this results in a significant increase in memory usage. While this is not an issue for the currently used datasets, this might be one when even larger timetables and networks might be investigated in the future. Here, the limits of consumer hardware might be reached and prosumer or professional hardware would be required to run. Hence, we decided to continue with the map-based approach for now, as it offered us no significant increase in runtime

when compared to its vector-based counterpart, but resulted in significantly lower memory usage.

Furthermore, as a result of the analysis it was found out that due to the changes made, a total speed-up of about 20% to 63% with an average of approximately 35% was achieved depending on device and dataset. While this is already a significant improvement to the initial implementation, there is more potential headroom as the increase is relatively small compared to the theoretically available power. Therefore, further improvements need to be made.

While locking and eventually waiting always introduces some kind of slow down and overhead, it can be seen in this approach's benchmark and profiling results that the overhead is still relatively high. Therefore, it is desirable to reduce as far as possible. This issue will be addressed in the subsequent chapter.

4.3 Summary

In this approach, we investigated the possible advantages of extracting the statistics variables and allowing each thread to modify a local separated copy of them. This was implemented by creating a class containing all statistics-related variables, of which then each thread used one particular instance of. This was possible due to the commutative nature of said variables. All contents are then merged by addition after the termination of the simulation for further evaluation. This implementation change, together with optimized synchronization showed significant improvements to the previous implementation. However, the resulting runtimes still offer much headroom for further improvements.

Chapter 5

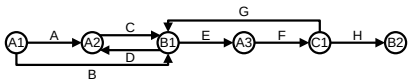
Approach 3: Sub-Graphs

While the previous iteration already improved the runtime by a bit, it strongly depended on the use of mutexes. As the evaluation revealed, despite the distribution to multiple locks this showed to still be a severe bottleneck. A large portion of the runtime was spent handling, i. e. creating, locking and unlocking, these mutexes. However, the synchronization done by the locks was needed as the chunks, i. e. the data each thread worked on had no clear separation. This means that while each thread focussed on a fixed set of vertices and edges, they could potentially be all connected and thus requiring each thread to always know the state of each infrastructure element to read from or write to. To counteract this, a new revised approach was planned to reduce the locking and blocking to a minimum.

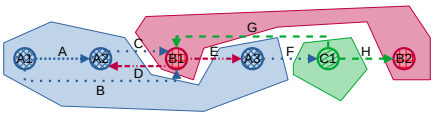
At first, as we noticed that the split of data for the statistics variables helped to divide the data into separated sections and reduce or even eliminate the use of mutexes for this part, a similar approach was tested for the rest of the data. However, while this split was rather uncomplicated for the statistics as the performed operations on said data were commutative, this was not necessarily the case for the rest of the data. Here, also non-commutative operations were performed. This means, that updates for these variables had to be synchronized and sent to all other threads so that each thread was working on the same version. Nevertheless, a prototype was implemented to test the viability of this division approach. However, first tests showed that synchronization needed to be performed very often increasing the overall overhead. This resulted in effectively unusable runtimes even for small datasets (small graph and timetables), with increased runtimes from several seconds to minutes by approximately an order of magnitude.

While with optimizations this overhead probably could have been reduced, I decided to discard this approach. As the synchronization of data sections was the main problem with this attempt, the division of the existing network into logical subsections was prioritized as it reduces the need for common updates and thus promised better maintainability and also results.

For this, the generation of chunks and with it the division of the infrastructure network was re-envisioned. Previously, the vertices and edges were ordered by their internal ID and then divided into equal-sized groups, which then represented the respective chunks. This division did not consider any special logic for example distance metrics between edges. The IDs were generated on the import of the infrastructure and therefore may theoretically address vertices and edges randomly in the network.



(a) Initial exemplary infrastructure graph.



(b) Exemplary infrastructure graph divided into three chunks for vertices and edges each as indicated by the differing colours/markings. Only one inner edge from A1 to A2. All other edges are outer edges of the respective chunk in which the source node resides.

Figure 5.1: Visualization of infrastructure network split into three chunks (A/blue, B/red, C/green).

This is where the new approach comes into play: the division of the vertices and edges shall now be done in logically separated sub-graphs to improve the work order on the one hand and on the other hand reduce the need for synchronization as each thread works on an encapsulated part of the network.

To be precise, the infrastructure graph G' as described in Section 2.2 shall be divided into p sub-graphs $G_i'' = (V_i'', E_i'', \overline{E_i''})$ for $i \in [1,p], V_i'' \subseteq V', E_i'', \overline{E_i''} \subseteq E'$ so that each sub-graph is vertex disjunct, i. e. $V_i'' \cap V_j'' = \emptyset$ for all $i, j \in [1,p], i \neq j$. The definition of the sub-graphs are a little bit different than it is known from the original graph's one: it comprises of two edge sets E_i'' and $\overline{E_i''}$ besides the regular vertex set. The first one defines all *inner* edges that are contained within the sub-graph G_i'', ergo $E_i'' = \{(u,v) \mid u,v \in V_i'', (u,v) \in E', u \neq v\}$ for $i \in [1,p]$. However, as the different sub-graphs are connected and trains need to be able to pass from one sub-graph to another, the outgoing and incoming edges of each partition need to be kept track of. Therefore, $\overline{E_i''}$ defines the *outer* edges of sub-graph G_i'' so that $\overline{E_i''} = \{(u,v) \mid u \in V_i'', v \notin V_i'', u \neq v\}$ for $i \in [1,p]$. That way, trains that pass multiple sub-graphs can be scheduled via the connecting edges. This division has the advantage, that the scheduling of the vertices and inner edges are completely isolated and do not require synchronization. Only changes affecting connecting edges, i. e. incoming or outgoing edges, need to be shared between threads.

The aforementioned chunk division is visualized in Figure 5.1. The raw graph is depicted in Subfigure 5.1a. The divided graph can then be seen in Subfigure 5.1b. In this example there are three chunks: these are coloured blue (A), red (B) and green (C). Each chunk is visualized by the coloured polygon containing the respectively coloured/shaped vertices. The respective inner and outer edges are marked by the respective colour and line shape. In the example only the blue chunk (A) has an inner edge—from A1 to A2, other edges are outer edges of the respective chunk.

To achieve this separation, data available in the respective datasets is used: the data has two intrinsic levels of regional information, the organizational unit region (i. e. North, West, East, South, South-East, South-West and Centre) as well as finer regional area (such as Cologne, Frankfurt, Munich). The latter information is encoded within the operational control points short name. Each short name uniquely identifies the OCP and its location. The first letter(s) identify the area and the rest the OCP within said area. The areas are derived from the former German Railroad Divisions

that were managing the different regions until the transition from a government-owned to private company with the foundation of the Deutsche Bahn AG in 1994. A downloadable list of these *DB Betriebsstellen*, the Operational Control Points, as of 2018 can be found under [8].

With these two levels of regional information the graph can be partitioned into the respective regions. Each region then is processed by a thread. Due to this separation most of the vertices are encapsulated and are independent. Only vertices that have an incoming connection from another region must be aware of the state of the respective connecting edge. If an incoming edge has been successfully scheduled in the previous time step, the vertices can be worked on in parallel without the need for synchronization. Any changes on the incoming edges during the respective time step will be considered either in the next time step or during the scheduling of passing requests at a later point during the current time step. Furthermore, the inner edges only have points of contact within the respective region, i. e. source and target vertices are both in the same region. Therefore, while scheduling the inner edges, no synchronization needs to be done and the only requirement is that the source vertex has been successfully scheduled at the current time. The only other point of synchronization between threads now happens during the scheduling of edges that connect two regions. Here it is required that the chunks, in which the target vertex resides, may not do any other calculations. It shall also be noted that, as each incoming edges is inevitably also an outgoing edge of another vertex, these have to be considered only once. Because the scheduling of edges pulls all available trains from a vertex, we associated it with the region of its source vertex.

The scheduling of passing requests is not changed in this approach and will be done in a non-parallelized manner as before. Therefore, passing requests need to be scheduled to finalize a time step and must be completed prior to the next one, thus marking a potential bottleneck.

Another change in this approach was the general handling of the scheduling within time steps. To recall, formerly each vertex or edge was assigned to one specific chunk which by itself was again associated by one particular thread. Therefore, it was possible that one thread would receive one or multiple small chunks whereas other threads were burdened with large ones. As all threads need to wait for all others to complete the scheduling of infrastructure elements in one time step, this also means a potentially prolonged simulation time due to an unequal load distribution. To reduce the risk of such inequalities and wait times, a task queue with prioritization was developed.

For each chunk that needed to be scheduled within a time step, a task is created and inserted in the queue, then to be prioritized by its current scheduling stage (i. e. scheduling vertices, inner or outer edges). However, as multiple tasks will ultimately be executed in parallel, there need to be some constraints to safeguard the correct order of execution and consistency of the data for some stages.

To be able to schedule the vertices, all other connected chunks, ergo sub-graphs that have an outgoing edge to one or more vertices of the currently inspected chunk, need to be completed in the previous time step. Therefore, all chunks save the last scheduled time for the vertices and outgoing edges. Since time is modelled discretely as before, the previous time is known and can then be checked for. The scheduling of inner edges does not require any constraints except that it is subsequently processed after the vertices have been scheduled. Lastly, the scheduling of outgoing edges requires that the vertices of the connected chunks have already been calculated in the

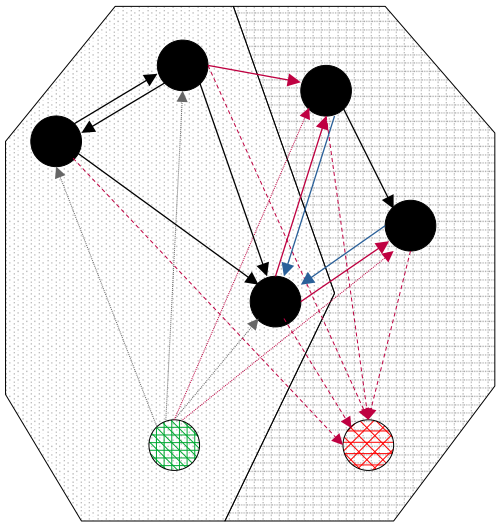


Figure 5.2: Visualization of dependencies between chunks. Chunks are depicted by the underlying polygons, green/red hatched vertices are start/end nodes. Red and blue edges are incoming or outgoing edges of respective chunk requiring synchronization and constraints.

current time step.

This behaviour can be seen in Figure 5.2: there are two chunks as marked by the respective polygons in the background. For the left chunk, all incoming edges are marked in blue, all outgoing ones in red; for the right one it is vice versa. The green and red vertex are the source and target vertices. In this particular example, both chunks depend on each other. This means, that for the left chunk to be able to schedule its vertices, the right chunk needs to have finished scheduling the its outer edges, i. e. the blue ones. When this is the case, the left chunk may then proceed with the scheduling of its vertices, which pulls all possible trains from the respective incoming edges to the vertex. Here, only locking needs to be applied for those edges that connect the two chunks. After scheduling the vertices, the inner edges can be planned without any issues and necessity for locking as all inner edges can only be accessed by the current thread. Lastly, for scheduling the outgoing, and thus connecting edges, i. e. putting trains from a vertex to the respective edge, the right chunk needs to have already finished the scheduling of its vertices. In the example, for the left chunk to schedule its outgoing, red edges, the right must have completed its vertices to free all trains that may depart. Here too, locking needs to be put in place when accessing edge-related data, to ensure non-parallel access.

To formalize this, firstly let the set of possible stages that a task can go through be defined by the ordered set $\mathcal{S} = \{\mathcal{V}, \mathcal{E}_{\text{in}}, \mathcal{E}_{\text{out}}, \mathcal{W}, \mathcal{F}\}$ with the strict order $\mathcal{V} < \mathcal{E}_{\text{in}} < \mathcal{E}_{\text{out}} < \mathcal{W} < \mathcal{F}$. In the first three stages $(\mathcal{V}, \mathcal{E}_{\text{in}}, \mathcal{E}_{\text{out}})$ the vertices, then the inner and lastly the outer edges are scheduled. If the outer edges cannot be scheduled, due to unsatisfied constraints, the task enters the stage $\mathcal{W}$ waiting until the respective constraints are fulfilled and the outer edges can be scheduled. This special waiting task is introduced as the algorithm would otherwise potentially lock up: if a stage

was not possible and is re-inserted as is, it would quickly be picked up again and thus either lock up or at least reduce efficiency. Thus, the task is reinserted with a lower priority state so that other task, that can and potentially need to be processed first can be worked on. Once a task could be completely scheduled, it resides in the finished state $\mathcal{F}$.

Lets recall from the algorithm description in Section 2.2, a time step $t \in \mathbb{T}$ is a discrete point from the time horizon $\mathbb{T} = [T_{\text{MIN}}, T_{\text{MAX}}] \subseteq \mathbb{N}$. Let a chunk G_i'' for $i \in [1, p]$ be identified by a unique identifier. In this case, we use its fixed index i. Then a *SchedulingTask* can be defined as $q = (t, s, c), t \in \mathbb{T}, s \in \mathcal{S}, c \in [1, p]$ and comprises of the time t and stage s which is to be scheduled and lastly the corresponding chunk c as identified by its index.

This task can then be inserted into the priority queue Q with ordering relation $\leq$ so that

$$\mathsf{Q} \subseteq \{(t, c, s) \mid t \in \mathbb{T}, s \in \mathcal{S}, c \in [1, p]\} \text{ and}$$

$$\leq = \{((t, c, s),(t', c', s')) \mid (t < t') \vee (t = t' \wedge s < s') \vee (t = t' \wedge s = s' \wedge c < c')\}.$$

Now that the scheduling via the task queue has been explained, we need to formally define the previously mentioned constraints. For that, we firstly declare $t_{\mathcal{V}}, t_{\mathcal{E}_{\text{out}}} \in (\mathbb{T} \cup \{-\infty\})^p$ which for each chunk represent the last successfully processed time point, or $-\infty$ if none was processed yet, of the vertices or outgoing edges respectively. Then, the vertices (stage $\mathcal{V}$) of a chunk G_i'' with $i \in [1, p]$ can be scheduled in time step $t \in \mathbb{T}$ iff

$$\forall j \in [1, p].\ (t_{\mathcal{E}_{\text{out}},j} = t - 1) \vee (t_{\mathcal{E}_{\text{out}},j} = -\infty),$$

with $i \neq j, (u, v) \in \overline{E_i''}, (u, v) \in \overline{E_j''}, u \in V_j'', v \in V_i'', t_{\mathcal{E}_{\text{out}},j} \in (\mathbb{T} \cup -\infty)$.

Similarly, for the scheduling of an outer edge $e = (u, v)$ in chunk G_i'' with $i \in [1, p]$ at time $t \in \mathbb{T}$, the following constraint must hold to be schedulable:

$$\forall j \in [1,p].\ t_{\mathcal{V},j} = t,$$

with $i \neq j, u \in V_i'', v \in V_j'', (u, v) \in \overline{E_i''}, (u, v) \in \overline{E_j''}, t_{\mathcal{V},j} \in (\mathbb{T} \cup -\infty)$.

5.1 Implementation

The main change in this iteration consists of the division of the network into the aforementioned independent regional chunks. Furthermore, the scheduling process was modified to allow for a better balanced workload of scheduling tasks. This was achieved in two parts: implementation of a task queue handling the different steps and assuring that those get executed correctly and secondly the optimization of the chunking algorithm.

To start, we want to explain the changes in the network division. During the initialization of the simulation, it is now checked to which region a vertex belongs prior assigning it to the respective chunk. As we have seen above, there are two possible levels of detail on whose basis this region assignment can be made, i. e. how granular the infrastructure shall be divided. Thus, this regional level can be selected via a newly introduced `GranularityLevel` parameter which can be provided to the simulation. It accepts level zero (0) and level (1) where level 0 is the coarse selection by organization unit and level 1 is the finer division by lower region. This information can be inferred from the infrastructure data: the organizational unit is saved as a

vertex property and thus can easily be accessed. The region can be inferred from the vertex name property: the first letter indicates the regional area to which it belongs to. The following letters identify the operational control point in particular and its location in more detail. For most of the data, this OCP code follows the pattern (without hyphens) R-L(LL)-(T), where R is the region, L is the locality, identified by one to three letters. If there is special type of locality, such as freight stations, this is sometimes marked with one additional character T at the end. For instance, *KA* for Aachen Central Station is region *Cologne (KA)* and locality *Aachen Central Station (KA)*. However, localities within the same city often are associated with the next largest locality by using more descriptive letters in the locality code. For example, *Aachen West Station* a smaller station near the Aachen Central Station has the code *KAW* (region *Cologne* KAW, city *Aachen* KAW, locality *West Station* KAW).

While this would theoretically allow for an even finer division, these locality codes are often unreliable, incomplete or inconsistent. Furthermore, not every larger city has the locality code defined solely by the second character but also by the third as well, resulting in hard or even impossible to separate locations without more information. For example, *KAU* identifies *Au (Sieg) Station* (region *Cologne* KAU, city/locality *Au (Sieg)* KAU), a completely different city on the other side of Cologne. Both Aachen and Au share the second letter A. But for the separation, one could not tell if the U describes a locality near Aachen as above or a completely other city as it does here. Therefore, this was omitted from the actual implementation.

The definition of the levels is modelled as an `enum`, and is shown in the Listing 5.1. With regard to the implementation, the following can be noted: initially the linkage between infrastructure elements and the respective chunk was done with the implementation of maps. However, as the access to this map occurred very frequently the extended lookup times accumulated and affected the runtime noticeably. Therefore, the implementation was adjusted to vectors which have much better access times as maps. Some maps were used as helpers which however were seldom used and thus affected the runtime only marginally.

Listing 5.1: Granularity Levels

```
1 // Granularity for chunk gen
2 enum GranularityLevel {
3   ORGA_UNIT, // Based on organization unit
4   REGION,    // Based on region (first letter if node id)
5 };
```

To handily save all information regarding a chunk, a `Chunk` class was introduced. It handled the properties of a chunk, such as name, ID, vertices and inner/outer edges. Furthermore, some information about the simulation state was saved like the last time step in which the vertices were successfully scheduled.

Besides the revised division of the infrastructure the scheduling algorithm was adjusted, too. A `std::priority_queue`[1] for `SchedulingTasks` was used to implement a *TaskQueue*, holding all scheduling steps as so-called *SchedulingTask*s. A `SchedulingTask` contains information about the chunk that should be processed, the current state it is in, e. g. scheduling of vertices and lastly the time step it schedules. These scheduling states are also modelled as an `enum` and can be found in Listing 5.2. The tasks for different chunks are placed in a priority queue, which

[1] `https://en.cppreference.com/w/cpp/container/priority_queue`, last accessed on 20. April 2021

prioritizes the element according to their current state and the time step; the lower the time and the lower the status (as indicated by the enumeration index), the higher the priority. The threads then take tasks from this queue to work on, an try to process them. If a task could temporarily not be processed, due to yet unfulfilled constraints, it is ignored, enters a specialized waiting state with lower prioritization and is placed back into the queue to be re-tried at a later point in time. This switch to a designated waiting state with lower priority allows the dependents to be processed first, before a retry takes place. As all tasks are guaranteed to finish, also all depending tasks will finish at some point. The dependencies of each scheduling task are in such a way, that they can be resolved to an execution order which is solveable. That is, either a task is dependant on a task with an earlier point in time or it depends on a task at the same time point. In the later case, if it the dependency is unresolved, the dependee is re-scheduled with a lower priority stage as the dependant. Thus, by the ordering relation $\leq$ as described above, there is always an execution order where the dependants are scheduled prior to the dependees. A deadlock thus cannot occur. Furthermore, chunks are not assigned to fixed threads but can now be worked on any available thread depending on the current workload.

Lastly, it is to mention, that the passing requests are calculated centrally after all other SchedulingTasks have been completed for the current time step. This is still necessary as the time steps are performed in discrete steps and thus require special handling. While it would also be possible to reschedule the necessary chunks in the same time step, the previous implementation was more efficient as it only updated the necessary vertices or edges. This is not possible using the SchedulingTasks with the current implementation.

This approach continues to use the separated statistics approach from the previous iteration presented in Chapter 4. While the previous implementation benefited from the strict association of vertices/edges to chunks which were assigned to fixed threads, the map-based implementation still held up with regard to speed and memory usage as the train timetables are relatively sparse. However, more memory (approx. 500MiB to 1GiB) was used with this implementation as threads will possibly work on multiple chunks over the time. Nevertheless, this new approach changes the general scheduling behaviour significantly and can thus be seen as alternative rather than extension of the previous iterations.

Listing 5.2: Scheduling States

```
1 enum scheduling_states {
2   QUEUED_INITIAL,  // Waiting to schedule Vertices
3   SCHEDULED_VERTICES, // Waiting to schedule inner Edges
4   SCHEDULED_INNER_EDGES, // Waiting to schedule outer Edges
5   QUEUED_WAITING_FOR_CONNECTING, // Waiting for constraints of out edges
6   FINISHED
7 };
```

5.2 Analysis

This time, the approach was only evaluated on the desktop computer. As the laptop results have been inconsistent in the previous iterations, it was omitted in favour of time and power consumption as a complete test run would take several hours. If not stated otherwise, the results are shown for GranularityLevel `ORGA_UNIT`. The *5h*

datasets are depicted in detail in the appendix in Figure A.1c.

Visual Results. The results have been visualized as before and can be seen in a the graph of Figure 5.3.

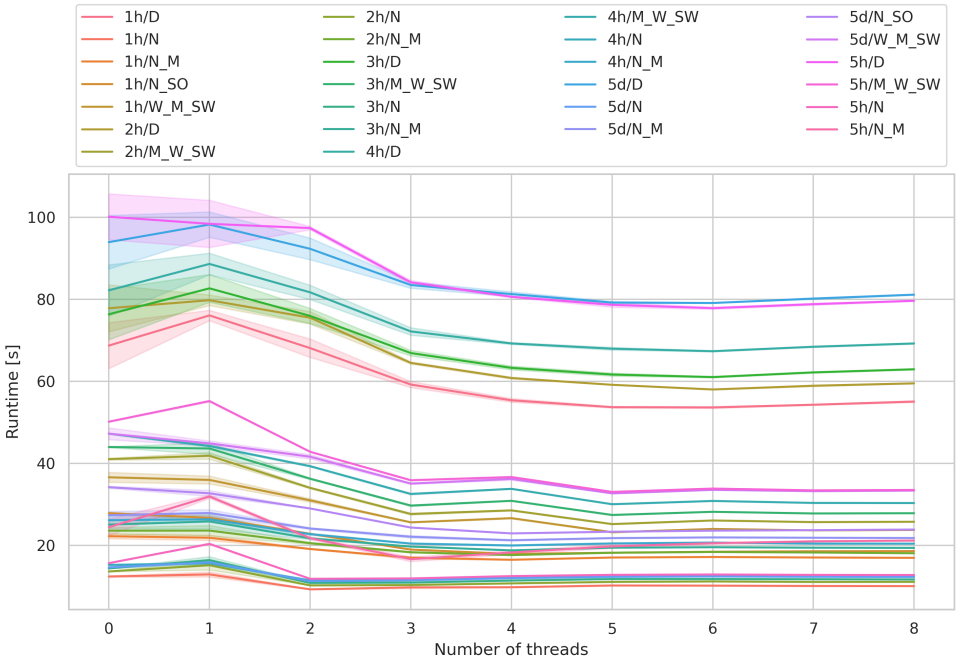
It can also be seen, that the runtimes for the variants using three or more threads are extremely consistent. The variation as indicated by the coloured confidence intervals is very narrow. This indicates a relatively stable simulation. However, when inspecting runs with two or less threads and especially the baseline approach, the variation is relatively high when compared to the rest of the runs. This behaviour is especially pronounced in the longer runs, i. e. for large infrastructure network and large timetable. This behaviour was confirmed after re-running the tests at a later time. Therefore, a temporary issue with the device can be ruled out. The cause for this could also not be found and thus this behaviour is unexplainable at the moment. Similarly, the strong increase in runtime for the *5h/D* dataset using the *Region* level and utilizing five or six threads is significant. Also, for this dataset and GranularityLevel combination, the variances are much higher than it is obeserved for other dataset-thread-level combinations. This behaviour also persisted after a re-run.

It can further be seen that the efficiency, i. e. the performance of the threaded variant using only one thread and the non-threaded baseline, is significantly worse for large datasets (large infrastructure and timetables) using the GranularityLevel *Region* instead of *OrgaUnit*.

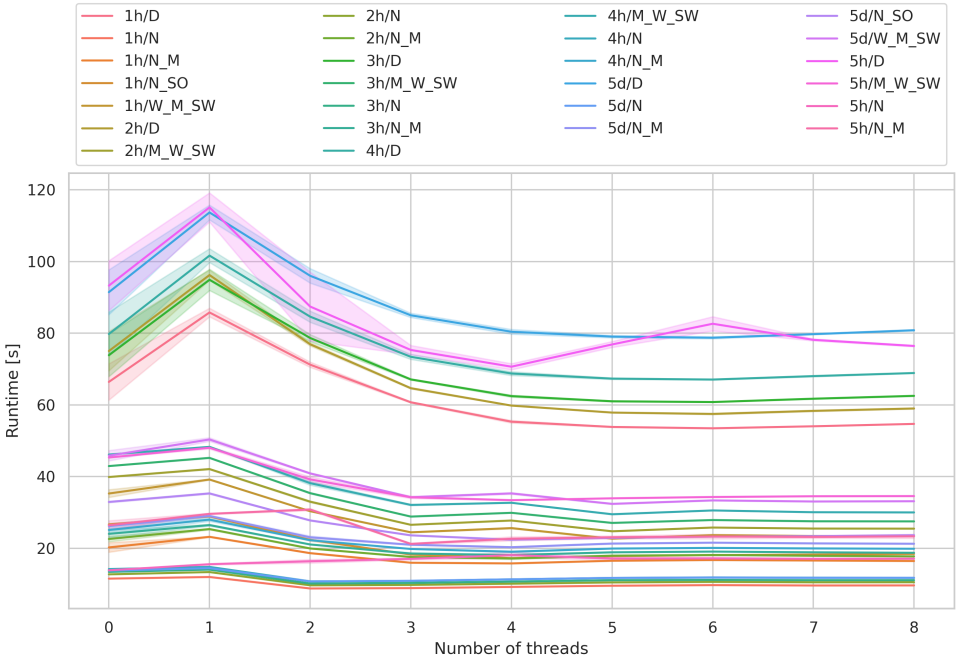
Nevertheless, this approach resulted in a noticeable speed-up compared to the baseline results. Similarly to the previous approach, the results showed no worse decrease in speed-up with increased thread count. However, when considering the *5h/D* dataset, a spike in runtime when using more than four threads can be observed. This is not present in the other dataset runs. This dataset was run again at a later point in time to rule out an overloaded system, but the pronounced curve persisted. As for why this might be, is currently not clear.

Analytical Results. The runtimes observed during the benchmark run over all available datasets were noted. The averaged times, as calculated previously, can be fund in Table 5.1.

From the overall results, it can be noted that a speed-up to the baseline results were achieved. However, in comparison to the previous attempt from Chapter 4, the fastest runs were not the ones with the highest thread count but often ranged from four to six, with an average of 4.1 for the coarse GranularityLevel *OrgaUnit* and 4.0 for the finer *Region* level. This already indicates to some limitations. Nevertheless, the overall runtimes and speed-ups are very similar to those of the previous iteration. Therefore, the speed-up and efficiency shall be examined in the following.



(a) Runtimes for GranularityLevel OrgaUnit.



(b) Runtimes for GranularityLevel Region.

Figure 5.3: Runtimes of sub-graph approach on the desktop with various datasets for differing thread counts.

	GranularityLevel OrgaUnit						GranularityLevel Region					
Dataset	Baseline [s]	1-Thread [s]	Fastest [s]	Speed-up	Efficiency	Best Thread	Baseline [s]	1-Thread [s]	Fastest [s]	Speed-up	Efficiency	Best Thread
1h/D	68.739	76.09	53.658	1.281	1.107	6	66.429	85.80	53.491	1.242	1.292	6
1h/N	12.452	12.964	9.317	1.336	1.041	2	11.553	12.011	8.805	1.312	1.040	2
1h/N_M	22.272	21.899	16.526	1.348	0.983	4	20.258	23.215	15.799	1.282	1.146	4
1h/N_SO	27.89	26.748	17.952	1.554	0.959	4	26.677	28.892	17.281	1.544	1.083	4
1h/W_M_SW	36.632	35.987	23.241	1.576	0.982	5	35.299	39.201	22.690	1.556	1.111	5
2h/D	77.838	79.796	58.04	1.341	1.025	6	75.042	96.237	57.494	1.305	1.282	6
2h/M_W_SW	41.061	41.883	25.226	1.628	1.02	5	39.899	42.117	24.764	1.611	1.056	5
2h/N	13.678	15.2	10.284	1.33	1.111	2	12.751	13.386	9.686	1.316	1.050	2
2h/N_M	23.625	23.643	17.678	1.336	1.001	4	22.617	25.373	17.163	1.318	1.122	4
3h/D	76.364	82.69	61.057	1.251	1.083	6	73.881	94.914	60.812	1.215	1.285	6
3h/M_W_SW	44.013	43.649	27.455	1.603	0.992	5	42.951	45.239	27.098	1.585	1.053	5
3h/N	14.423	16.449	10.882	1.325	1.14	2	13.435	14.075	10.180	1.320	1.048	2
3h/N_M	25.102	25.869	18.82	1.334	1.031	4	24.048	26.491	18.140	1.326	1.102	4
4h/D	82.2	88.639	67.362	1.22	1.078	6	79.792	101.666	67.087	1.189	1.274	6
4h/M_W_SW	47.27	44.268	30.07	1.572	0.936	5	46.165	48.321	29.513	1.564	1.047	5
4h/N	15.279	15.671	11.508	1.328	1.026	2	14.225	14.827	10.799	1.317	1.042	2
4h/N_M	26.165	26.389	20.018	1.307	1.009	4	25.128	28.008	19.052	1.319	1.115	4
5d/D	93.95	98.242	79.115	1.188	1.046	6	91.46	113.701	78.755	1.161	1.243	6
5d/N	14.665	15.925	11.312	1.296	1.086	2	13.783	14.469	10.777	1.279	1.050	2
5d/N_M	27.389	27.925	21.288	1.287	1.02	4	26.2	29.028	20.324	1.289	1.108	4
5d/N_SO	34.219	32.735	22.958	1.49	0.957	4	32.939	35.324	22.417	1.469	1.072	4
5d/W_M_SW	47.267	44.897	32.716	1.445	0.95	5	45.854	50.37	32.409	1.415	1.098	5
5h/D	100.138	98.414	77.862	1.286	0.983	6	93.284	115.106	70.674	1.320	1.234	4
5h/M_W_SW	50.184	55.208	33.075	1.517	1.1	5	45.338	48.067	33.437	1.356	1.060	4
5h/N	15.707	20.403	11.892	1.321	1.299	2	13.887	15.571	15.571	0.892	1.121	1
5h/N_M	24.469	31.953	16.671	1.468	1.306	3	26.443	29.622	21.249	1.244	1.120	3
Max	100.138	98.414	79.115	1.628	1.306	6.0	93.284	115.106	78.755	1.611	1.292	6.0
Min	12.452	12.964	9.317	1.188	0.936	2.0	11.553	12.011	8.805	0.892	1.040	1.0
Mean	40.884	42.444	30.23	1.383	1.049	4.192	39.205	45.809	29.826	1.336	1.125	4.038

Table 5.1: Results from the benchmarks run on the desktop computer with GranularityLevel OrgaUnit on the left and region on the right. *Baseline, 1-Thread, Fastest* in seconds.

Speed-Up. The speed-up for each dataset has, again, been calculated for the best average non-baseline run. The resulting numbers can be found in column *Speed-Up* in Table 5.1. Like in the previous iterations, we investigate the value for the *1h/D* dataset for the coarse GranularityLevel *OrgaUnit* as example below.

$$
\begin{aligned}
t_B &= 68.7s && \text{(average baseline)} \\
t_6 &= 53{,}7s && \text{(average runtime with } N = 6 \text{ threads)} \\
S_6 &= t_B/t_6 \approx 1.28
\end{aligned}
$$

The resulting speed-up value is, with a value of 1.28, only marginally higher as recorded previously. Similarly, the values from other datasets are in line with this result. Consulting the table, it can be seen that the typical speed-up ranges from about 1.188 (GL *OrgaUnit*) / 0.892 (GL *Region*) to approximately 1.6 for both GranularityLevels (GL). While the runs measuring the slowest for the new implementation are mostly observed in the larger datasets (longer timetables and larger networks) and the faster runs in the smaller ones. However, it can be seen that the smaller networks regardless of timetable size overall did not profit much from the additional threads as their best thread ranged between two to four threads whereas the larger networks, also regardless of timetable size, performed best with four to six threads. With an average speed-up of 1.38 this approach only performed marginally better than the previous attempt.

However, when comparing the baseline values to those of the previous attempts, it can be seen that in general the algorithms ran noticeably faster: approx. 69 seconds versus 79 seconds and 86 seconds for this particular dataset. So, while the algorithm did not gain any significant improvements in parallel runtime, the overall performance seemed significantly better to the initial one. Which is interesting as the algorithm in its baseline configuration was not changed much with the exception of only some small adjustments for the statistics class and optimizations.

Efficiency. Like for the speed-up, the efficiency has been calculated in Table 5.1 on the basis of the average runtime with only one thread compared to the non-threaded baseline. As for the example results for the *1h/D* dataset:

$$
\begin{aligned}
t_B &= 68.7s && \text{(average baseline)} \\
t_1 &= 76.0s && \text{(average of parallel w/ 1 thread)} \\
r &= t_1/t_B \approx 1.11
\end{aligned}
$$

In the table above, it can be seen that the efficiency ranges from 0.936 to 1.299 with a mean value of 1.039 for GranularityLevel *OrgaUnit*. This indicates that this approach fared reasonably well when comparing the threaded variants against the baseline. The overhead induced by the parallelization of the simulation depends on the dataset but mostly is only within 5-10 %. In a number of cases the metric was smaller than one, indicating that the application ran faster in the threaded variant

and thus having "negative" overhead.[1] Overall, an average value of around 1.04 is very reasonable good hinting at no significant overhead, especially considering the theoretically more computation intensive chunk division.

The results using the GranularityLevel *Region*, showed similar numbers. However, due to the higher number of chunks the management overhead seemed to weaken the efficiency. The efficiency values for the GL *Region* ranged from 1.3 to 1.04 with an average of 1.13 and thus indicate a significantly higher management overhead compared to the coarser GranularityLevel.

Profiling. After analysing the profiler results for the *5h/D* data, the changes showed effect. As most of the mutex locks could be removed due to the data separation, the time needed for locking logically reduced to only 3.4%. This is a reduction of about one order of magnitude. As this reduction only partially could be observed in a speed-up, it appears that the major bottleneck in this implementation seems to be due to the sequential calculation of time.

The introduction of the Chunk class and handling of such, took about 1.3% of the runtime. The handling of the task queue added approximately another 1%.

Most of the other time consuming operations were obviously still introduced by `boost` calls such as iterator advancements and graph access.

As mentioned in the approach description, the simulation with the sub-graph implementation used more RAM than the previous iteration: the run on the *5h/D* dataset showed a typical memory usage of 500 MiB to 1GiB. While this is a bit more, this still a very acceptable value for this large dataset. Therefore, it seems viable to continue to use a map-approach.

Learnings. The algorithms was modified in this chapter slightly to accommodate for the changes regarding the advanced data division and the TaskQueue. These changes did introduce some overhead, however it appears as if it cancels itself out with the performance gained in general. We noticed that the simulation did not perform much better when compared to the baseline in this approach but increased overall performance when compared to previous iterations. Furthermore, in contrast to the previous approach, the runtimes of the threaded variants performed similarly with increased thread count and did not loose efficiency after a certain point.

5.3 Summary

In this chapter, a refined approach with optimized chunk assignment and processing was presented. While the implementation resulted in slightly faster processing times, the algorithm still does not run as fast and efficient as it could. The division into smaller chunks by defining different GranularityLevel, showed only few differences in processing speed. The profiling results did not show any large proportion of waiting times for locks or variable assignment. This leads to the conclusion that possibly the sequential processing of the simulation with regard to the stages and time steps is the next larger bottleneck. However, the parallelization of time steps comes with some issues: further constraints need to be put in place to assure correct order of processing.

[1] To recall, the baseline used the original scheduling mechanism while the variant using one thread used the newly implementation with the TaskQueue. Therefore, it is possible that the simulation runs faster in the new implementation as in the original one.

Due to the fact that currently the passing requests need to processed at the end of each time step and prior to starting the next, with the current implementation this is not possible. Therefore, the project needs to be restructured slightly.

Chapter 6

Approach 4: Parallel Time Processing

In the previous chapter, we improved the algorithm by making the processing of chunks more versatile and further optimized the subdivision of the infrastructure network. These changes decreased the overall runtime and also improved the efficiency. However, the evaluation revealed further headroom with regard to efficient usage of threads. Therefore, in this chapter, we will investigate how far the relaxation of sequential time execution and discretization in conjunction with the previously presented changes may help in this regard.

The goal of this approach is to resolve the sequential simulation of time steps in such a way that, whenever possible, the next time step is simulated without waiting for all other calculations to finish. To allow for time steps to be simulated in parallel, firstly the behaviour of the *passing requests* needed to be adjusted. To recall the issue, the passing trains—passing multiple infrastructure elements in one discrete time step—currently block the next time step from being simulated. To solve this, the time is modelled as a decimal, or double-precision floating-point to be precise, instead of integer values.

Therefore, we define a new time horizon for the decimal time: let $\mathbb{T}'$ be the time horizon for floating point number so that

$$\mathbb{T} = \{t \mid T_{\text{MIN}} \leq t \leq T_{\text{MAX}}, t \in \mathbb{R}_{\geq 0}\} \subseteq \mathbb{R}^{\geq 0}$$

where T_{MIN} and T_{MAX} are the lower and upper bounds of the simulation, as known from before.

The original inputs were not modified so that these can be used further. Thus, all trains that have nodes with the same departure time as the arrival time, need to be handled and adjusted within the algorithm so that these don't get stuck or only moved one element per scheduling event. Thus the algorithm has been adjusted so that passing requests are recognized on the fly and will be scheduled at the technically nearest future point in time. As computers have a finite precision for decimals, there are finitely many numbers that can be modelled. The difference between these double-precision floating-point numbers is thus used to go to the technically next possible time step.

The way how the simulation processes each chunk does not need to change much except for that it need to account for the time parallel execution. To recall: for each

time step that a chunk needs to be scheduled, a SchedulingTask is created starting at stage $\mathcal{V}$ indicating that the vertices are to be scheduled next, then followed by $\mathcal{E}_{\text{inner}}$ and $\mathcal{E}_{\text{outer}}$ for scheduling the inner and outer edges of the respective chunk. If the outer edges cannot be scheduled, then the simulation enters a lower priority waiting state $\mathcal{W}$. Each thread then works on one of these tasks.

The introduction of non-integer time steps and the possible parallel execution furthermore introduce more issues: firstly, the conditions for when the vertices and outer edges, i. e. the edges connecting two chunks, can be computed, change.

To reliably tell, when the vertices of a sub-graph can be scheduled at a given point in time, it is now important to know when the last relevant changes occur and when only changes are made that can not effect the current scheduling anymore. That is, it typically takes a certain amount of time to traverse a route section. For example, when a train enters an edge from vertex u to v at time $t_1 = 1$ and the minimal time to traverse the edge (u, v) is $\delta t(u, v) = 5$ then the arrival time t_2 of the train at the next vertex must be $t_2 \geq t_1 + \delta t = 6$. That, means, when scheduling a vertex, all trains having a later arrival time than the current one, will not arrive on time at v and are thus not relevant to v's scheduling. If no trains pass along the given edge, then there is no need for waiting.

Therefore, for all edges E'' of G'' we define the minimal travel time as $\delta t : E'' \to \mathbb{R}^{\geq 0} \cup \{\infty\}$ as the fastest time a train with ID id has travelled across this edge according its timed path π_{id} in the given timetable. If no train passes along the edge (u, v) then $\delta(u,v) = \infty$. We can then define the vertex scheduling constraint more generally: a chunk G''_i for $i \in [1, p]$ can schedule its vertices at time $t \in \mathbb{T}'$, iff:

$$\forall j \in [1, p].\ t_{\text{out},j} + \min_{(u,v) \in \overline{E''_j}} (\delta t(u,v)) \geq t \vee \delta t(u, v) = \infty$$

where $i \neq j, (u, v) \in \overline{E''_i}, (u, v) \in \overline{E''_j}, u \in V''_j, v \in V''_i$ and $t_{\text{out}} \in (\mathbb{T}' \cup \{-\infty\})^p$ is the last successful processed point in time for each of the p chunks.

The scheduling of a chunk's inner edges still does not require any checks, as all connected vertices are within the respective chunk and are inherently already processed in sequence.

When considering the processing of the outer edges, i. e. the edges that connect two chunks, further conditions apply: if an outgoing edge is scheduled, the target vertex must already be scheduled in the current time step. If that is not the case, then the worst case, a train would get delayed due to insufficient capacity on the track despite a potential free at the target node. Hence, similarly to the conditions for the scheduling of vertices described above, the target vertex $v \in V''$ of the edge $(u, v) \in E''$ needs to have cleared all occupiers that could reach the target at time t. To realize this, each chunk now keeps track of all the potential times in which an update needs to be done. This is done in form of an ordered list $C \subseteq \mathbb{T}'^p$ that contains all future update times ordered in ascending order of its elements. The list C_j gets updated, whenever the respective chunk j is scheduled. All newly found times are added and the current time is removed.

This list can now be used to formalize the constraint for the scheduling of outer edges: when edge $(u, v) \in E''$ is to be scheduled at time $t \in \mathbb{T}'$, the chunk $j \in [1, p]$ the target vertex $v \in V''$ resides in, needs to have been updated its vertices at latest point in time prior to t. In other words, as long as the next time to consider is prior to t, the edge may not yet be scheduled. Formally: the outer edges of chunk G''_i with

$i \in [1, p]$ may be scheduled iff:

$$\forall j \in [1, p].\ \min(C_j) \geq t$$

where $i \neq j, (u, v) \in \overline{E_i''}, (u, v) \in \overline{E_j''}, u \in V_i'', v \in V_j''$.

6.1 Implementation

As for the implementation, the changes were based on the previous iteration as presented in Chapter 5.

To begin, the data type of `Time` was changed from `integer` to `double`. This involved only a few changes within the code. Most of the logic resulting from this relaxation were handled by the changes in the algorithm.

While it appears by the approach described above as if only a few things need to be changed, in practice this was not the case. Due to the relaxation and the time parallel simulation, a lot of changes needed to be made implementation-wise.

First, during the initialization phase of the algorithm, the minimum travel distance per edge gets calculated. This is done by inspecting all trains in the timetable and for each edge transition, the travel distance is noted and the minimum from all trains that pass this edge is then saved in a `std::map minIncomingTravelTime` which mapped the chunk id of all connected chunks to its minimal travel to the current one. Furthermore, the list of time points to consider was implemented in the chunk object as a `std::set<Time>` property.

With this information, the constraints ultimately could be adjusted: first, to the scheduling of vertices. As described above, the vertices of a chunk can be scheduled if no relevant changes may occur at the given time. However, some smaller other constraints were needed to correctly schedule the vertices. A chunk's vertices can be scheduled at time t if one of the following constraints is fulfilled:

- the chunks are physically connected but no trains are passing on the connecting edges in the current configuration. In this case, the *minimal travel distance* δt for *all* connecting edges is *infinite*.
- the source chunk has been processed at time $t_{\text{out},j} < t$ added to the absolute minimal travel distance $\min_{(u,v)\in\overline{E_j''}} \delta t(u, v)$ for all connecting edges is equal or later than the current time step $(t_{\text{out},j} + \min_{(u,v)\in\overline{E_j''}} (\delta t(u,v)) \geq t)$
- the source chunk has been processed at all time points prior to t and the next time to consider t' is in the future $(t' > t)$, if it one exits.

As for the outer edges, the only constraint that needs to be fulfilled is that all time steps prior to the currently inspected one have been finished.

Furthermore, a small change to the scheduling was done. Instead of inserting the tasks for the next time step once the queue is empty and all tasks have finished, this is now done dynamically. Once a thread finishes and tries to find new tasks. If no new tasks are available, it tries to generate new tasks from the list of which times to consider. For that, for each chunk the next point later than the last worked on is selected and inserted into the queue. If this is not possible, for example as no new time steps to consider are known, the thread enters one of the two following states: it either goes to a sleep state to spare resources if other threads are still working and

may find further times to be considered. Alternatively, it terminates when all other threads are not actively working anymore as no new time points can be found and the simulation is finished. In the former case, once other threads then finish their work, they send a notification for the sleeping threads to wake up and try again.

Furthermore, the algorithm has been modified to accommodate passing requests. Passing requests have previously locked up the simulation in such a way that the next time step could only be calculated once these special cases had been addressed. Now that we have (nearly) continuous-time between two integer time points, passing requests will be scheduled in the next smallest time step. This was realized, by inserting task to the queue at $t+$ `std::numeric_limits<Time>::epsilon()` which represents the difference between 1.0 and the next larger double-precision floating-point number representation. That is, a small amount of time is considered to have passed which in turn means that passing requests are scheduled in a separate time step and thus do not completely block the scheduling any more.

As a closing remark, it shall be mentioned, that the introduced changes, especially the relaxation of time from integers to floating point, required also changes to the non-threaded baseline. Due to the time constraints of this thesis, it was not possible for me to implement a dedicated non-threaded baseline variant. If started with the flag `-t 0`, i. e. the baseline, a variant with one thread is used. Furthermore, instead further optimizations beyond a working implementation were not possible.

6.2 Analysis

The implementation for time parallelism was tested on the desktop computer for all of the given datasets for thread counts from one to eight. The baseline could not be tested in the current implementation as indicated above. Tests with the laptop were again omitted. If not stated otherwise, the results are shown for GranularityLevel `ORGA_UNIT` (see Chapter 5). For a detailed graph of the *5h* datasets, refer to Figure A.1d in the appendix.

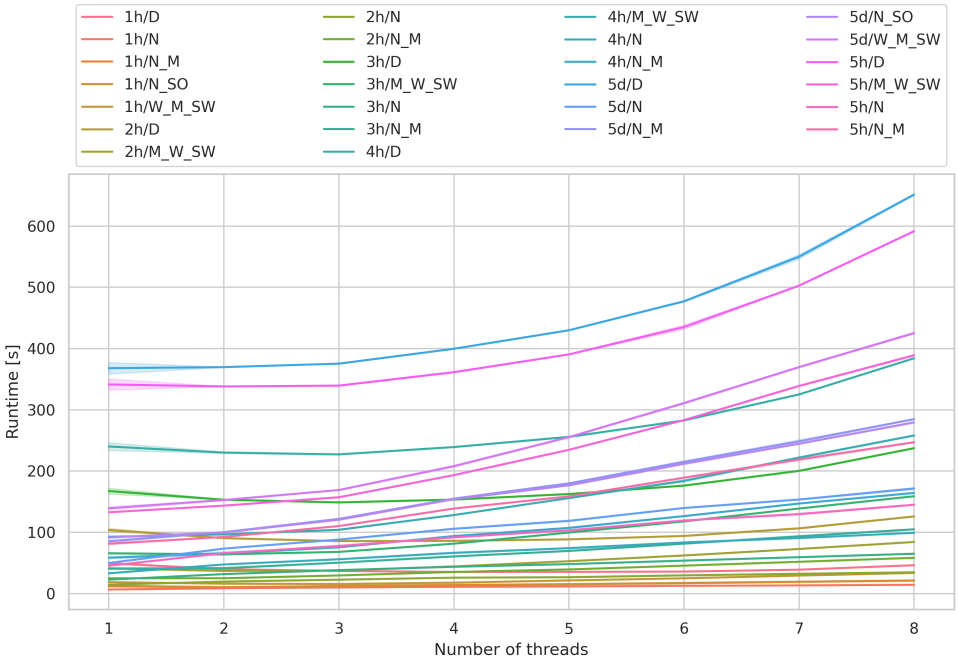
Visual Results. Like in the previous chapters, the test results were visualized through Python, Seaborn and Pandas. The results can be found in Figure 6.1.

It can be seen that for increasing thread count the runtime follows exponential growth for large datasets with regard to network and timetable size. For smaller datasets, the increase is not as pronounced but still clearly visible. As the baseline variant could not be tested due to timely constraints, a direct comparison is not possible. However, only a slightly better runtime is observable with two to three threads.

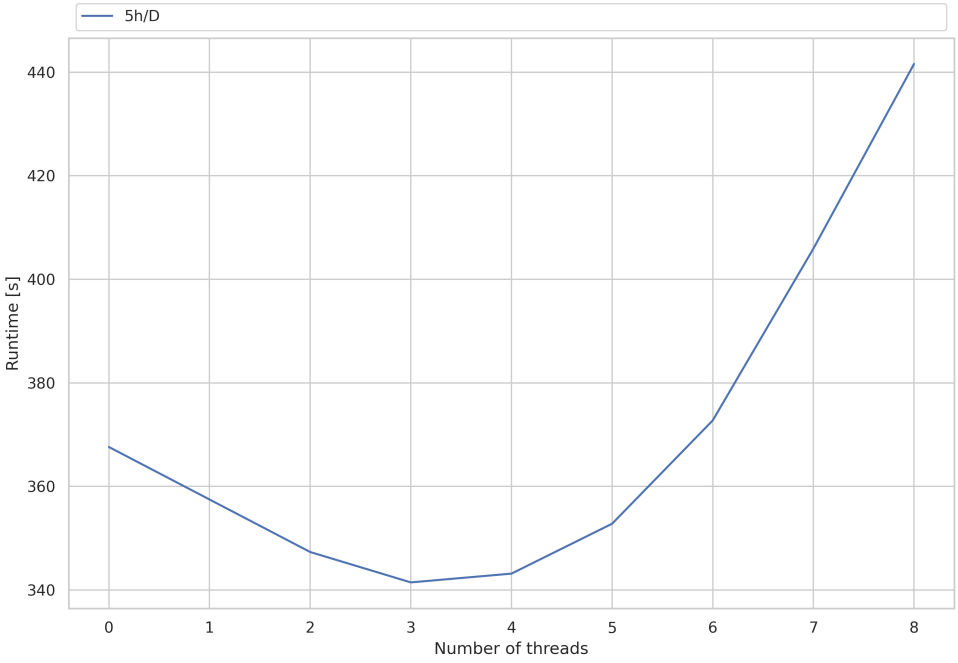
Nevertheless, it can clearly be seen that the results are far from optimal. In the current form, this implementation is not practical and needs further refinements to provide a benefit.

Also, when comparing the results with a finer GranularityLevel, as can be seen in Figure 6.1b, some improvements can be seen towards the three to four threads mark. However, despite the improvements, if more threads are used, these gains deteriorate quickly.

Analytical Results. To investigate the runtime behaviour, the results have been noted in Table 6.1.



(a) Runtimes on desktop for various datasets with GranularityLevel *OrgaUnit*.



(b) Runtimes on desktop for *5h/D* with GranularityLevel *Region*.

Figure 6.1: Runtimes of parallel time approach with various datasets for differing thread counts.

It can clearly be seen that this approach did not fare well in its current implementation. The best runs were mostly performed when run with one thread but never better than four threads. This shows, that the algorithm in most cases did not profit from any multi-threading at all.

Not only does it not benefit well with more threads, but in some cases also performs severely worse when compared to the previous implementation (see Table 5.1). Comparing the *1h/D* data: this time the best run took 338 seconds whereas the previous attempt finished in 77.8 seconds.

However, in other cases the new approach is much quicker: for example, the best average run with four threads for the *1h/D* dataset took 35.4 seconds whereas the best run in the sub-graph implementation and the same dataset ran in 53.7 seconds (run with six threads).

But it becomes clear where this issue originates from when comparing the given two datasets: as both datasets use the D infrastructure graph, the network is exactly the same. The only difference is the timetable size: 5450 trains in *1h* versus 15744 in *5h*. While the number of trains only tripled ($15744/5450 \approx 2.89$), the runtime increased nearly ten-fold ($338s/35.4s \approx 9.55$). This shows that the runtime is heavily influenced by the timetable size.

Nevertheless, this behaviour is actually to be expected. Previously, the number of times that needed to be considered was fixed. Furthermore, all steps from T_{MIN} up to T_{MAX} were planned, regardless of the actual time window, the trains were using. For example, if the timetable has one train from 500 to 700, and $T_{\text{MIN}} = 0, T_{\text{MAX}} = 1500$ then only 200 of the 1500 steps would be necessary.

However, due to the relaxation of the time, scheduling times are generated on-demand. When there are only a few trains, very few time steps have to be considered. Coming back to the example from above, now (assuming no passing requests occur) then only the 200 time steps are simulated. However, when there are many trains, then there are possibly also more time steps that have to be considered. This also multiplies when delays or passing requests occur as those events introduce even more times to consider. Furthermore, many of those events, especially delays, create "alternative timelines" that not only introduce one new time to schedule but multiples—a delayed train keeps getting delayed with a certain possibility, but also the non-delayed version needs to be tracked and so on. Furthermore, these issues are intensified by the constraint checking for each scheduling step and point in time, which, while looking simple on paper, do require a certain amount of processing power.

Speed-Up. For speed-up in this approach, we use a modified variant of the Definition 2.4.1. Instead, to compare the fastest run with the baseline run, we compare it to the one-threaded variant. This allows us to get a rough estimation of a possible speed-up.

For the *1h/D* dataset the speed-up can therefore be calculated by:

$$
\begin{aligned}
t_1 &= 49.578s && \text{(average of } N = 1 \text{ thread)} \\
t_4 &= 35.321s && \text{(average runtime with } N = 4 \text{ threads)} \\
S_4 &= t_1/t_4 = 1.4
\end{aligned}
$$

However, as can be seen from the table, for many other datasets the speed-up is practically not given in most of the cases as the fastest runs were with one thread.

This is reflected in an average of 1.06 as a speed-up value for all datasets.

Efficiency. The efficiency cannot reliably be measured in this iteration, as the concrete baseline is not implemented. Thus we omit this value here. However, this value is expected to be in line with the above results and thus show no significant improvements.

Profiling. Like before, the implementation was profiled using *Perf* in conjunction with *CLion*.

While running the profiling, only an average CPU usage of 40% across all active cores regardless of the dataset was noticed, indicating that the potential of the cores could not be used. This is probably due to unsatisfied constraints preventing the threads to efficiently schedule the respective chunks, slowing down the entire simulation.

The evaluation of the Perf run with dataset *1h/N*[1] revealed that more than 15% of this approach's runtime was due to the output of the current step to the command line. Unfortunately, this was discovered after the benchmark run and due to timely constraints, I was not able to redo the benchmark with the output disabled. However, smaller manual tests showed roughly 15-25% improvements. Thus the overall performance can easily be increased by disabling the status output.

Further, in this particular instance, approximately 25% of time was spent handling the required mutexes for queue and chunk modification. Additionally, the insertion and management of inserting new time steps to TaskQueue caused another 30% of the overall runtime, most of which was due to further mutex handling.

Learnings. In the evaluation of this approach, we had to conclude that the current implementation of time parallel execution does not offer any significant advantage to the previous chapters. Except for few examples, most runs experienced severe runtime increases when compared to the previous iterations. While this was partially expected, as the introduction of floating-point time also was expected to add more times that need to be rendered by the simulation. Despite the relaxation of the algorithm to allow for parallel time simulation, it appears as if either the constraints of the algorithm are too strong or due to the high dependence on each other not much parallel execution of time steps is possible.

The wear results also are partly due to a non-optimal implementation as a large portion of time was spent handling mutexes. While these are necessary for correct synchronization between chunks and valid task extraction, here a possible point for improvement can be found. If the synchronization and task handling can be reduced, the overall runtime will benefit from these changes.

Part of the penalties added by the increased time steps could possibly be mitigated by scheduling certain infrastructure elements in particular instead of chunks. Because, in most cases, only a handful of elements is expected to change and thus the scheduling of all other vertices or edges introduces preventable overhead.

However, it can be observed that the algorithm performs better if a finer GranularityLevel is chosen compared to the coarse level used in the rest of the evaluation. This

[1] In previous examples the *5h/D* dataset was chosen as it is both large in size and has a long timetable. However, for this run, I chose the smaller *1h/N* dataset as it could be analyzed faster and but also showed no speed-up.

Dataset	1-Thread	Fastest	Speed-up*	Best Thread
1h/D	49.578	35.421	1.400	4
1h/N	6.715	6.715	1.000	1
1h/N_M	11.892	10.227	1.163	2
1h/N_SO	12.799	10.634	1.204	2
1h/W_M_SW	18.732	15.680	1.195	3
2h/D	103.936	85.752	1.212	3
2h/M_W_SW	41.227	37.160	1.109	2
2h/N	14.528	14.528	1.000	1
2h/N_M	25.100	25.100	1.000	1
3h/D	167.470	148.892	1.125	3
3h/M_W_SW	65.855	64.167	1.026	2
3h/N	22.834	22.834	1.000	1
3h/N_M	40.954	40.954	1.000	1
4h/D	240.170	227.288	1.057	3
4h/M_W_SW	92.820	92.820	1.000	1
4h/N	33.213	33.213	1.000	1
4h/N_M	58.626	58.626	1.000	1
5d/D	367.966	367.966	1.000	1
5d/N	50.079	50.079	1.000	1
5d/N_M	85.453	85.453	1.000	1
5d/N_SO	91.650	91.650	1.000	1
5d/W_M_SW	139.412	139.412	1.000	1
5h/D	341.481	338.163	1.010	2
5h/M_W_SW	132.802	132.802	1.000	1
5h/N	45.532	45.532	1.000	1
5h/N_M	81.490	81.490	1.000	1
Max	367.966	367.966	1.400	4
Min	6.715	6.715	1.000	1
Mean	90.089	87.022	1.058	1.615

Table 6.1: Results from the benchmarks run on the desktop computer. *Baseline, 1-Thread, Fastest* in seconds.

shows that the implementation benefits from smaller more independent sub-graphs and which alleviates the negative aspects observed above slightly. Nevertheless, the overall runtimes are still much worse as in the previous attempts.

At last, there are two other takeaways from the profiling: always profile the algorithm prior to running any enduring tests and don't underestimate the impact of logging to the console.

6.3 Summary

In this approach, we modified the algorithm to allow for non-discrete floating point time and thus too parallel execution of time steps. For this, the constraints of the scheduling steps have been adjusted so that all trains are scheduled correctly.

The results have shown that, in its current form, the implementation does not offer significant improvements. I believe this is due to the strong coherence and dependences of the infrastructure network. This hinders the execution of scheduling tasks in parallel. Due to the time now being non-discrete, more points have to be considered, increasing the overall workload. Additionally, the increased scheduling load is slowed down further by a non-optimal implementation. With an revised implementation, the overall results can be improved. However, it is noted that due to the current scheduling order, not much time-parallelism is expected as many points in time require others to finish. However, if the constraints can be relaxed further and scheduling steps are more independent, then this approach might benefit from this.

Chapter 7

Approach 5: On-Demand Scheduling

After the time parallel approach, we decided to investigate the effect of only scheduling the infrastructure elements at a time where an update might occur. Previously, all vertices and edges of the infrastructure graph were scheduled in each time step regardless if a train is waiting, arriving or departing. This meant that, especially for large infrastructures with small timetables, a lot of elements were unnecessarily inspected each time step.

Therefore, in this chapter, the algorithm is modified in that way, that all infrastructure elements are scheduled on-demand and thus reducing the overall computational complexity. As it was observed in Chapter 6, enabling time-parallel simulation increases the need for bookkeeping and validity checks to such a degree that any benefits by the possible relaxation of time were mostly diminished. Thus, in this approach we again focus on the sequential simulation of time steps.

If the scheduling is to be done in parallel, within each time step, the infrastructure elements to be scheduled in that particular step are firstly checked for dependencies on each other: that is, if edges are planned, their source vertices are required to be scheduled first and thus have priority. These priority vertices are scheduled sequentially in advance. Once completed, the rest of the vertices and edges is then distributed to the available threads and processed in parallel. If the simulation is executed in non-threaded mode, the scheduling of vertices is done first, followed by the scheduling of edges, thus ensuring the correct order and not requiring the prioritization. For the parallel part, only few requirements need to be made when compared to the time-parallel simulation: vertices are independent of each other and do not interfere, thus they can be scheduled parallelly without the need for further checks. For the edges, if the source vertex was scheduled, the only possible point of conflict is if two edges that share the same source vertex are processed simultaneously. However, as this occurs seldomly, in this attempt this will be handled by mutual exclusion when accessing the data. Furthermore, this on-demand scheduling has the advantage that the division into chunks can be spared. Additionally, due to this reduction, the overall number of boost references and iterator calls are reduced which were diagnosed in the previous iterations to be one of the largest non-threading related contributors to the runtime.

During the initialization, the trains to be scheduled according to the given timetable are loaded. As the trains are placed on the edges from the artificial start vertex to

their first stop, this first target element is inserted in the list of elements to consider for the respective starting time.

Once initialization has concluded, the algorithms then finds the first point in time $t \in \mathbb{T}'$ that is larger than T_{MIN} and smaller than T_{MAX}. For this time step, all necessary vertices are then scheduled as before. During the scheduling phase, new points in time are discovered when updates to the network and the train's location are made. These changes include the removal of Blockers and the next waypoint in the itinerary of the train either as a regular next point or as passing train. Blockers are emplaced on the infrastructure elements as safeguards to prevent other trains from entering until a safe amount of time has passed. This is defined globally. To ensure that the complete time span is blocked correctly, a the next scheduling may only occur just slightly after the time has passed. For that a new variable $EPS \in \mathbb{R}^{\geq 0}$ is introduced which defines a small time delta that is added to the blocking time before inserting it in the list of points to consider. Passing requests are modelled similarly; if a train would pass multiple infrastructure elements, for each element on this passing itinerary, a limited amount of time shall pass. For that, once a train enters an infrastructure element only passing by, the next element is considered only reachable in time $t + EPS$. All other arrivals and departures are considered as before, i. e. when on time, the arrivals and departures according to the timetable are used and when delayed possible compensation methods are applied.

Once a time step has completed, the algorithm checks the list of time steps to consider for the next largest point in time $t' > t$ within the time horizon $\mathbb{T}'$, repeating the steps described above, until no suitable point in time can be found or the end of the time horizon is reached.

7.1 Implementation

In this approach, some of the changes of the previous implementation were reverted. The TaskQueue and the handling of chunks were not needed and could be removed. The double-precision floating point representation of time was kept on the other hand.

After removing the legacy code, two variables—`vertices2consider` or `edges2-consider`—, one for vertices and one for edges, were introduced. These map the points in time to the respective vertices and edges that require scheduling in this time step.

Starting with time T_{MIN} all further point in time are calculated during the scheduling of elements. For each of the points in time that are discovered during the scheduling, the associated vertex or edge is inserted into the respective map. For each scheduled point in time, the infrastructure element is blocked a certain time. Thus the next possible change is shortly after the blockage is released. To ensure that the entire blocking period is spend, the next point to considerer is scheduled at `t + BLOCKING_TIME + EPS` where `EPS` is a small offset. In this implementation it was set to `EPS = 0.001`. Furthermore, if the train was able to enter the respective infrastructure element, the next point in time is its earlies possible departure time `epdt` at the current element, which is its departure adjusted for possible abbreviated stop times or faster travel time to counteract delays. If a train was not able to enter a given infrastructure element due to it being blocked, it will automatically be considered when the blockage is resolved. Should a train depart only partially, both, the current and next waypoint are considered at their respective next possible departure

times. Lastly, the passing requests need to be handled. In this case, we used a similar approach to the previous implementation: as the time is modelled as floating point, we insert a new time to be considered which is slightly in the future. For this, we schedule the respective next infrastructure element at `t + EPS`.

The simulation processes each time step on demand, that is, it searches for the next available time step. As vertices and edges are decoupled, it might occur that in a time step only vertices and no edges need to be scheduled, or vice versa. Therefore, the respective smallest points in time of vertices and edges larger than the last processed one, need to be compared and only the smallest one will be processed accordingly. Once the correct time step has been found, the vertices and edges that are to be scheduled will be extracted. Before starting the scheduling, the source vertices of all edges are compared to the list of vertices that are to be scheduled at that point in time. If a vertex and the corresponding edge is planned to be scheduled in the same time frame, the vertex is transferred into a priority list as it is required to be scheduled first. Once the prioritization is complete, these vertices will be scheduled first in sequence and non parallelly. Concludingly, all remaining vertices and edges are scheduled by enqueueing the respective scheduling tasks in the ThreadPool. This will then use all available user-defined threads. In the baseline variant, this prioritization can be omitted, as all vertices are scheduled first in sequence followed by the edges. Once all infrastructure elements have been scheduled in one time step, the simulation proceeds with the next smallest point in time larger than the just completed one. As the vertices and edges might have different points in time that need to be considered next, only the smallest one is processed in this time step, while the other is scheduled in the future.

7.2 Analysis

In the following, we will analyse the results from these changes. As before, the simulation was executed on the desktop computer using the benchmark. In total ten rounds per thread and dataset were conducted. The resulting runtimes were noted in a CSV-file.

Visual Results. After executing the simulation, the resulting runtimes per dataset were plotted using Python, Pandas and Seaborn and are depicted in Figure 7.1 with a more detailed graph of the *5h* datasets in Figure A.1e in the appendix.

It clearly be seen that using only one thread introduces a slight but noticeable overhead compared to the non-parallelized baseline ("0 threads"). However, the longer a simulation run took, the higher the overall speed-up when using the threaded variant. This is especially visible with the longest three results in the graph corresponding to the *4h/D, 5h/D* and *5d/D*. All three represent datasets with both large infrastructure networks and large timetables. This indicates that the new algorithm makes the best use of threads when there are many trains to be scheduled in a large network. This is to be expected from the implementation, as only the scheduling of vertices and edges is done in parallel per time step. And as the number of infrastructure elements to be scheduled per time step increases with the size of the time table, the improvements by parallelization are expected to be the best in this case.

It can also be noted that the observed variance in runtime was very low for all datasets indicating a very stable result.

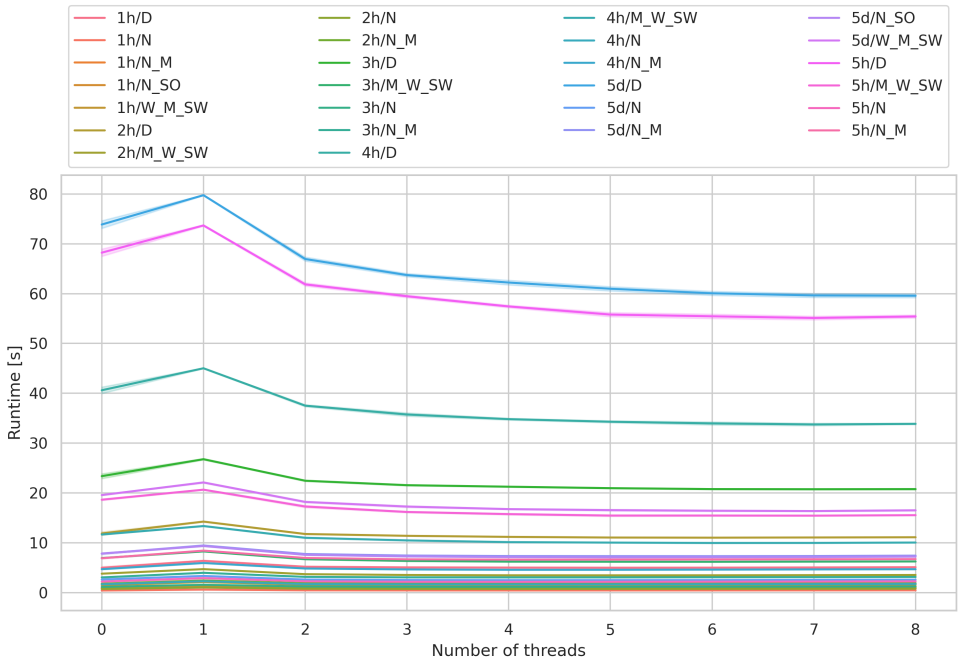


Figure 7.1: Runtimes of On-Demand approach on desktop.

While the improvements within this approach appear to be adequate, when compared to the results from the Sub-Division Approach presented in chapter 5, the overall improvements are very good. To compare the results, we compiled two additional graphs: the first showing the runtimes of both approaches (Sub-Graph Approach with GranularityLevel OrgaUnit and this one) and can be seen in Figure 7.2. The second shows relative runtime improvements when comparing the baseline and fastest runs of the respective datasets. It is presented in Figure 7.3.

In Figure 7.2 the runtimes of the simulation per dataset is shown on the desktop. On the left, the baseline runs are compared whereas on the right the respective fastest runtime is depicted. Errors are shown in form of standard deviation. It becomes clear that this new on-demand approach reduces the runtime of the simulation in both cases significantly, regardless of threading. Especially, for large datasets with only a small timetable, e.g. *1h/D*, the reduction in runtime is from over 60 seconds in the baseline or approximately 55 seconds for the fastest run to less than ten using the new on-demand scheduling.

The improvements in runtime can be found in Figure 7.3. The graphs show the relative improvement in runtime. The blue *F/F* bars show the relative improvements when comparing the baseline runs of the on-demand and the sub-graph approaches. The orange *F/B* represent the relative runtime improvements of the fastest run of the on-demand to the baseline of the sub-graph approach and lastly the green *F/F* bars depict the comparison of the respective fastest runs. Each bar group is per dataset.

Here too, the extend of speed-up becomes more clear. In many cases, the relative improvement is larger than 80% when comparing the results of the on-demand to the sub-graph approaches. This graph, too, shows that this on-demand algorithm reduces

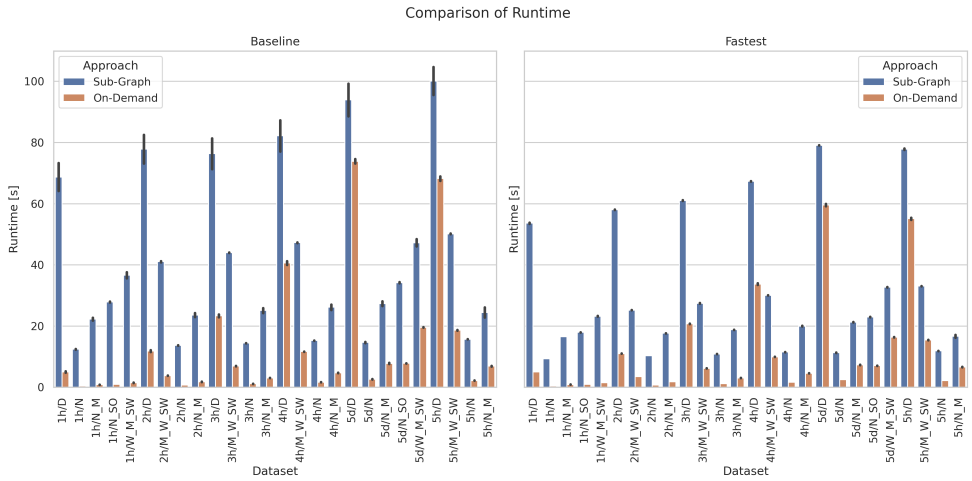


Figure 7.2: Runtimes of Sub-Graph approach (GL OrgaUnit) and On-Demand approach on the desktop computer. Error bars show standard deviation. Left graph compares baseline runs, right one compares the fastest runs of each approach.

the load significantly for especially small timetables.

Analytical Results. To put the visuals into numbers, the runtime results are compiled in Table 7.1.

It can be seen that the this approach benefited from multithreading, too. For all datasets, the on average fastest multithreaded run was achieved with at least four threads. The most relevant influencing factor is the size of the timetable, as for most datasets using a small timetable (1h–3h), the fastest variant seldomly used more than four or five threads. Datasets using a larger timetable, such as the 4h and 5h ones, more often were able to achieve faster runtimes utilizing five to seven threads. This is also expected as the larger datasets require more scheduling operations per time step and thus can make use of the available threads for parallel computation.

Speed-Up. Next, we will compare the achieved speed-up as has been done in the previous chapters. For this, the formula as in Definition 2.4.1 is used. As exemplary values, we will use the results of the *1h/D* dataset as reference:

$$
\begin{aligned}
t_0 &= 4.984s && \text{(average of } N = 0 \text{ thread)} \\
t_5 &= 5.002s && \text{(average runtime with } N = 5 \text{ threads)} \\
S_5 &= t_1/t_5 = 0.996
\end{aligned}
$$

As this value is below 1.0, this indicates that for this dataset, the threaded variant was slightly slower. However, previously we have seen that the on-demand approach benefits from larger timetables. Therefore, if we compare this value to the speed-up for the *5h/D* or *5d/D* datasets, we get a speed-up of approximately 1.24. This means, that the threaded variant was about 25% faster than the baseline. While

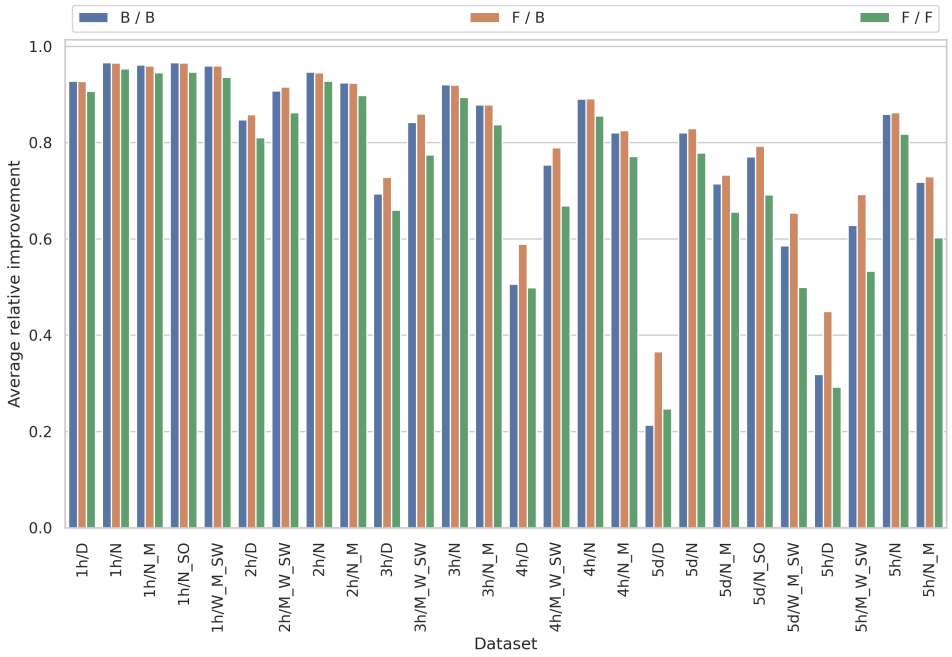


Figure 7.3: Relative improvements of runtimes comparing the Sub-Graph approach (GL OrgaUnit) and On-Demand approach on the desktop computer. Blue (B/B) shows the relative improvement of the baseline runs, orange (F/B) compares the fastest run of the on-demand approach to the sub-graph baseline, and lastly the green (F/F) depicts the relative improvement of the fastest runs.

these results are not as good as the previous, approaches when only comparing the results within the approach, the overall speed-up to the previous variants is significant. Furthermore, it can be seen in the Table 7.1 that speed-up increased with the size of the timetable. This might indicate, that the improvements could be better for larger timetable datasets as more scheduling tasks per time step would be necessary and could be executed in parallel.

Efficiency. Like for the previous attempts, we also calculate the efficiency of the threaded variants. This is done as described in the formula of Definition 2.4.2. For the example calculation, we again use the *1h/D* dataset for reference.

$$
\begin{aligned}
t_B &= 4.984s && \text{(average baseline)} \\
t_1 &= 6.380s && \text{(average of parallel w/ 1 thread)} \\
r &= t_1/t_B \approx 1.28
\end{aligned}
$$

A value of 1.28 indicates an overhead of 28% when using the threaded variant compared to the baseline approach. This result is in the middle of the remaining datasets. The values range from 1.080 for the *5h/D* and *5d/D* datasets to 1.433 for

Dataset	Baseline [s]	1-Thread [s]	Fastest [s]	Speed-up	Efficiency	Best Thread
1h/D	4.984	6.380	5.002	0.996	1.280	5
1h/N	0.424	0.604	0.434	0.976	1.426	4
1h/N_M	0.855	1.225	0.904	0.946	1.433	4
1h/N_SO	0.949	1.297	0.959	0.990	1.367	4
1h/W_M_SW	1.492	2.122	1.501	0.994	1.422	4
2h/D	11.868	14.254	11.029	1.076	1.201	6
2h/M_W_SW	3.801	4.713	3.474	1.094	1.240	5
2h/N	0.732	1.040	0.746	0.982	1.421	4
2h/N_M	1.788	2.433	1.808	0.989	1.361	4
3h/D	23.373	26.775	20.75	1.126	1.146	7
3h/M_W_SW	6.944	8.269	6.184	1.123	1.191	6
3h/N	1.146	1.573	1.158	0.990	1.373	4
3h/N_M	3.056	3.976	3.058	0.999	1.301	4
4h/D	40.614	45.026	33.752	1.203	1.109	7
4h/M_W_SW	11.661	13.362	9.964	1.170	1.146	6
4h/N	1.672	2.281	1.661	1.007	1.364	4
4h/N_M	4.704	5.948	4.580	1.027	1.264	4
5d/D	73.873	79.759	59.577	1.240	1.080	8
5d/N	2.633	3.351	2.503	1.052	1.273	4
5d/N_M	7.812	9.461	7.322	1.067	1.211	6
5d/N_SO	7.844	9.354	7.085	1.107	1.193	5
5d/W_M_SW	19.58	22.109	16.371	1.196	1.129	7
5h/D	68.223	73.687	55.114	1.238	1.080	7
5h/M_W_SW	18.644	20.659	15.447	1.207	1.108	7
5h/N	2.216	2.909	2.167	1.023	1.313	4
5h/N_M	6.899	8.427	6.621	1.042	1.221	4
Max	73.873	79.759	59.577	1.240	1.433	8.0
Min	0.424	0.604	0.434	0.946	1.080	4.0
Mean	12.142	13.742	10.341	1.067	1.248	5.0

Table 7.1: Evaluation results of the on-demand approach on the desktop. Runtimes in seconds.

1h/N_M dataset. From the results it can be seen that datasets with both, a large infrastructure network and timetables, tend to result in a better efficiency.

Comparison to previous iterations. In the visual analysis, we already started to compare the new on-demand approach to the previous non-time-parallel implementation, the sub-graph approach as presented in Chapter 5.

Figure 7.3 already indicated that depending on the dataset, improvements in runtime of over 80% could be achieved. In the following, we want to investigate this further and therefore compiled a table containing the numerical improvements. This can be found Table 7.2.

With the help of the table, we can confirm the visual results: for short timetables that are scheduled on large infrastructure networks, the runtime of the algorithm can be reduced up 96.6% (see *1h/N_SO*) for the baseline comparison and up to 95.3% (see *1h/N*) for the comparison of the respective fastest threaded runs.

	Baselines		Fastest	
Dataset	Absolute Difference [s]	Relative Difference [%]	Abs. Diff. [s]	Rel. Diff [%]
1h/D	63.755	92.749	48.656	90.677
1h/N	12.028	96.596	8.883	95.338
1h/N_M	21.417	96.160	15.622	94.527
1h/N_SO	26.941	96.596	16.993	94.656
1h/W_M_SW	35.140	95.927	21.740	93.540
2h/D	65.970	84.753	47.011	80.997
2h/M_W_SW	37.260	90.743	21.752	86.230
2h/N	12.946	94.648	9.538	92.750
2h/N_M	21.836	92.430	15.870	89.770
3h/D	52.992	69.393	40.307	66.016
3h/M_W_SW	37.069	84.223	21.271	77.475
3h/N	13.277	92.055	9.724	89.358
3h/N_M	22.046	87.826	15.761	83.750
4h/D	41.587	50.592	33.610	49.894
4h/M_W_SW	35.609	75.331	20.106	66.864
4h/N	13.606	89.055	9.847	85.570
4h/N_M	21.461	82.022	15.438	77.122
5d/D	20.077	21.370	19.538	24.696
5d/N	12.032	82.048	8.808	77.869
5d/N_M	19.577	71.478	13.966	65.604
5d/N_SO	26.375	77.077	15.873	69.140
5d/W_M_SW	27.687	58.576	16.345	49.959
5h/D	31.915	31.871	22.748	29.216
5h/M_W_SW	31.540	62.849	17.627	53.295
5h/N	13.491	85.892	9.725	81.778
5h/N_M	17.570	71.805	10.050	60.287

Table 7.2: Numerical improvements of Baseline variants and the fastest variants for each approach and dataset respectively comparing Sub-Graph (GL OrgaUnit) and On-Demand approaches.

This, again, is very logical when comparing the two scheduling mechanisms. In the sub-graph variant, the entire infrastructure network was checked in each time step even though only a handful of trains are scheduled within. This resulted in a large overhead. In the on-demand variant, only the particular infrastructure elements at which some change occurs are scheduled for each time step. With increasing timetable size, the number of elements that need to be scheduled increase as well thus leading to more scheduling steps. Therefore, the improvements in runtime get smaller. Nevertheless, for the given datasets, in the worst case a reduction of 21.4% (*5d/D* baseline) or 24.5% (*5d/D*) are observed.

Profiling. To investigate possible bottlenecks with regard to parallel execution, the on-demand simulation was profiled using the *5h/D* dataset. Analysis was done using *Perf* in conjunction *CLion*.

In the results, no significant bottlenecks were found. The scheduling of vertices

consumed about 41% of the time. However, 77% of this or 32% in total were due to the calculation of capacities. Scheduling the edges utilized roughly 5% of the time. As for using the ThreadPool for managing and distributing the tasks to threads, this introduced roughly 1% to 2% of additional overhead. The required mutex locks for management and safe access did not appear in the listing and thus made up of less than 1% of the time.

The CPU utilization during the scheduling was around 40% and used about 500 MiB of RAM.

This indicates that the simulation does not suffer from any direct bottleneck with regard to parallel execution but rather from the lack of parallelly executable tasks.

Learnings. The implementation of the on-demand scheduling improved the overall runtime significantly. While the improvements of parallelized variants is relatively small, these are nevertheless promising. As optimization of this approach was not possible due to the limited time of this thesis, further improvements could be made. For example, as the algorithm benefits from parallel scheduling of elements per time step, it would be feasible to increase the effectiveness of parallel computation if certain time segments would be merged and thus more trains being scheduled at the same time. This would also mitigate problems with high train loads and the discovery of to many fine granular time steps, overloading the simulation. One could also try to completely revert the floating point representation and use discrete integer time, however this would invoke special handling of passing trains.

Regardless, this implementation showed that the algorithm has much optimization potential. However, it is hard to do so with parallel execution: as only a few calculations can be split so that these can be processed in parallel.

7.3 Summary

In this chapter, we presented an overhaul of the way the fundamental way the network is scheduled. Instead of checking the entire infrastructure network in each time step for possible updates, only elements that are known to change during a certain time step will be processed. This reduced the overall execution time of the simulation significantly, especially for large networks with only a small number of trains being scheduled. In its current implementation, the simulation benefits of parallel scheduling the best with regard to runtime when a large number of independent elements are scheduled in one time step. However, in the datasets and current implementation this is rather rarely the case and thus only limited improvements in threaded execution were observer. This, nevertheless, offers point for improvement in future iterations which were not possible due to the limited time available for this thesis.

Chapter 8

Conclusion

To conclude this thesis, I will summarize what approaches were implemented to achieve parallelization in the probabilistic train timetable simulation. The summary is followed by a discussion about the results and learnings that were found while working on this topic. Lastly, this chapter and with it this thesis will conclude by presenting some thoughts on possible future work.

8.1 Summary

In this thesis, we investigated the probabilistic railway timetable simulation, as presented by Haehn, Ábrahám and Niessen in [15] with regard to parallelization approaches in order to make the best possible use of available hardware. To achieve parallelization, different approaches inspecting several aspects with regard to the parallel execution of simulation steps were developed in an iterative manner.

The initial implementation brought basic parallelism to the simulation algorithm, by dividing the available input network into equal sized parts. This approach, however, did not yield any significant improvement due to extensive locking and waiting of threads as well as the rudimentary parallelism technique implemented.

In the following, second attempt, the issues that were discovered during the initial implementation and its analysis, were addressed. With these changes, we were able to reduce the overall runtime of the algorithm by approximately 36% on average. However, while these improvements were not neglectable, the resulting gains are rather small when compared to the available hardware power.

Thus, in the subsequent third attempt, we tried to achieve better hardware utilization and thus faster runtimes by splitting the infrastructure into logical sub-graphs. With this, the amount of synchronization was aimed to be reduced. However, the implemented changes were only able to reduce the runtime marginally.

The next attempt inspected the implications of parallelly processing time steps. This task was found to be quite difficult, as conditions and constraints needed to be put in place in order to guarantee a correct simulation result. While the time parallel approach was able to reduce the overall runtime from datasets with a small time table due to on-demand scheduling, for larger datasets the increased amount of time steps due to the on-demand scheduling and the relaxation of time from integer to floating-point numbers, the runtime increased drastically the more threads were used. The combination of additional constraints and checks along with the increased number of

time steps resulted in relatively poor parallel performance, with exponential runtime in the worst cases. Due to time constraints of this thesis, no further optimization could be implemented, resulting in sub-par algorithm runtimes.

In a final approach, some parts of the previous iteration were kept, such as the floating-point representation of time and the concept of on-demand scheduling was extended. The time-parallel simulation was reverted due to its issues described above. To reduce the number of unnecessary scheduling steps for infrastructure elements at points in time where nothing happens, a list of elements and time steps is dynamically generated throughout the simulation. This leads to an on-demand scheduling of elements at only those points in time, when an update is required. This showed to be very effective in reducing the overall runtime of the simulation, especially for large networks with only a small timetable. This resulted in runtime reductions of up to 95%. This approach implemented parallelization during each time step, similarly as before. As only a limited number of updates occurs, many of the elements can be scheduled parallelly without the need for many constraints. However, due to the current implementation, only few elements are simulated in a time step and thus reducing the effectiveness of the parallel execution. This and a possible flooding of time steps, like observed in the previous iteration, can lead to severe issues regarding the runtime if not accounted for. A possible solution would be to merge certain time steps, on the one hand increasing the number of parallelly schedulable trains and on the other hand reduce the overall points in time to consider. Nevertheless, an average increase of only 7% was noted, however datasets with larger timetables benefited significantly from parallelization for a speed-up of up to 24%. Due to timely constraints, this could not be optimized within the scope of this thesis, offering potential future work.

8.2 Learings

In the course of this theses, we were able to establish that threads offer an relatively easy way to enable parallel computing for an algorithm. Modern C++ standards offers many interfaces for the inclusion of threads in C++ projects. While it might be easy to add multi-threading, doing so efficiently is far from easy. During the work on this thesis, we implemented several different approaches making use of multi-threaded computing and parallel computation of tasks within the simulation. We learned that, due to the shared memory in the nature of threads, synchronization is important to avoid conflicts and data corruption. However, this also introduces bottlenecks and waiting points which can severely influence the effectiveness of the parallelization. Hence, it is important to find a balance in pre-processing and dividing the shared data in such a way, that conflicts are avoided and the need for synchronization is reduced to a minimum. Regardless of the data, it may also be the case that some operations that might be scheduled in parallel are severely dependent on each other so that the effective simultaneous processing is not possible.

Thus concluding, while achieving simple parallelism in a given algorithm can be achieved relatively fast with modern C++. But how to achieve efficient parallel execution is dependant on various factors, such as the algorithm itself, the operating system and the hardware. Further, dependency management and synchronization overhead are the main initiators of increased runtime and the goal is therefore, to reduce such as much as possible.

8.3 Future work

As the scope of this master's thesis was limited and due to timely constraints, only a couple of approaches and optimizations could be implemented and evaluated. As the results so far show, there is still headroom for optimization and improvements with regard to the runtime of the simulation algorithm, thus offering interesting topics for future work.

To optimize the utilization of available hardware and thus allow for faster runtimes, the algorithm itself can be investigated. As this thesis only introduced rather minor changes to the simulation mechanics, in particular the scheduling of trains, the results are limited to what this current state allows for. As we have seen in the results above, the parallel execution is often times severely limited by the dependencies of the vertices and edges especially with regard to their timely execution. Hence, when the algorithm would be rewritten and customized so that those limiting dependencies are minimized, it would allow for better parallelizability. Here, especially the possible focus on train-based simulation instead of an infrastructure-base one could be of advantage.

Concerning the first approaches where the infrastructure network was divided into separate chunks, it can be investigated in how far this can be improved on. With better chunk generation, the dependencies between to chunks could potentially minimized and thus result in less synchronization and bookkeeping.

Furthermore, as it was noted in Chapter 7, the reduction of constraints and on-demand scheduling reduced the overall runtime of the simulation significantly, regardless the use of multi-threading. Threads were only able to speed up the already good results by only a bit. The issue here is, that only a handfull of trains are scheduled in each time step and that the time step resolution might be to fine. As a detailed resolution of time steps might also lead to problems and slowdowns, the investigation of either reverting to integer-based time steps or an optimized merging procedure of times might offer better utilization of threads. This approach in general, seems to be an interesting starting point for further inspection due to its good performance and few bottlenecks.

Additionally, the time-parallel approach may be revisited. Here, improvements in runtime might be possible by reducing the number of required checks and constraints. For this, an optimized hybrid time-parallel on-demand approach might offer further improvements. Furthermore, a revised implementation might also resolve some issues with regard to the prolonged execution time for long timetables on large networks.

Also other implementations of multithreading in C++ for example by using OpenMP[1] could be used instead of the native C++11/14/17 standards. While this requires severe changes in the implementation as it works quite different[2] to the standard C++one. However, depending on the workload, it may also affect the overall runtime, as it has been found to perform better in certain situations [37].

Furthermore, this thesis focussed on the usage of threads for parallelization as it is a very powerful resource for this task with relative easy implementation possibilities. However, as presented in Chapter 2, multiple processes can be used besides threads. These require some additional work with regard to distributed data and its consistency, as well as harder communication overhead.

Lastly, the simulation could extended with further features such as delay predic-

[1] `https://www.openmp.org/`

[2] OpenMP uses compiler directives instead of active code. Thus requiring other implementations and a compatible compiler. However, most modern popular compiler support it.

tion. Another interesting extension could the implementation of other methods to recover from delay, for example re-routing delayed trains when certain infra structure element capacities are stressed. Furthermore, the introduction of events influencing a trains arrival or departure times, such as passengers blocking the doors, bad weather or other incidents could be examined.

Bibliography

[1] Umut A. Acar, Arthur Chargueraud and Mike Rainey. *An Introduction to Parallel Computing in C++*. Version 1.2. Archived on 2021-02-12. Mar. 2016. URL: https://web.archive.org/web/20210212165921/https://www.cs.cmu.edu/~15210/pasl.html (visited on 12/02/2021).

[2] Advanced Micro Devices Inc. *Ryzen 5000 Series Desktop Processors*. 2020. URL: https://web.archive.org/web/20210211184208/https://www.amd.com/en/processors/ryzen (visited on 11/02/2021).

[3] Gene M. Amdahl. 'Validity of the single processor approach to achieving large scale computing capabilities'. In: *AFIPS Joint Spring Conference Proceedings 30*. ACM Press, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560.

[4] Keith Briggs and Christian Beck. 'Modelling train delays with q-exponential functions'. In: *Physica A: Statistical Mechanics and its Applications* 378 (2 10th Nov. 2006), pp. 497–504. DOI: 10.1016/j.physa.2006.11.084.

[5] Beda Büchel, Thomas Spanninger and Francesco Corman. 'Empirical dynamics of railway delay propagation identified during the large-scale Rastatt disruption'. In: *Scientific Reports* 10.1, 18584 (29th Aug. 2020). ISSN: 2045-2322. DOI: 10.1038/s41598-020-75538-z.

[6] Francesco Corman and Pavle Kecman. 'Stochastic prediction of train delays in real-time using Bayesian networks'. en. In: *Transportation Research Part C: Emerging Technologies* 95 (Oct. 2018), pp. 599–615. DOI: 10.3929/ETHZ-B-000281218.

[7] Marco Danelutto et al. 'Parallelizing High-Frequency Trading Applications by Using C++11 Attributes'. In: *2015 IEEE Trustcom/BigDataSE/ISPA*. IEEE, Aug. 2015. DOI: 10.1109/trustcom.2015.623.

[8] DB Netz AG. *Betriebsstellenverzeichnis – Open Data*. German. 2018. URL: https://data.deutschebahn.com/dataset/data-betriebsstellen (visited on 26/02/2021).

[9] Deutsche Bahn AG. *Wettbewerbskennzahlen 2019/2020*. German. Archived. June 2020. URL: https://web.archive.org/web/20210118235716/https://www.deutschebahn.com/de/konzern/Wettbewerb/verkehr_politik-3844352 (visited on 18/01/2021).

[10] European Environment Agency. *Passenger and freight transport demand in Europe*. 19th Dec. 2019. URL: https://www.eea.europa.eu/ds_resolveuid/653cb59bd80f49de891d49195f159971 (visited on 08/12/2020).

[11] Stefano de Fabris, Giorgio Medeossi and Giuliano Montanaro. 'trenissimo: Improving the microscopic simulation of railway networks'. In: *Computers in Railways XVI : Railway Engineering Design and Operation*. Vol. 181. WIT Press, July 2018, pp. 199–211. DOI: 10.2495/CR180181.

[12] Jesús Fernández-Villaverde and David Zarruk Valencia. *A Practical Guide to Parallelization in Economics*. Working Paper. National Bureau of Economic Research, Apr. 2018. DOI: 10.3386/w24561.

[13] Burkhard Franke et al. 'OnTime – Network-wide Analysis of Timetable Stability'. In: *5th International seminar on railway operations modelling and analysis* (Copenhagen, Denmark). May 2013.

[14] HaCon. *Train & Capacity Planning*. URL: https://web.archive.org/web/20210214130258/https://www.hacon.de/en/solutions/train-capacity-planning/ (visited on 14/02/2021).

[15] Rebecca Haehn, Erika Ábrahám and Nils Nießen. 'Probabilistic Simulation of a Railway Timetable'. en. In: *20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2020)*. Ed. by Dennis Huisman and Christos D. Zaroliagis. Vol. 85. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 16:1–16:14. ISBN: 978-3-95977-170-2. DOI: 10.4230/OASICS.ATMOS.2020.16.

[16] Intel Corporation. *Intel Core i3 Processors*. 2020. URL: https://web.archive.org/web/20210126021409/https://www.intel.com/content/www/us/en/products/processors/core/i3-processors.html (visited on 26/01/2021).

[17] Intel Corporation. *Intel Core i9 Processors*. 2020. URL: https://web.archive.org/web/20210211183010/https://www.intel.com/content/www/us/en/products/processors/core/i9-processors.html (visited on 11/02/2021).

[18] L. Muller J-P. Bendfeldt U. Mohr. 'RailSys, A System To Plan Future Railway Needs'. In: *International Conference on Computers in Railways*. Vol. VII. Bolgna, Italy: WIT Press, 2000, pp. 249–255. DOI: 10.2495/CR000241.

[19] Ralf Jahr, Mike Gerdes and Theo Ungerer. 'A pattern-supported parallelization approach'. In: *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores - PMAM '13*. ACM Press, 2013. DOI: 10.1145/2442992.2442998.

[20] David Janecek. *LUKS® Leistungsfähigkeitsuntersuchung Knoten und Strecken*. German. 14th June 2010. URL: http://ifev.rz.tu-bs.de/RailAutomation/RA2010/Internet/5_Janecek.pdf (visited on 14/04/2021).

[21] Peter Kipfer. 'Distribution and Parallelization Strategies for Integrated Simulation, Visualization and Rendering Systems'. PhD thesis. Universität Erlangen-Nürnberg, 18th Feb. 2003.

[22] Wei Li and Wei Zhu. 'A dynamic simulation model of passenger flow distribution on schedule-based rail transit networks with train delays'. In: *Journal of Traffic and Transportation Engineering (English Edition)* 3.4 (Aug. 2016), pp. 364–373. DOI: 10.1016/j.jtte.2015.09.009.

[23] Xuejun Liang, Ali A. Humos and Tzusheng Pei. 'Vectorization and Parallelization of Loops in C/C++ Code'. In: *Frontiers in Education: Computer Science and Computer Engineering*. MERCURY LEARNING & INFORMATION, 1st Feb. 2018. ISBN: 160132457X.

[24] Milos Ljumovic. *C++ Multithreading Cookbook. over 60 recipes to help you create ultra-fast multithreaded applications using C++ with rules, guidelines, and best practices*. Birmingham, UK: Packt Pub, 2014. ISBN: 9781783289790.

[25] Arnoud Mouwen. 'Drivers of customer satisfaction with public transport services'. In: *Transportation Research Part A: Policy and Practice* 78 (Aug. 2015), pp. 1–20. DOI: 10.1016/j.tra.2015.05.005.

[26] A. Nash and D. Huerlimann. 'Railroad Simulation Using OpenTrack'. In: *WIT Transactions on The Built Environment*. Vol. 10. WIT Press, 2004, p. 10. ISBN: 1-85312-715-9. DOI: 10.2495/CR040051.

[27] OpenTrack. *OpenTrack Railway Simulation*. URL: https://web.archive.org/web/20210214125348/http://www.opentrack.ch/mobile/opentrack_e/opentrack_e.html (visited on 14/02/2021).

[28] James Reinders et al. *Data Parallel C++*. Apress, 2021. DOI: 10.1007/978-1-4842-5574-2.

[29] REPARA Project. *Reengineering and Enabling Performance and poweR of Applications*. Site not reachable. Snapshot via archive.org. 2015. URL: https://web.archive.org/web/20190916033500/http://repara-project.eu/ (visited on 16/09/2019).

[30] RMCon Internatonal. *RailSys Software Suite*. URL: https://web.archive.org/web/20210214131945/https://www.rmcon-int.de/railsys-en/railsys-suite/ (visited on 14/02/2021).

[31] Sudhakar Sah and Vinay G. Vaidya. 'A Review of Parallelization Tools and Introduction to Easypar'. In: *International Journal of Computer Applications* 56.12 (Oct. 2012), pp. 30–34. DOI: 10.5120/8944-3108.

[32] trafit. *Punctuality prognosis with OnTime*. 2020. URL: https://web.archive.org/web/20210214185016/https://www.trafit.ch/en/ontime (visited on 14/02/2020).

[33] trenolab. *tenissimo. An intuitive approach to railway simulation*. URL: https://web.archive.org/web/20210214124721/https://www.trenolab.com/tools/trenissimo/ (visited on 14/02/2021).

[34] VIA Con. *LUKS – Analysis of lines and junctions*. URL: https://web.archive.org/web/20210214132552/https://www.via-con.de/development/luks/.

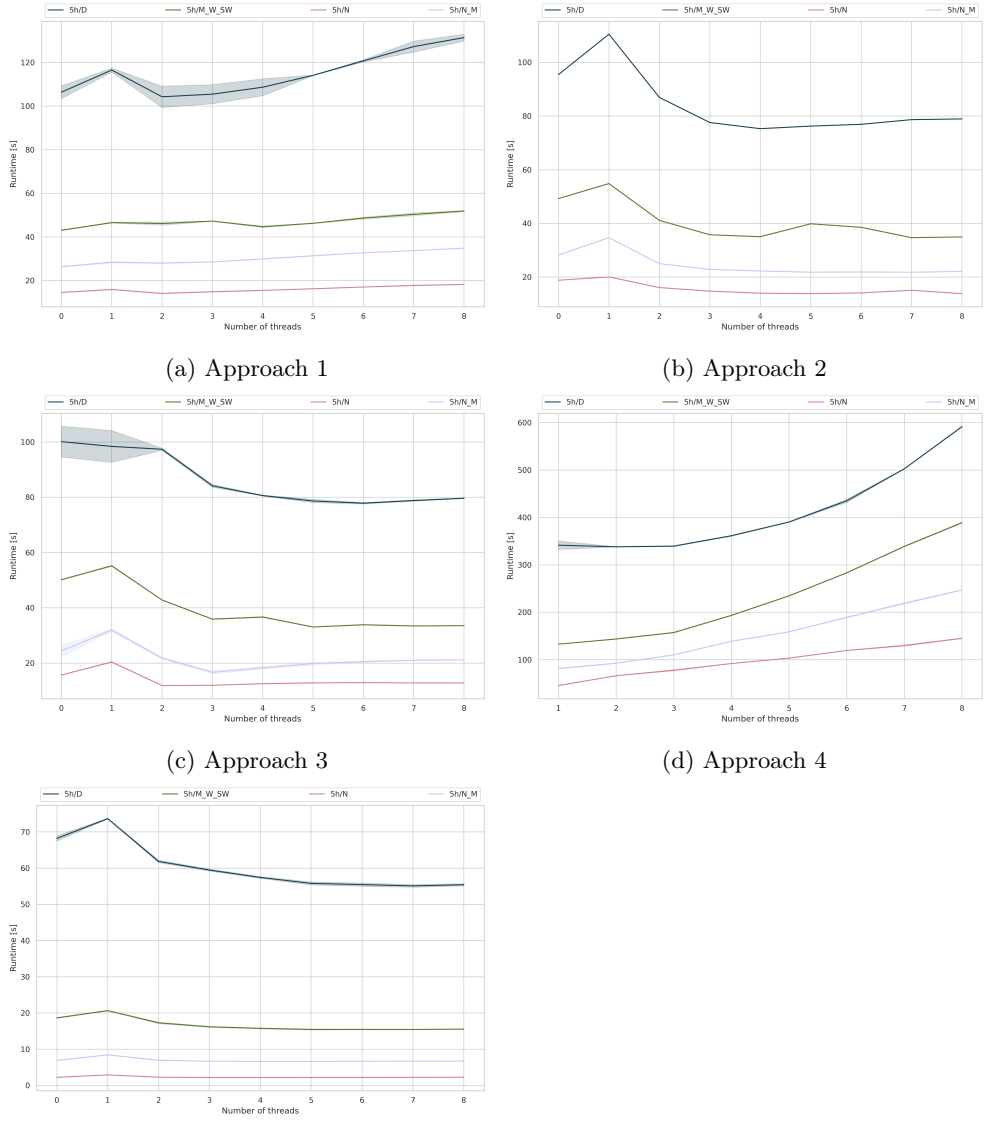
[35] Anthony Williams. *C++ Concurrency in Action*. 2nd ed. Manning Publications, 22nd Feb. 2019. 592 pp. ISBN: 1617294691.

[36] Jürgen Wolf. *Grundkurs C++. eine kompakte Einführung in die Programmiersprache C++*. 2nd ed. Bonn: Galileo Press, 2013. ISBN: 9783836222945.

[37] Joel Yliluoma. *SIMD and Vectorization: Parallelism in C++*. Informal Study on Parallelism in C++ and published on GitHub and YouTube. 10th Jan. 2019. URL: https://github.com/bisqwit/cpp_parallelization_examples (visited on 19/04/2021).

Appendix A

Additional Visualizations



(a) Approach 1

(b) Approach 2

(c) Approach 3

(d) Approach 4

(e) Approach 5

Figure A.1: Runtimes for each approach on the desktop only showing the *5h* datasets.

Zentrales Prüfungsamt/Central Examination Office

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Assenmacher, Lukas Marian | 331 357

Name, Vorname/Last Name, First Name | Matrikelnummer (freiwillige Angabe) Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende ~~Arbeit/Bachelorarbeit/~~ Masterarbeit* mit dem Titel

I hereby declare in lieu of an oath that I have completed the present ~~paper/Bachelor thesis/~~Master thesis* entitled

Parallelization of a Probabilistic Railway Timetable Simulation

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Aachen, 18.05.2021

Ort, Datum/City, Date | Unterschrift/Signature

*Nichtzutreffendes bitte streichen

*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Aachen, 18.05.2021

Ort, Datum/City, Date | Unterschrift/Signature