GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Cars in the traffic of a city network and resulting traffic jams in Go

Valerius Mattfeld, Bianca Vetter

MatrNr: 11580056, 21861974

Supervisor: Jonathan Decker

Georg-August-Universität Göttingen
Institute of Computer Science

October 14, 2023

Abstract

In this report, we implement a vehicle simulation using both a sequential and parallelized approach. The goal was a working implementation as well as an analysis of whether the provided implementation is parallelizable and how well it scales. As for the results, both implementations are executable, and various parameters are adjustable by the user. The proposed solution utilizes a realistic city map and the parallelization is achieved by splitting the map into sub-graphs and having each of the nodes manage one sub-graph respectively. The results inspecting strong scaling were promising. On the other hand, the results describing the weak scaling are ambiguous due to the lack of data.

Statement on the usage of ChatGPT and similar tools in the context of examinations

In this work I have used ChatGPT or a similar AI-system as follows:

- ☐ Not at all
- ☑ In brainstorming
- ☐ In the creation of the outline
- ☐ To create individual passages, altogether to the extent of 0% of the whole text
- ☐ For proofreading
- ☑ Other, namely: GitHub Copilot

I assure that I have stated all uses in full.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

HPC High-Performance Computer

MPI Message Passing Interface

OSM OpenStreetMap

API Application Programming Interface

XML Extended Markup Language

JSON JavaScript Object Notation Format

DFS Depth-First-Search

1 Introduction

1.1 About this project

This project was developed as part of the *Practical Course on High-Performance Computing* at the Georg-August-University of Göttingen. After a week-long, hands-on course on parallel computing approaches and tools was concluded, a solution to a non-trivial problem of choice had to be developed. The objective was to first establish a sequential solution that would then be expanded by a parallelized approach. Finally, an analysis of the program was also required, in which the scalability. of the solution is explored.

As the title suggests, this particular report covers a solution for building a city network that focuses on the movement of vehicles through the mentioned network. The idea is to build a simulation-type of a program that emulates city traffic. This could be used on a specific city layout with a realistic number of participating vehicles that serve a research purpose on, for example, bottlenecks or traffic jams at rush hours. The solution could be used to test out different layouts that could help improve the overall time an average vehicle spends in traffic getting from one place to another.

The main goal of this project is to have a working solution that can be executed in a sequential as well as a parallel manner. Furthermore, an analysis of the parallelization is required. The analysis should provide information on whether different resource configurations influence the parallelization and how. As elaborated in section 2, real-world data is the most optimal data to benchmark on. There we explore the possibilities of having OSM at hand, [Map].

1.2 Implementation Language

Due to the following reasons, our initial choice of programming language to write our simulations was Go. It separates itself in the following aspects from other programming languages:

- **Concurrency support**: Go has built-in support for concurrency and Goroutines and methods, which can help handle multiple tasks efficiently at the same time. This is particularly useful for traffic simulations, where multiple crews or vehicles need to be simulated simultaneously, [var]
- **Easy and readable**: The Go syntax is simple and easy to use, which can help to quickly create and maintain traffic mapping codes, e.g. [MV]
- **Garbage Collection**: Go has a garbage collector that automatically controls memory allocation and allocation, which can help reduce memory leaks and improve the overall performance of the simulation, [var]

By choosing this language, we kept the overhead of an already difficult concept of a program low while retaining its aspects in an intuitive manner.

We had written our initial versions of the simulation in Go but were encountering issues.

Firstly, and most importantly, the OpenMPI wrapper[1], was incomplete and unable to interface properly with the underlying OpenMPI C Library[2]. This was most visible when serializing data structs and encountering segmentation faults when working and developing with it. The latter, slowed development times down and was able to only slowly deliver results, if any.

Secondly, when a feasible solution was viable in our prototype, send and receive messages over MPI were empty on larger world sizes than 2. That indicated to us, that the underlying Go-Garbage collector freed crucial information from memory.

Since the aim of the project was to build a traffic simulator with MPI capabilities, we came to the conclusion to rewrite the program in Rust, although having an initially weaker concurrency model and harder-to-read syntax.

The most important reason is that Rust provides a working MPICH[3] wrapper[4]. Applying the structure and working of the existing Go code base was surprisingly straightforward because we were able to apply our previously won insights to the new prototype. The garbage-collection issues were resolved by the Rust language design because the language omits the need for a garbage collector. Furthermore, Rust has the side-effect of making the program faster, therefore allowing a more efficient use of available resources.

However, at a larger scale, we encountered a synchronization problem with using a mutex-mechanic, which will be explored later in the report.

To summarize, the initial challenges of using Go were successfully addressed by the rewrite in Rust.

When referring to source code, variables, etc. further down in the document it will be Rust, if not specified otherwise.

1.3 Overview of the report

In Section 2 we describe the process of acquiring the data and the preprocessing steps, as well as the tools used. We also discuss the potential usage of a graph and justify the reason for omitting it. Both the sequential and the parallel implementations are discussed in depth in Section 3. This includes the code structure of the projects as well as the algorithms. Section 4 consists of the analysis. This Section contains benchmarking, efficiency scaling based on resources, and speedup. Finally, the conclusion to the project and future work can be found in Section 5.

[1] `https://github.com/sbromberger/gompi`, Accessed: October 14, 2023, [Bro23]
[2] `https://www.open-mpi.org/`, Accessed: October 14, 2023
[3] `https://www.mpich.org/`, Accessed: October 14, 2023
[4] `https://github.com/rsmpi/rsmpi`, Accessed: October 14, 2023

2 Data, Tools and Preprocessing

The most essential element for our simulation is a map. Intuitively, we decided to represent the map as a graph structure early on. A subsequent decision that we needed to make was whether to use a constructed graph or real-life data. Because our aim is to simulate a realistic scenario. We found that OSM provided us with the necessary data.

2.1 OpenStreetMap

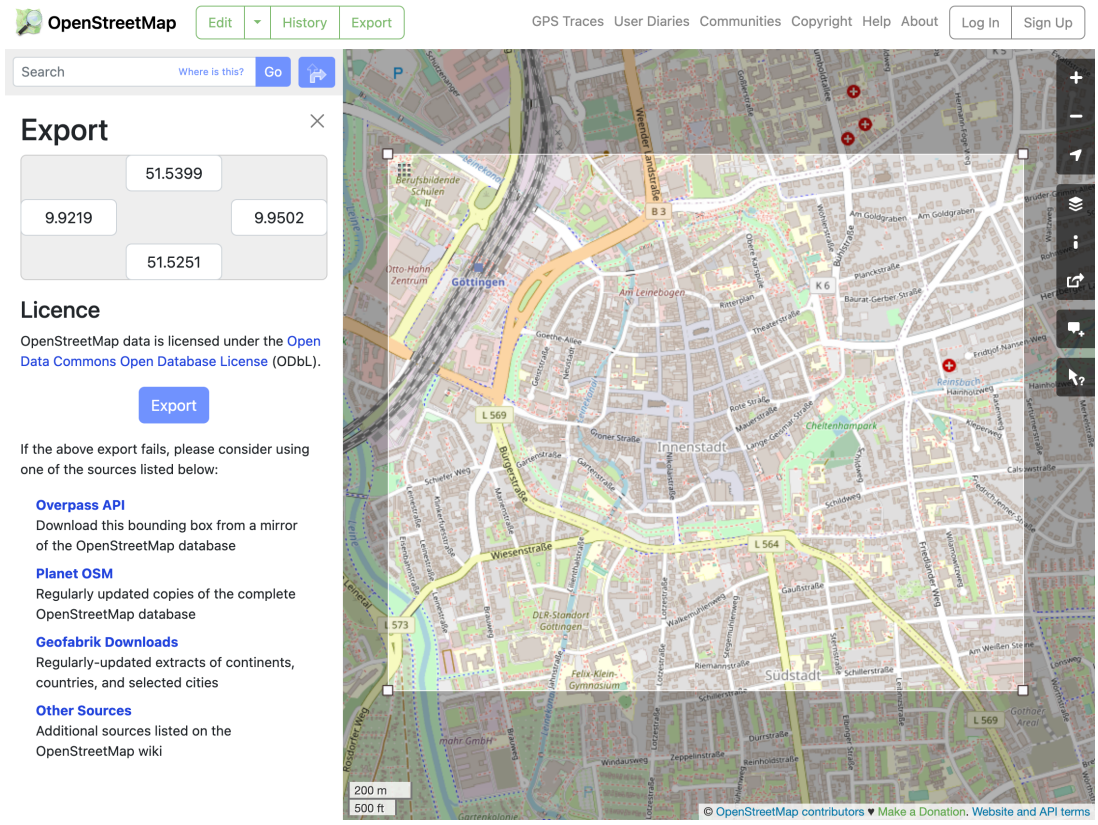

Figure 1: Displaying the export feature of OSM by extracting nodes from the city centre of Göttingen.

According to the official documentation, OSM is a crowdsourced project that accumulates and distributes open geographical data. It is an alternative to other map services like Google Maps, Bing Maps, and Apple Maps, just to name a few.

The OSM web app makes it possible to either manually select the area of interest or enter the desired longitude and latitude and simply download the necessary data as an `.osm` file. The file is represented in Extended Markup Language (XML). OSM follows a specific XML schema, which is already noted in the libraries.

In our case, the selected area was Göttingen city center, see Figure 1.

The data includes streets, roads, and other object markings that can be of geographical relevance, as in Figure 1

Furthermore, OSM references alternative Application Programming Interface (API) endpoints for exporting bigger areas of interest. This is needed because the OSM web app provides only small exports of data, limited to 50000 nodes. This may seem as much, but

is not, because of the nature of the object. The structure of a node is elaborated in-depth in Section 2.2.

2.2 GeoDataFramesand Graph Extraction

The acquired data was preprocessed in Python since the rich ecosystem offers a variety of data processing options. Those options also include libraries, such as OSMnx [5], that help parse the data obtained via the exported files.

Firstly, the imported data is converted to a MultiDiGraph object that represents a multidirectional graph. The transitional graph object is then mapped into two GeoDataFrames[6]. A GeoDataFrame is an extension of the Pandas DataFrame. Each extracted GeoDataFrame contains the edge and node data, respectively.
There are two types of nodes:

- **Object nodes** represent different objects one can find within the traffic area. Examples include various buildings, like houses or grocery stores, trees, and street lamps.
- **Intersection nodes** represent either the starting or ending point of one or more edges.

The GeoDataFrame representing the nodes consists of columns including, but not limited to: the corresponding `osmid` and geographical coordinates. An edge is defined as a connection between two intersection nodes. The GeoDataFrame holding the edge information, integrates columns such as `osmid`, source and target node, number of lanes, street name and type, maximal allowed speed, length, and more. Since the transitional graph object is a multidirectional graph, every two-way street is depicted as two separate edges between two corresponding nodes.

The object nodes are redundant for this project's purpose. The edges were filtered using the *highway* and *landuse* attributes, omitting cycleways, railways, and similar. Considering some of the edges' data was missing and the before-mentioned attributes were left empty, an OSM account was created and the labels were added manually.

It was at this point, that the decision against a visualization of the map along with the vehicles was made. The reason is, deeming the visualization outside the scope of this project and is non-critical to performing an analysis.

The aforementioned steps provide a solid foundation for creating viable input data for our simulation.

2.3 Graph

We initially considered RedisGraph[7] as a viable dependency in our simulation program. Several reasons led to this consideration.

- The RedisGraph database eliminates potential duplicates of edges and nodes, by design
- Redis[8] offers a pub-sub interface that enables a 'common ground truth' for all worker

[5] `https://osmnx.readthedocs.io/en/stable/`, Accessed: October 14, 2023, [OSM]
[6] `https://geopandas.org/en/stable/`, Accessed: October 14, 2023
[7] `https://docs.redis.com/latest/stack/deprecated-features/graph/`, Accessed: October 14, 2023
[8] `https://redis.io/commands/?group=pubsub`, Accessed: October 14, 2023

nodes

- Since Redis is intended to be used as a shared cache and for its performance, it seemed justified to have it as a permanent dependency in our simulator

However, since the requirements of the module are focused on the MPI capabilities, the pub-sub module and shared cashing features of Redis are unsuitable in this context. Therefore, the Redis instance is purely reduced to its graph database advantages. This is only used once in the data processing step, i.e. the RedisGraph database is only used once and in the initial step of producing input for our simulator.

Early versions of our simulation made use of the RedisGraph capabilities directly by querying the graph with Cypher[9]. This gave us an insight into the actual simulator import requirements.

Initially, we wrote a script for transforming the preprocessed `.osm` file to the RedisGraph data. Later on, it was decided that we would serialize the graph from the RedisGraph database to the JSON format. This is the input data representing the final graph in the simulator.

This venture was taken outside the simulation project into a separate open-source project, namely the *OSM-Map-Graph-Converter for OpenStreetMaps*.[MV23c]

It packages the preprocessing of the OSM export into a web server instance, allowing a generally directed graph conversion as described in Listing 1.

[9] `https://opencypher.org/`, Accessed: October 14, 2023

```
// vertex
{
  "x": 9.9268353,
  "y": 51.5331826,
  "osm_id": 28095800
}

// edge
{
  "from": 208650206,
  "to": 277409422,
  "length": 7.829,
  "max_speed": 30.0,
  "name": "Waageplatz",
  "osm_id": "24827765"
}

// Response
{
  "filename": "file.osm",
  "length": 187,
  "graph": {
    "vertices": [/*...*/],
    "edges": [/*...*/]
  }
}
```

Listing 1: Sample of a Directed Graph exported from OSM as JSON. Each vertex contains GPS coordinates and an `ID` given by OSM. The edges connect two vertices and hold additional data, representing a road or highway. The response part of the JSON holds the graph itself and the graph size with a file name.

3 Implementation

3.1 Project Structure

The main project, which is the rewritten code base of the initial Go-based project [MV23a] in Rust [MV23b], makes use of the project structure elaborated in Listing 2.

Starting with a general description of the project layout, we can find some language-specific configuration files, like the `Cargo.toml` and `Cargo.lock` which hold the majority of the meta-information for Rust, e.g. dependencies, authors, etc. [Fou] Also, there is a `README.md` which contains some general information about this repository.

The `assets/benchmarking` directory has files regarding the benchmarking process, which python scripts generate the foundation for our analysis in Section 4.

Other files in `assets/` help to bootstrap the simulation. the `compile.sh` file prepares the environment on the High-Performance Computer (HPC) frontend node and the

`graph.json` is the data extracted from the Göttingen city center as described in Section 2.

The business logic is contained by the `src/` directory, containing various Rust modules. The submodules `graph` and `models` represent data structures and input-models.

Most notably, the models represented in the `models` directory are `graph_input.rs` deserializing the JSON input data from the `graph.json` file, and the vehicle, explained in-depth in Section 3.2.

The data structure of the graph is contained in the `graph/` module. Housing the partition model in `rect.rs` and a directed graph of the deserialized OSM map in `osm_graph.rs`.

`main.rs`, `cli.rs`, `error.rs`, and `utils.rs` provide the simulation with basic functionalities, like random float generation, a CommandLine interface, and error specifications.

The heart of the simulation resides in `world.rs`, which puts all the algorithms, the main-loop, thread-communication, MPI message transfers, and graph bootstrapping together.

Lastly, the `vmpi.rs` file hosts project-specific MPI-critical code, which defines interfaces and methods of information exchange.

On how such an interface might look like, is depicted in listing 3.

3.2 Vehicle Struct

Vehicles are implemented as a struct.[10]

The `Vehicle`s, in addition to their functions, serve as the most important building block for the project. Each `Vehicle` struct is assigned a number of data fields. Their functionalities are explained in more detail in the Sections 3.3 and 3.4

- The `path_ids` variable is a list of node IDs that define the path the vehicle is going to take
- The `speed` indicates the current speed of the `Vehicle` in $\frac{m}{s}$
- The `is_parked` flag-variable serves as an indicator of whether a `Vehicle` has reached its destination, i.e. the last node in the `path_ids` list
- The `distance_remaining` keeps track of the remaining distance the `Vehicle` has to travel along the `edge`
- `prev_id` **and** `next_id` are needed to determine both the current and the next `edge` in the path; therefore indicating its location.
- Another data field `marked_for_deletion`, that serves as a flag-variable, is needed. This data field is relevant only in the context of the MPI approach
- The `steps` data field counts the number of times the `Vehicle` entered the function *`step`*

[10]Disclaimer for further reading: vehicle stylized `Vehicle` marks the struct called *vehicle*, whereas non-stylized version marks the mean of transport.

3.3 Sequential Approach

In both the sequential and MPI approach, the program requires a few parameters:

- `input_file`: The input file should contain the necessary graph data; a JSON file is expected
- `parallelism`: This sets whether a parallel approach is to be used or not, it can be used with or without MPI; either `SingleThreaded` or `MultiThreaded` value is expected, it defaults to `single-threaded`
- `num_vehicles`: The user inputs how many vehicles should be part of the simulation; an `int` is expected, it defaults to `100`
- `logging_level`: The possible levels are `Debug`, `Info`, `Warn`, and `Error`, it defaults to `Info`
- `thread_runtime`: Is only relevant if `parallelism` is set to be `True`. There are two options: `RustThreads` and `Tokio`, it defaults to `rust-threads`
- `mpi`: The user can choose a simulation that uses MPI. It is important to note that when using MPI `parallelism` cannot be set to `SingleThreaded`; a `bool` value is expected, it defaults to `false`; deactivating the MPI capabilities.
- `min_speed`: The user can set a minimal speed in meters per second for the vehicles, a `Float` value is expected; the default is `5.5`
- `max_speed`: The user can set a maximal speed in meters per second for the vehicles, a `Float` value is expected; the default is `8.5`

If we wish to run the sequential approach, the parameter `parallelism` must be set to `SingleThreaded` and `mpi` must be set to `false`. After those are set the first step is to build a graph from the input file. The graph is built by building the nodes, i.e. vertices and edges as structs. Following is the setup of the time-tracking mechanism to measure the time required for vehicles to reach their respective destinations.

Now, the `Vehicle`s can be constructed. Each `Vehicle` is assigned a random initial speed.

The bounds are adopted from the user input. In addition, a source and a destination node are also randomly picked for each `Vehicle`. The Depth-First-Search (DFS) algorithm generates a path between the set end nodes. In our opinion, it does not really matter what kind of algorithm one would pick, so we went with what we consider to be the most intuitive approach, or, the easiest to implement from the respective graph-library. It would be possible to exchange this algorithm in the future, or perhaps allow for a user to pick between multiple solutions.

Finally, we *drive*. The `Vehicles` drive-in steps. Each step represents one second in the simulation. A step can be one of the three:

- Drive
- Shift
- Park

Drive: A `Vehicle` drives when it has not yet reached its final destination or the end of the `edge` it is currently driving on. For a `Vehicle` taking a step in the form of *driving* means it is moved along the edge for a certain amount of meters. Naturally, the amount depends on the speed at which the `Vehicle` is set to be moving. As already mentioned in Section 3.2, the distance the `Vehicle` has left to travel on the current `edge` is marked in `distance_remaining`.
Shift: When a `Vehicle` has reached the end of the current `edge`, it must be moved to a neighboring `edge`. This step does not always happen at the exact *end* of the `edge`. Instead, before a step is taken, we check whether the `distance_remaining` is larger than the distance the `Vehicle` would travel within a second. If that is not the case, we save the remainder to `delta`. Now we move the `Vehicle` to the neighboring `edge` and simply add the value saved in `delta` to the total length of the `edge`. Now the `Vehicle` can proceed, taking steps in the form of *driving*. As for determining the neighboring `edge`, we use the `node`s saved in `prev_id` and `next_id`. Firstly, they must be updated by simply setting the next element in `path_ids`, respectively. When MPI is enabled, the Vehicle gets marked for deletion and is sent to the next node, which has the nodes in their paths in scope.
Park: This movement form is quite self-explanatory: when a `Vehicle` has reached its final destination, the variable `is_parked` must be set to `True`. Thereby notifying the root, that the current Vehicle is done driving.

All three movement types are managed in the function named `step`. The simplified overview of the explained functionality is depicted in Figure 2.

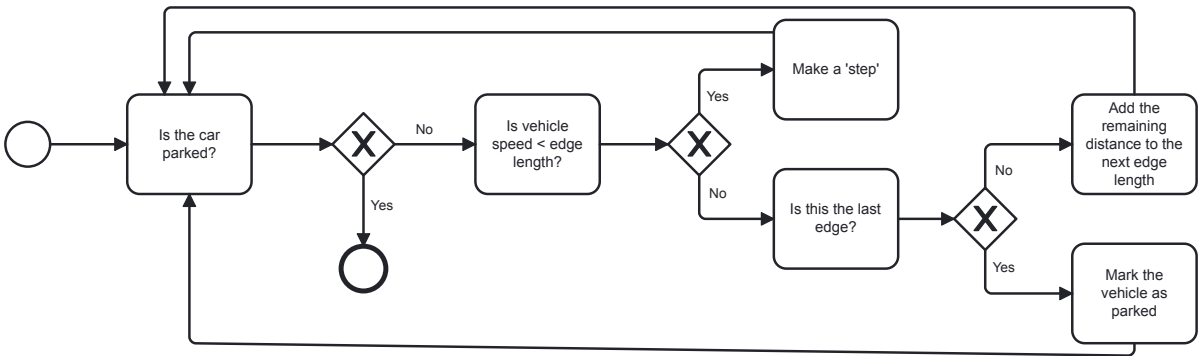

Figure 2: The basic workflow of the `step` function.

After testing the sequential approach, it became evident that there were some `Vehicles` with infeasible paths. Our suspicion was confirmed with further testing. This was the case when the source and destination `nodes` of the path were not connected in any way. We fixed this by picking new source and destination `nodes` and running the search algorithm until a feasible path was found.

At this point, we would like to discuss the driver model of the simulation. Firstly, a driver model determines how the vehicles move. There are two categories: microscopic and macroscopic driver models. The former of the two, as the name itself already implies, defines how each of the vehicles behave individually. This would mean, that each would have a separately defined speed, acceleration, deceleration, and the distance it keeps from a vehicle driving in front of it. Contrary to that, in a macroscopic model, all vehicles have either the same values for the mentioned parameters or some of the parameters are omitted. One would pick the microscopic model for a more realistic simulation. However,

since this was not our main goal for this particular project, we went with a macroscopic model. The vehicles only have different speeds at the start. This partially disappears when they are moved to an edge where the speed must be adjusted due to the speed limit. Acceleration, deceleration, and the distance concerning the vehicle in front are completely left out. Nonetheless, it is still possible to implement a microscopic driver model at a future time.

When the simulation is finished, it outputs the number of steps taken and the time it took for all `Vehicles` to finish driving. This concludes the sequential approach of the simulation.

3.4 MPI Approach

MPI stands for Message Passing Interface. It is used for parallelization, so the different processes can communicate and share the necessary data. The parallelization happens by having multiple processes that each have their tasks, that can mostly work concurrently.

Since the processes work concurrently, the efficiency of the program is expected to improve. More often than not, the processes use the same resources, so they must be able to communicate. To guarantee efficiency, it must be possible to address a specific process. Therefore, all processes have a rank.

In MPI we have a main process called the *root* process, and it has the rank 0, by default. The sub-processes, also referred to as *leaf* processes or tasks, have ranks ranging from 1 up to N-1, where N marks the total number of processes.

For the processes to be able to communicate, one uses objects called *communicators*. A communicator can either be an intra-communicator or an inter-communicator.

The former is a communicator that enables communication for processes within a single group. An inter-communicator makes it possible for processes that are part of different groups to communicate.

In our case, the MPI approach represents an extension of the sequential approach. To indicate to the program one wants to use MPI, the user must set `mpi` to `True` and `parallelization` to `True` as well.

Similarly to the sequential approach, a graph is built in the same way. The idea was to then split the graph into smaller sub-graphs, and have each of the `leaf` processes work on one sub-graph. This implies that the number of `leaf` processes equals the number of required sub-graphs. A user can set the desired number of `leaf` processes when setting up the MPI world size. The number of leafs corresponds to $size_{world} - 1$.

The process of splitting the graph into sub-graphs looks as follows:

1. Firstly, iterate through the vertices and find the bottom left and top right vertex. This defines the main rectangle in which all vertices are included. The root process monitors this main rectangle throughout the program.

2. This rectangle is then split into smaller partitions or sub-graphs, all of equal size. The split happens along the x-axis only. This means that the partitions maintain the same height as the main rectangle and the width is evenly divided by the number of `leaf` processes.

3. Finally, all `nodes` are copied to the sub-graph. Afterward, all `nodes` outside the bounds are removed from the scope of the `leaf` process. Note, that this means that the sets of `nodes` within each sub-graph are pairwise disjoint. As for the `edges`, they

are also copied from the main graph based on the leftover `nodes` in a sub-graph. This implies, that the `edges` defined by `nodes` in different partitions are only visible by the `root` process.

A visual example of how a graph partition might look is depicted in Figure 3. In this figure, the graph depicts the city center of Göttingen, and there are four `leaf` processes P1-P4.

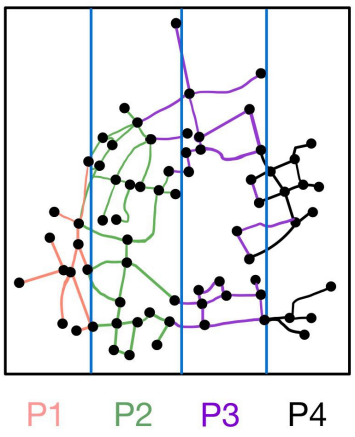

Figure 3: Example of a graph split into four sub-graphs with a designated `leaf` process.

After the sub-graphs have been completely set up, each `leaf` process gets assigned to one, individual sub-graph. Just like the sequential approach, the `Vehicles` are created, and a path is set up for each. Given the partitioning of the main graph, it is possible that the path of a vehicle stretches through multiple sub-graphs. To account for that, the `step` function was adjusted accordingly.

The three types of movement managed by the `step` function that we listed and explained in Section 3.3 still persist in this approach as well. *Drive* remained unchanged.

We kept the functionality of *Shift* intact. However, we extended it to account for the case in which the path of the `Vehicle` is not solely preserved within one sub-graph. The first change happens when the `Vehicle` has reached the end of the current `edge`.

Before the `Vehicle` can potentially be shifted to the next one, we check whether the following `node` in the `Vehicle`'s path is assigned to the current sub-graph.

If that is the case, we can simply make the switch to the next `edge` as described in the Section 3.3. When that is not the case, the following happens:

1. First and foremost, the remainder of the current edge length is stored in the `Delta` variable. Similarly to the sequential approach, an `edge` can have a remainder if the leftover length is smaller than the distance a `Vehicle` would travel in a second.

2. The length of the next edge, which is split between the partitions, is retrieved and added to the `Delta`. This unfolds by the `leaf` process sending an `EDGE_LENGTH_REQUEST` to the `root`. The `root` process then responds to the `leaf` that sent the request with the length of the `edge`. The status of the message containing the length is marked with the tag `EDGE_LENGTH_RESPONSE`.

3. After the length is added to `Delta`, the `Vehicle` is then serialized by the `leaf` and sent to the `root` process with the `LEAF_ROOT_VEHICLE` status tag. We briefly explored an alternative communication approach of assigning each node a lookup table. The table contains information for each vertex and its corresponding processing node. This way, each vehicle can be directly transferred to the corresponding node. Thereby omitting the workaround by getting proxied by the root node. However, this approach quickly leads to issues, like program panicking, or worse performance for a higher number of vehicles, > 100.

4. The last step of this process for the `leaf` process is to set the flag-variable `marked_for_deletion` to `True`. This variable indicates that the `leaf` process is not responsible for that `Vehicle` anymore. It is important to note, that the `Vehicle` is not parked, i.e. is still not at its destination. However, the calculations have been handed over to another subprocess.

5. The `root` process must now retrieve the rank of the `leaf` process, which manages the sub-graph that contains the following `node` in the path. Then, it forwards the serialized `Vehicle` to the according `leaf` process with the status tag `ROOT_LEAF_VEHICLE`.

6. The `leaf` process, that receives the message, deserializes the `Vehicle` it has received within the message. Afterward, it updates the `next_id` and `prev_id`. This step ensures the `leaf` process works with an `edge` that is entirely contained within the bounds of the sub-graph.

7. Now, the `Delta` that contains the length of the `edge` that is split between the sub-graph and the remainder of the previously completed `edge`, can be added to the new `edge` from the perspective of the vehicle. This way, the `Vehicle` does not drive in between partitions, and we ensured that the whole distance is driven.

8. Finally, the `Delta` can be set to zero and the `Vehicle` continues to drive like it is described in the Section 3.3.

The implementation is summarized in Listing 5.

As for *Park*, an additional functionality has been added.

After a `Vehicle` has been *parked*, the `leaf` process sends a message to the `root` with the status tag `LEAF_ROOT_VEHICLE_FINISH`. This signals to the `root`, that a `Vehicle` has reached its destination. The `root` process keeps a counter of `Vehicles` that have reached their destination. This is triggered by receiving this message.

When the number of `Vehicles` that have reached its destination matches that of the number of created `Vehicles` the `root` process broadcasts a message with the status tag `ROOT_LEAF_TERMINATE` to all `leaf` processes. This notifies the processes, they are to terminate.

Finally, the `root` ends the `mpi` altogether.

A simplified workflow of what was described is depicted in Figure 4.

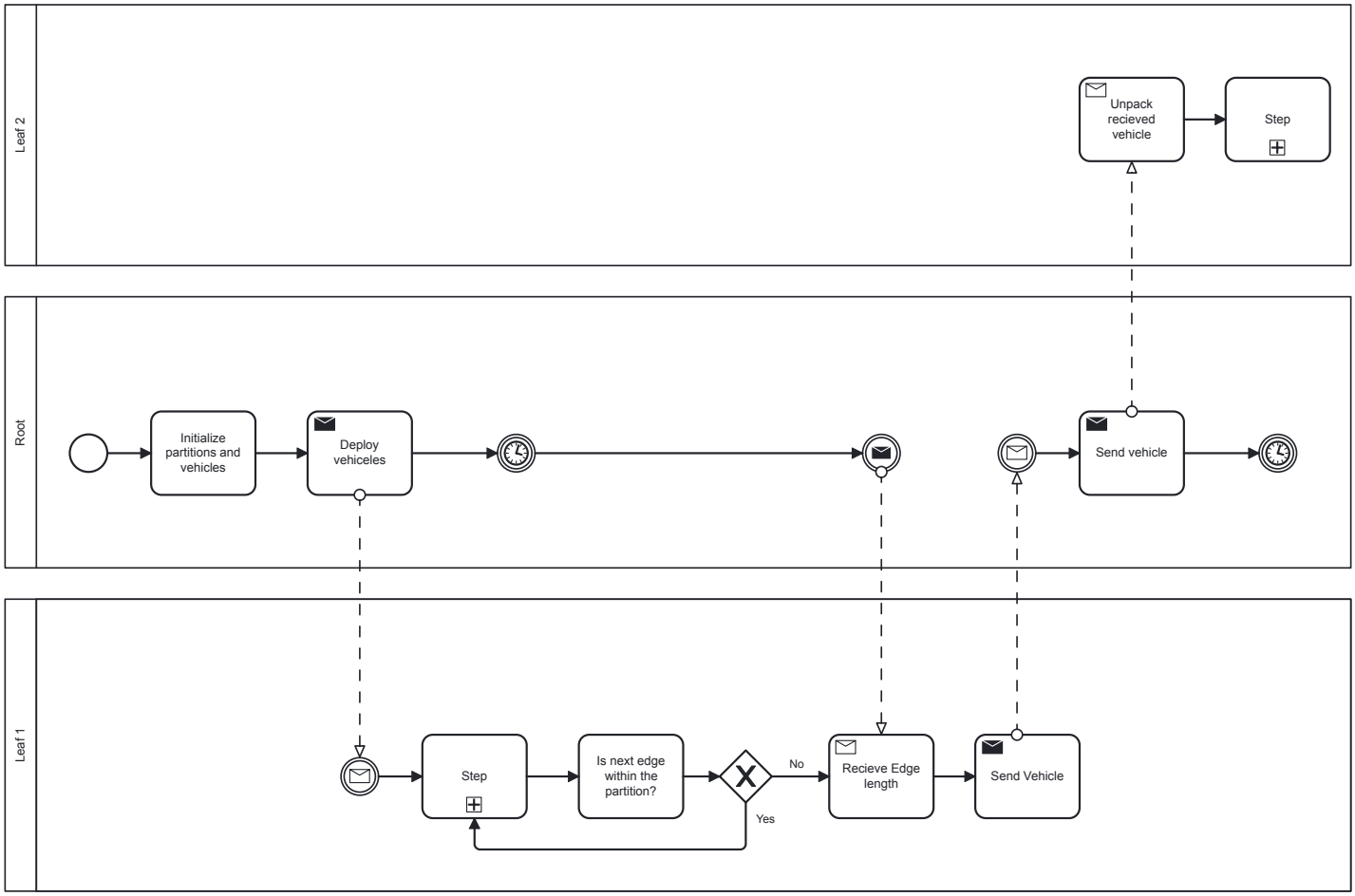

Figure 4: Basic workflow of moving the vehicle to another leaf process. Only one transition is depicted, solely to facilitate a clear understanding of the process.

As for the communicator objects, only one has been used, and it is an inter-process-communicator since the `leaf` processes were not grouped and treated as separate entities.

After some testing, it became clear that when partitioning the main graph, there were some vertices left that did not belong to any of the sub-graphs. It was assumed that this error was due to inaccurate numeric precision, i.e. rounding errors of the vertex coordinates. One solution was to add an additional width-padding for the sub-graphs. However, while this reduced the number of unallocated vertices, it did not completely

solve the issue. Therefore, we designed a mechanism to add the unassigned vertices to the first partition. This can be optimized by designing an assigning algorithm that either randomizes the target partition or assigns the node to the nearest sub-graph.

4 Analysis

4.1 Benchmarking

For benchmarking, we used the following parameters:

- Number of vehicles: 1, 10, 100, 1000 and 10000
- Number of nodes/processes: 1, 2, 3, 4, 5 and 8
- Number of cores/workers: 2, 4, 8, 16, 32 and 48
- Complex computation: included and excluded

Number of vehicles: Benchmarks containing 10000 vehicles were omitted from the analysis. This is due to the high error rate. The error rates for each of the vehicle number options are listed in Table 1. An error is defined by finding a `panic`-message in the log for a specific run or a non-zero exit code.

Number of nodes: There were also some additional node options: 6, 10, 15, and 19. We found that they were not particularly useful in the analysis, so they were omitted.

Number of cores: For the same reason as the omitted node values and 100% error rates, the following core values were omitted as well: 96, 144, and 192.

Complex computation: A complex computation is added as a placeholder to the code, see listing 6.

Currently, the code does not include any particularly use-case-relevant complex computation.

However, at some point in the future, if, for example, a visual simulation or a microscopic driver model is implemented, this would impact the complexity of the program. Therefore, we decided to take it into account for the analysis, see listing 6.

Vehicles	Error Rate
1	33%
10	28%
100	28%
1000	52%
10000	86%

Table 1: Error rate per number of vehicles

4.2 Strong Scaling

To test the efficiency of the parallelization, we compared the average runtimes for different core and node options. As for the vehicle number, we set it to 100. The number seemed reasonably big for our particular graph, without compromising the correctness. As can be seen in Table 1, the next bigger value for the vehicles is 1000 and has an error rate of 52%. Because of the high error rate, we deemed 100 vehicles the best option.

To test the efficiency based on the increasing number of nodes - in addition to vehicles - we also had to decide on a fixed number of cores.

Similarly, testing efficiency based on the growing number of cores requires a fixed number of nodes.

To decide which values to pick, we plotted a box plot depicting different combinations. The resulting plot is depicted in Figure 5.

We can see, that for the node-based efficiency, we must pick a relatively higher number of cores, as there is more data for bigger core numbers. This is because the node number must be lower than the core number, for the parallelization to make sense. The whiskers on each side of the boxes denote the variability of the minimum and maximum. Therefore, we decided to go with 8 cores, since they produced the most reliable data.

As for the core-based efficiency, we stuck with 2 nodes. The reasoning is similar to before.

It can be seen in the Figure, that the 2-node configuration has a good amount of data in comparison to other ones, and it is by far the most stable configuration as well.

Starting with the core-based efficiency, in Table 2 we listed the impact of different numbers of cores on the average runtime. The number of cores ranges from 2 to 48. As can be seen from the Table, it is clear that the average runtime does improve significantly by adding more cores.

Moreover, the visualization in Figure 6 shows the data comes pretty close to the optimum that depicts an exponential decay curve. The drop to 0.0 for 4 cores is a result of an error and, therefore, missing data.

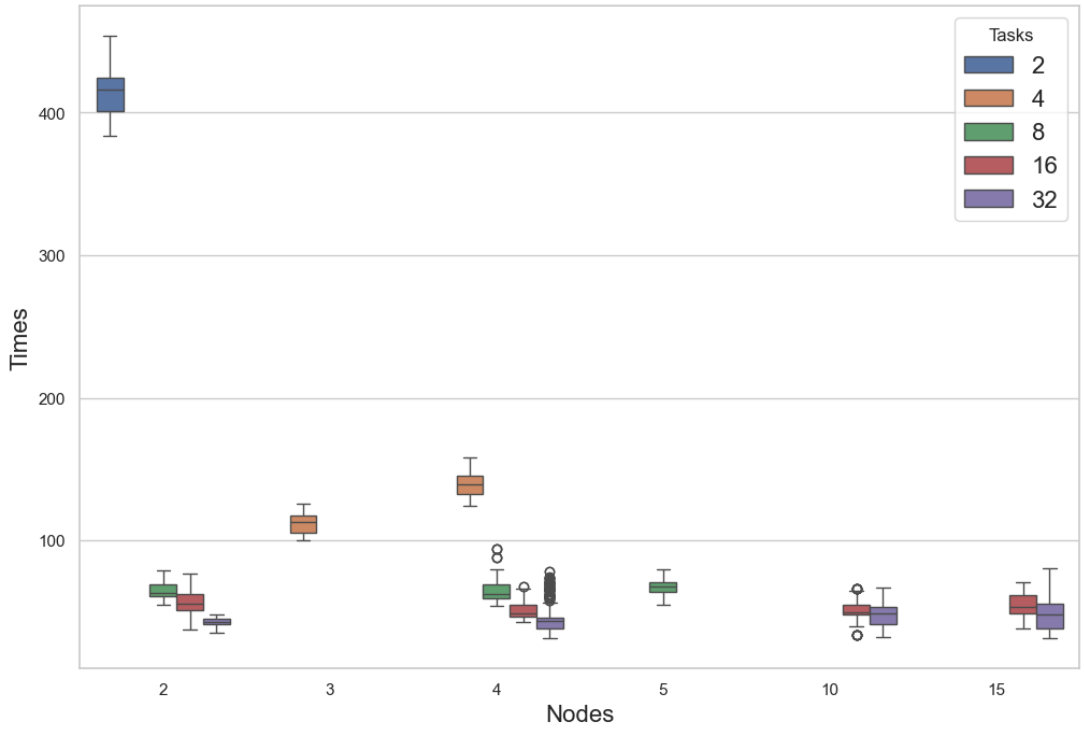

Figure 5: Box-plot showing the time for 100 vehicles for different node and core combinations.

Similarly, for the node-based efficiency, while the number of cores is set to 8, the number of nodes ranges from 1 to 5. The values listed in Table 3 show the similarity of values for 2, 4, and 5 nodes.

Although the value for 3 nodes is missing, also due to an error, we can safely assume that it would also be in the mid-60s value range.

Vehicles	Nodes	Cores	Average time in s
100	2	2	416.9
100	2	4	0.0
100	2	8	65.2
100	2	16	56.5
100	2	32	42.8
100	2	48	31.1

Table 2: Impact of different numbers of cores on the runtime for 100 vehicles and 2 nodes.

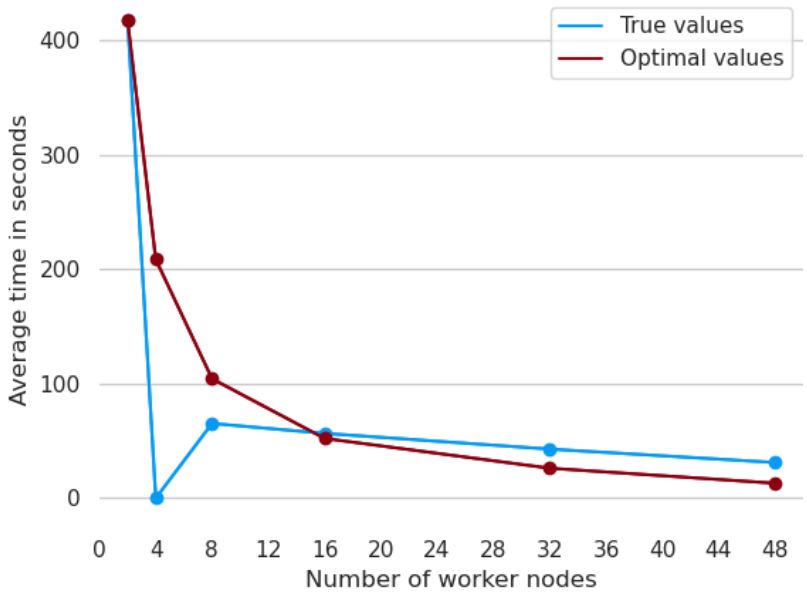

Figure 6: The efficiency for 2 nodes and different core values.

This data is also visualized in the Figure 8. In this Figure, if we were to eliminate the drop at the 3-node mark and just directly connect the data from 2 to 4 nodes, it becomes clear that the increase of nodes past 2 does not change the average time significantly.

Merging the node and core efficiency results, we conclude that the optimal number of nodes and cores is 2 and 8 respectively.

We did expect the increase in cores to be more efficient than the increase in nodes; however, not this drastic. We assume that the reason for this behaviour could be the partitioning of the graph. The program is parallelizable on a core/task level because the more cores we have, the number of vehicles each core handles decreases.

On the other hand, the number of nodes is proportionate to the number of partitions of the main graph. Since the program goes heavy on message passing, this is a problem, especially given the smaller size of the graph.

If the main graph were bigger, we assume, that increasing the number of nodes could have a somewhat bigger impact on the average times.

However, this improvement would probably not be as significant as reducing the message overhead. We have two ideas on how this could be done:

1. A different way of partitioning the graph: instead of splitting the graph vertically,

a better option would be to find K nearest neighbours. This could minimize the number of times a vehicle leaves the current sub-graph and enters another one. A possible partition of the described approach is illustrated in Figure 7.

2. A different architecture: instead of each leaf process handling the calculations on a particular sub-graph, the leaf processes handle a number of vehicles on their complete route. This could also lead to some imbalances in the case that some processes handle a bigger number of vehicles driving a shorter path and others handling vehicles with predominately longer paths. However, this would probably not be worse, but it needs to be tested. Due to the time constraints, this will be part of the future work.

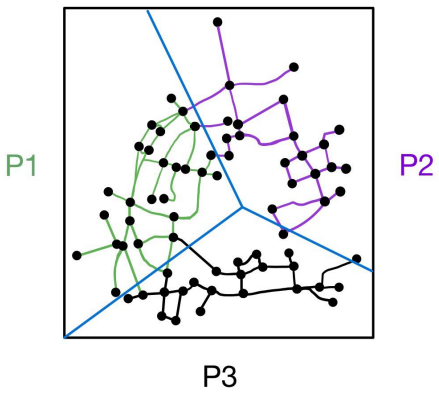

Figure 7: Graph-partition with K nearest neighbours.

Vehicles	Cores	Nodes	Average time in s
100	8	1	123.7
100	8	2	65.2
100	8	3	0.0
100	8	4	65.2
100	8	5	67.4

Table 3: Impact of different numbers of nodes on the runtime for 100 vehicles and 8 cores.

4.3 Speedup

Another analysis we did was the speedup. We calculated the average time for the sequential approach and compared it to the average time for one node but with multiple cores. As depicted in Figure 9 and also listed in 4, we can see that the values stay within the same range, and get even a bit worse. This would imply that the code is just not parallelizable. However, this is not the case when we run the code on a MacBook Pro 2021, Apple M1 Max with 32 GB RAM. As the numbers in Table 5 suggest, the speedup works very well. We are unsure what the reason for this misalignment is, but we assume it could be due to the scheduler on the HPC and different architectures, the M1 Pro processor operating on ARM.[app]

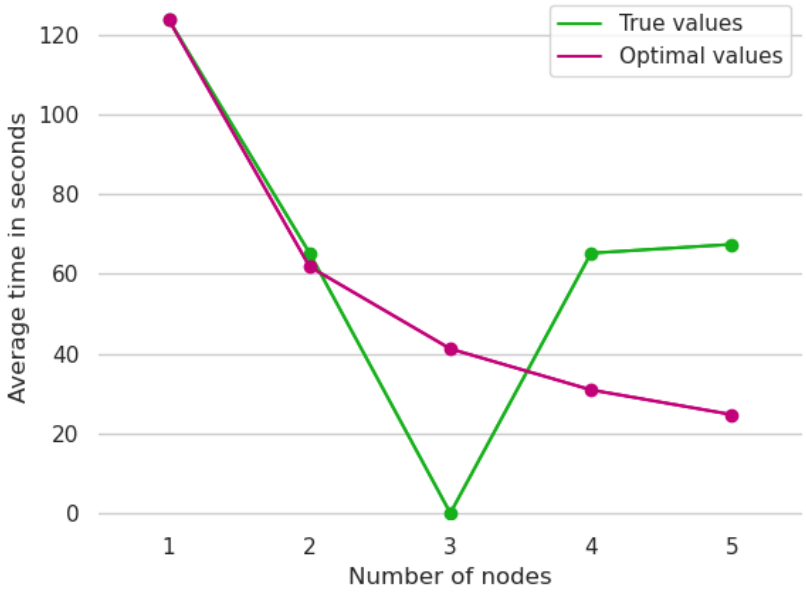

Figure 8: The efficiency for 8 cores and different node values.

Vehicles	Nodes	Cores	Average time in s	Speedup
100	1	1	122.2	1.000
100	1	2	123.5	0.989
100	1	4	124.4	0.982
100	1	8	123.7	0.988
100	1	16	127.8	0.956
100	1	24	123.4	0.990

Table 4: The speedup on an HPC for 100 vehicles

Vehicles	Nodes	Cores	Average time in s	Speedup
100	1	1	101.4	1.000
100	1	2	49.7	2.040
100	1	4	26.2	3.870
100	1	8	13.6	7.456

Table 5: The speedup on a MacBook Pro 2021, Apple M1 Max, 32 GB RAM for 100 vehicles.

4.4 Weak Scaling

Weak scaling is also an important part of the parallelization analysis.

However, due to the high error rates for the data containing 1000 and 10000 vehicles, it was not possible to produce a meaningful analysis. In Figure 11 we can see the comparison of the average time for 100 and 1000 vehicles, for different core options. The trend does look promising, however, as already stated, due to the errors and missing data, we can not make any certain claims.

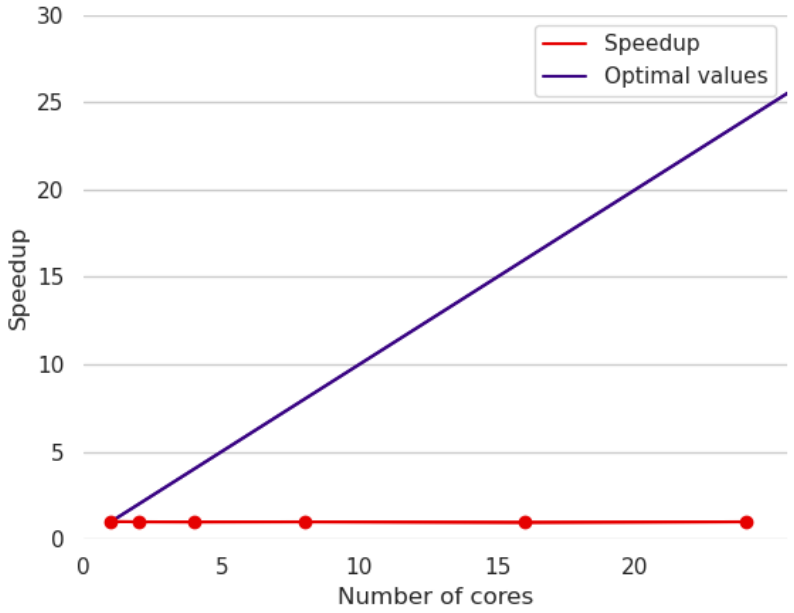

Figure 9: The speedup on an HPC for 100 vehicles.

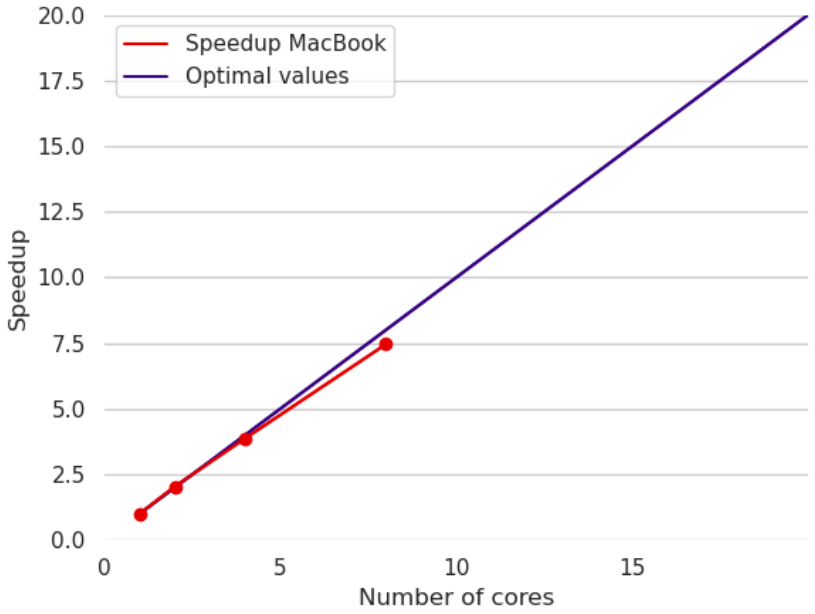

Figure 10: The speedup on a MacBook Pro 2021, Apple M1 Max, 32 GB RAM for 100 vehicles.

5 Conclusion

5.1 Summary

In this report, we implemented a vehicle simulation in a sequential and parallel manner. As already mentioned, we initially wrote the project in Go.[MV23a]

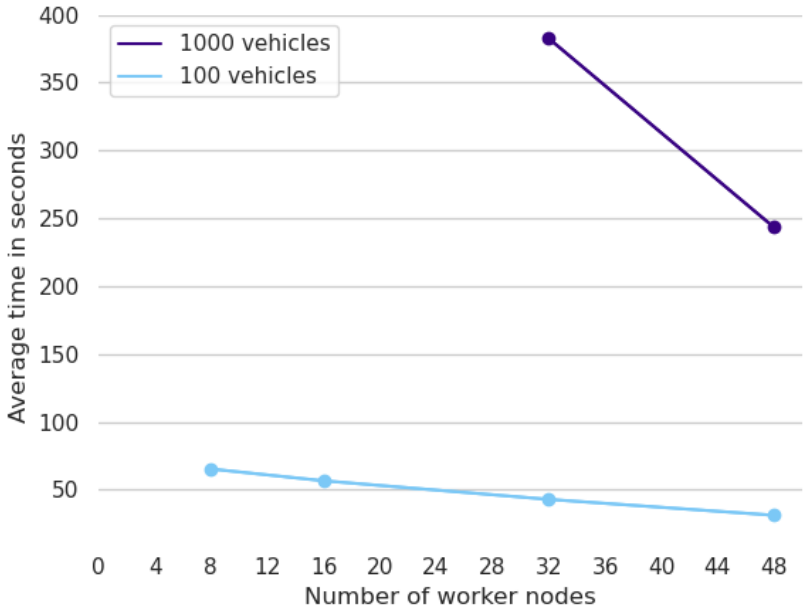

Figure 11: Average time comparison for 100 and 1000 vehicles and different core amounts.

We started off by retrieving and processing the necessary data to emulate a real-life street map.

We defined and implemented the vehicle `Vehicle struct`.

After this, it was time to think about the program flow for the sequential approach. This approach was rewritten several times due to problems with inadequate Go-wrappers, [Bro23].

Finally, we run some benchmarks, to get an idea of where we are currently at, performance-wise.

It was at this point that we had a presentation where we shared the state of the work and current results with other course participants and supervisors. This was a good opportunity to get some feedback as well as reassurance.

After the presentation, we proceeded with the design and implementation of the MPI approach. The main functionality within the `Step` function required multiple adjustments. The adjustments also impacted the flow of the sequential approach and required some additional rewriting.

When the implementation of the MPI approach was nearing the end, it became evident, that it would not be possible to finish the program with meaningful results.

This was due to the before mentioned inadequate Go-wrapper, [Bro23], in combination with Go's native scheduling procedures, that made it impossible for the MPI to work.

Therefore, we decided to rewrite the entire project in Rust, [MV23b]. This worked without any major issues and we quickly had a running, faster, and capable project.

We could now benchmark the different setups and analyse the results.

The analysis showed that the program is fairly parallelizable; however, it does depend on different input parameters. It also became evident, that the graph-partitioning should perhaps have been done somewhat differently.

5.2 Future work

As already mentioned, throughout the report there were several spots where we noticed room for improvement and possibilities to expand this project.

Improvements include:

- Different graph partition, such as a k nearest neighbours as depicted in Figure 7. This would make the code more efficient
- Changing the MPI approach architecture. The current architecture builds on the graph partition, however, in our opinion, an architecture that is based on vehicles could be more efficient for parallelization. What we mean by that, is for the leaf processes to handle a subset of vehicles and manage calculation for the entire drive
- Microscopic driver model implementation to make the simulation more realistic
- Synchronization of the vehicles
- Fine-tuning the *Shift* movement within the `Step` function. Currently, this happens within one, separate step. If we were to introduce a synchronization of the vehicles, the solution must be somewhat more seamless, and perhaps, majorly rewritten
- Visualization of the simulation

References

[app] apple. *Apple Unveils All-New iPad Air with A14 Bionic, Apple's Most Advanced Chip - Apple*. URL: https://www.apple.com/newsroom/2020/09/apple-unveils-all-new-ipad-air-with-a14-bionic-apples-most-advanced-chip/ (visited on 10/14/2023).

[Bro23] Seth Bromberger. *Sbromberger/Gompi*. Sept. 2023. URL: https://github.com/sbromberger/gompi (visited on 10/14/2023).

[Fou] The Rust Foundation. *The Manifest Format - The Cargo Book*. URL: https://doc.rust-lang.org/cargo/reference/manifest.html (visited on 10/14/2023).

[Map] Open Street Maps. *OpenStreetMap*. URL: https://www.openstreetmap.org/ (visited on 07/10/2023).

[MV] Valerius Mattfeld and Bianca Vetter. *Github - PCHPC - Graph*. URL: https://github.com/valerius21/pchpc/blob/main/streets/redisInfo.go#L33-L44 (visited on 07/10/2023).

[MV23a] Valerius Mattfeld and Bianca Vetter. *MPI Traffic Simulation in Go*. Version report-0. Oct. 2023. URL: https://github.com/valerius21/mpi-traffic-simulation-in-go.

[MV23b] Valerius Mattfeld and Bianca Vetter. *MPI Traffic Simulator in Rust*. Version 0.1.0. Oct. 2023. URL: https://github.com/valerius21/mpi-traffic-sim-rust.

[MV23c] Valerius Mattfeld and Bianca Vetter. *OSM Map-to-Graph Converter*. Version 0.0.0. July 2023. URL: https://github.com/valerius21/OSM-Map-Graph-Converter.

[OSM] OSMnx. *OSMnx 1.5.1 Documentation*. URL: https://osmnx.readthedocs.io/en/stable/ (visited on 07/10/2023).

[var] various. *The Go Memory Model*. URL: https://go.dev/ref/mem (visited on 07/10/2023).

A Work sharing

Throughout the entire project, work was distributed mainly, yet not exclusively, as described, with both parties having a role in various steps.

A.1 Valerius Mattfeld

- Go implementation
- Rust implementation
- Go to Rust rewrite
- Code Architecture
- Feature implementations, incl. MPI.
- Benchmarking
- Benchmark data processing

A.2 Bianca Vetter

- Research
- Data Preprocessing
- Pair-programming
- Algorithm-design
- Performance Analysis

B Code samples

```
mpi-traffic-sim-rust
├── Cargo.toml
├── Cargo.lock
├── README.md
├── assets
│   ├── compile.sh
│   ├── graph.json
│   └── benchmarking
│       └── scripts
│           ├── cc_generate_batch_loads.py
│           ├── cc_single-threaded.py
│           ├── cc_single-noded.py
│           ├── cc_generate_batch_loads_optimized.py
│           ├── cc_multi-threaded.py
│           ├── generate_batch_loads.py
│           └── generate_batch_loads_optimized.py
└── src
    ├── error.rs
    ├── world.rs
    ├── vmpi.rs
    ├── main.rs
    ├── prelude.rs
    ├── utils.rs
    ├── cli.rs
    ├── graph
    │   ├── rect.rs
    │   ├── mod.rs
    │   └── osm_graph.rs
    └── models
        ├── graph_input.rs
        ├── vehicle.rs
        ├── mod.rs
        └── vehicle_builder.rs
```

Listing 2: Project Structure of the traffic simulator, [MV23b].

```rust
impl MpiMessageContent<EdgeLengthRequest> for EdgeLengthRequest {
    fn to_bytes(data: EdgeLengthRequest) -> Result<Vec<u8>> {
        Ok(serialize(&data)?)
    }

    fn from_bytes(data: Vec<u8>) -> Result<EdgeLengthRequest> {
        Ok(deserialize(&data)?)
    }
}
```

Listing 3: MPI message exchange interface

```rust
/// Represents a vehicle that can move within a graph.
#[derive(Debug, Serialize, Deserialize)]
pub struct Vehicle {
    pub id: String,
    pub path_ids: Vec<Osmid>,
    pub speed: f64,
    pub delta: f64,
    pub next_id: Osmid,
    pub prev_id: Osmid,
    pub is_parked: bool,
    pub distance_remaining: f64,
    pub marked_for_deletion: bool,
    pub steps: u64,
}
```

Listing 4: Shortened version of the `Vehicle` struct from `vehicle.rs`

```rust
// Root to leaf vehicle sending tag
pub const ROOT_LEAF_VEHICLE: i32 = 1;

// Leaf to root vehicle sending tag
pub const LEAF_ROOT_VEHICLE: i32 = 2;

// Leaf asks root for edge length
pub const EDGE_LENGTH_REQUEST: i32 = 3;

// Root responds to leaf with edge length
pub const EDGE_LENGTH_RESPONSE: i32 = 4;

// Leaf to Root vehicle finishing notification
pub const LEAF_ROOT_VEHICLE_FINISH: i32 = 5;

// Root to leaf program termination notification
pub const ROOT_LEAF_TERMINATE: i32 = 6;
```

Listing 5: Leaf-root-leaf communication tags as found in `vmpi.rs`

```rust
    fn calculate_step(&mut self) {
        // NOTE: adding CPU-intensive placeholder by
        // generating a random prime number
        #[cfg(feature = "complex-calculation")]
        {
            let mut rng = rand::thread_rng();
            let number = rand::Rng::gen_range(&mut rng, 1_000_000..=3_000_000);
            let _some_unused_prime = primal::Primes::all().nth(number).unwrap();
        }
        // WARN: Actually crucial code. Do not remove.
        self.distance_remaining -= self.speed;
    }
```

Listing 6: Extendability location for the vehicle behavior.