Project Report: Streets4MPI

Julian Fietkau Joachim Nitschke

May 21, 2012

Project report for the course

“Parallel Programming”

Summer semester 2011 to winter semester 2011/12

Supervised by Julian Kunkel

Department of Informatics
University of Hamburg

Contents

Abstract

Streets4MPI is a street traffic simulation software that uses MPI and is written in Python. This report details how it was created, how it works, what it can do as well as what it might be able to do in the future, with a focus on run time evaluations and improvements.

1. Introduction

In April 2011, the authors of this document enrolled in the *Parallel Programming* project course. The main task in that course was to design a program to solve a non-trivial parallel processing problem. That was when the idea for *Streets4MPI* first came to light.

The basic idea of *Streets4MPI* is to pick a fixed amount of (start, destination) pairs in a street network. For each of these pairs, the shortest path is calculated, taking local speed limits into account. All calculated shortest paths are then traversed and the traffic load is recorded for each street, the cumulated results constituting a simulated day. From day two onwards, drivers are influenced to a semi-random extent by street congestions from the previous day. After a configurable number of days has passed, the street network is adapted: seldomly used roads are dismantled, heavily used ones are augmented to better cope with their usage.

Apart from the simulation core, Streets4MPI also contains a visualization component that can output the simulation results as traffic load heatmaps.

1.1. Project Task

The overall goal of the project was to create a successfully parallelized software application that uses parallel processing in a nontrivial manner. *Streets4MPI* fits this description, since the traffic load calculation involves a lot of synchronization between the processes, which makes parts of the algorithm very hard to parallelize, as we will see in later chapters.

By extension, *Streets4MPI* also acts as an evaluation of whether a computationally intensive parallel application like this is feasible to be written in Python. It would be interesting to see if a good parallelization efficiency is achievable.

2. Traffic Simulation

In this chapter we describe the mechanisms that our traffic simulation is built on. To be useful in a real-world city planning scenario a traffic simulation software would have to model a multitude of different influences on traffic behavior like day time, traffic lights, construction sites or right of way. Due to limited time, we focus on a more macroscopic perspective with a few selected core rules.

2.1. Discrete Macroscopic Simulation

We assume that a street network is given as a directed graph, where the edges and nodes represent the streets and crossings. To populate the network each node is assigned a static number of residents. Inspired by the work on interactive geometric city simulation by Weber et al. [2009] we further abstract from individual cars to a more general traffic model based on trips. A trip represents the main travel done by a resident over a certain time period, e.g. his way to work every day. Trips start at a designated street and end at another. Start and destination are connected via the shortest path that is in our case based on driving time. The driving time depends on speed limits and slowdown due to traffic load.

We could think of two types of drivers: The first one chooses his route dynamically in real-time. He just starts driving and takes another route if he sees a traffic jam. The second one plans his route statically in discrete steps. If he gets stuck in a traffic jam he sits it out but takes another route the next day. Real drivers are probably a mix of both types. However a parallel version of the first approach is way more complicated to implement since it requires sophisticated synchronisation methods.

Therefore our simulation runs in discrete steps. During each simulation step we calculate the traffic load and – based on that – adapt the traffic in the next step. To determine the traffic load we first calculate the trips' shortest paths and then set each street's traffic load value to the number of paths using this street. At the beginning of the next simulation step these results are used to update every street's driving time respectively the edge weights in the street network graph. The exact calculation of the driving time is described in the next section.

Our simplified approach has some implied limitations: Due to the trip based model our drivers only drive to one single destination every day. At first this seems to be relatively unrealistic. However if one sees all the trips starting at one node as an approximation for the total traffic caused by the residents living there, multiple destinations are in fact included. Another limitation comes with the discrete route planing. We assume that every driver knows the complete traffic state from the previous simulation step. Based on that he makes a decision which is independant from all the other drivers. In fact there is one resulting problem which we handle later in section 2.3. At first we will take a closer look at the driving speed calculation.

2.2. Braking Distance Based Driving Speed Calculation

Increasing the driving time with respect to the traffic load is one of the core mechanisms in our simulation. We use a function that is based upon an assumption: Our virtual drivers always keep safe braking distances. Given a typical braking deceleration a_{brake} we can calculate a maximum driving speed v that still ensures a full stop within a given braking distance l_{brake}:

$$v = \sqrt{a_{brake} \cdot 2l_{brake}}$$

Then – given a typical car length l_{car} – the available braking distance depends on the length of a street l_{street} and the number of cars driving over it at that time:

$$l_{brake} = \frac{l_{street}}{n_{cars}} - l_{car}$$

To ensure the result is always greater than zero, we include a minimum braking distance. We set this value close to zero to decrease the driving speed as much as possible since the street is completely blocked in the case of a negative available braking distance.

At this point we already have a relatively coherent model. But as we pointed out before, our trip based concept abstracts from individual cars. So we still need a mapping from the trips using a street per simulation step to a number of cars on the street at a certain point in time. We already described that a trip represents a resident's traffic over a certain time period, e.g. a day or 8 hours for day time only. So what is left is to distribute all the trips using a street over a given time period Δt:

$$n_{cars} = n_{trips} \cdot \frac{t_{trip}}{\Delta t} = \frac{n_{trips} \cdot l_{street}}{v \cdot \Delta t}$$

We further add a trip volume $n_{cars/trip}$ to model that our residents do more than one ride during a day:

$$n_{cars} = \frac{n_{trips} \cdot n_{cars/trip} \cdot l_{street}}{v \cdot \Delta t}$$

Note that now one can switch between a better performance or a more detailed simulation: By decreasing the number of trips and increasing every trip's volume, the simulation gets faster but less detailed since less destinations are chosen. On the other hand by increasing the number of trips and decreasing the trip volume, the simulation produces more detailed results with the cost of more computation complexity.

Combining the given formulae together we can now calculate the driving speed. Our virtual drivers always respect speed limits, therefore the actual driving speed v_{actual} is the minimum of the calculated driving speed v and a street specific speed limit v_{max}.

2.3. Preventing Oscillation with Randomness

In reality one can observe that during rush hour some streets are always blocked. Some people seem to be uninformed or naive enough to still take these routes although they are blocked every day. In our simulation on the other hand the drivers are wiser. If one route is blocked on one day they will take another faster route on the next day. The problem about this strategy is that every driver makes his decision independently and on the same basis. This leads to an oscillating behaviour: One day everybody takes route A, on the next day everybody switches to route B since it is now supposedly faster than route A. Figure 1 shows an example.

A more realistic scenario would be route A always beeing blocked while a part of the traffic spills over to route B. To achieve this we focused on the psychological aspect of the problem: Some drivers seem to be more tolerant when it comes to the weighing of traffic jams. To model this we introduced a random traffic jam tolerance t_{jam} for each

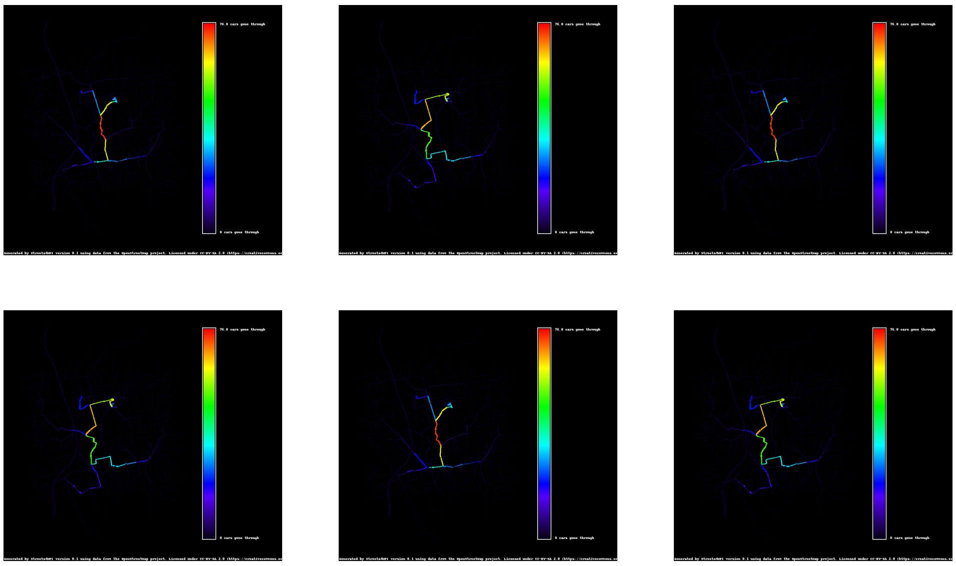


Figure 1: This sequence of images (read from left to right, top to bottom) shows how the independant route planning causes oscillating driving behavior: On the first day, the central road is completely jammed. The next day, all drivers avoid it, only to return to it one day later, and so on.

resident. This value varies between 0 and 1 and weighs the impact of the traffic load on the driving speed. We receive a perceived driving speed $v_{perceived}$:

$$v_{perceived} = v_{actual} + (v_{max} - v_{actual}) \cdot t_{jam}$$

If the traffic jam tolerance is high the driver plans his route based mainly on the ideal driving speed respectively the speed limit. On the other hand if the traffic jam tolerance is low the traffic load's impact is taken into greater account.

Although this is a quite simple solution for the oscillation problem the results are in fact satisfying. Figure 2 shows an example where a main route is always blocked while the changes in traffic happen mostly on peripheral streets.

2.4. Street Network Adaptation

In Weber et al. [2009] the traffic simulation is used to simulate a city's development over longer time periods like months or years. New streets are built, resident's move to different places, buildings are constructed and torn down again. Due to our simplified model and limited time we weren't able to implement such a complex simulation but decided to focus on one of the few static attributes that influence our traffic: the streets' speed limits.

We further assume that speed limits are mainly changed to in- or decrease a street's capacity. If a street is used very frequently it is widened and its speed limit is increased. On the other hand if a street is rarely used it is shrinked and its speed limit is decreased.

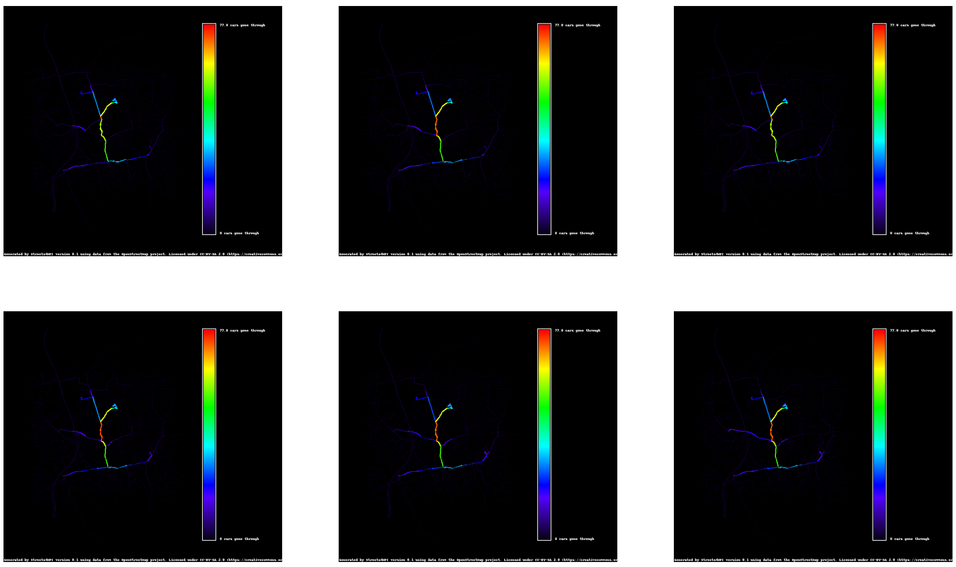


Figure 2: This sequence of images (read from left to right, top to bottom) shows how the artificial randomness in the traffic jam resistance factor causes some drivers to stick to a road even if it was jammed the day before, creating a bigger chance for the model to stabilize.

To achieve this we sum up the traffic load that is calculated during n simulation steps. Then – every n^{th} simulation step – we change the speed limit of the most and least used streets as described before.

By focussing on traffic capacity we exclude other factors that play a role in urban planning like noise disturbance or accident risks. This is a rough simplification but an easy way to make our street network react to the traffic over longer time periods.

3. Implementation

The *Streets4MPI* software provides several overarching capabilities on top of a sizable preexisting software stack. In this chapter, we will discuss the implementation in detail. To that end, we will summarize the technical challenges we were facing when we started out, explain where and how we obtained viable street data and showcase the internal architecture of *Streets4MPI*. We will also present the visualization component of our application, in particular the results that it can generate.

3.1. Technical Challenges

As mentioned above, we decided early on that we wanted to use *Python* (rather than C) as our main implementation language for *Streets4MPI*. This decision was made in light of our previous programming experience and the ability of Python to quickly produce feature-rich programs.

The obvious downside to using an interpreted, dynamically typed scripting language are the inevitable speed obstacles. Python in particular is known for additional barriers hindering successful parallelization [Python Project 2012].

However it bears mentioning that Python offers a very compelling list of available modules and packages that can be easily accessed and used. In the end, we used the following packages to ease our workload:

- `imposm.parser`[1] to parse geographic street data from the *OpenStreetMap*[2] project.
- `python-graph`[3] to be able to easily construct and traverse large graphs. We used this module's capabilities for the shortest path calculations.
- `Python Imaging Library`[4] to create color graphics from our simulated data and save them to disk.
- `mpi4py`[5] to interface with MPI for the parallel processing.

We started the implementation under the assumption that the finished Python software would be significantly slower than a hypothetical equivalent software written in C. Part of our project work was to gauge the software's efficiency and look for bottlenecks. We expand on this in chapters 5 and 6.

3.2. Data Source

The successful execution of *Streets4MPI*'s premise obviously relies on the existence of viable street network data. As mentioned in section 2 the data would have to contain geographical information about all streets in any given area as well as their connectedness. In addition we need information about existing speed limits. Additional data about street type or size or other auxiliary data would be beneficial. Furthermore such data would preferably be freely available so that the results of the program's execution may also be freely shared.

Fortunately for *Streets4MPI*, such a data source is provided by the *OpenStreetMap* project. It contains all relevant data (and much more), is sufficiently accurate for our purposes and is available under a Creative Commons license [OpenStreetMap Wiki 2012]. There are also some drawbacks: The project makes only few promises concerning data integrity and the data format is poorly specified in some places. Still, it is a much better alternative to other external data sources and to the idea of generating street network data completely on our own.

Streets4MPI does not use a so-called *OpenStreetMap API server*, but instead relies on local copies of the data that `imposm.parser` can read. We supply a small dataset for

[1]`http://dev.omniscale.net/imposm.parser/`
[2]`http://openstreetmap.org/`
[3]`http://code.google.com/p/python-graph/`
[4]`http://www.pythonware.com/products/pil/`
[5]`http://code.google.com/p/mpi4py/`

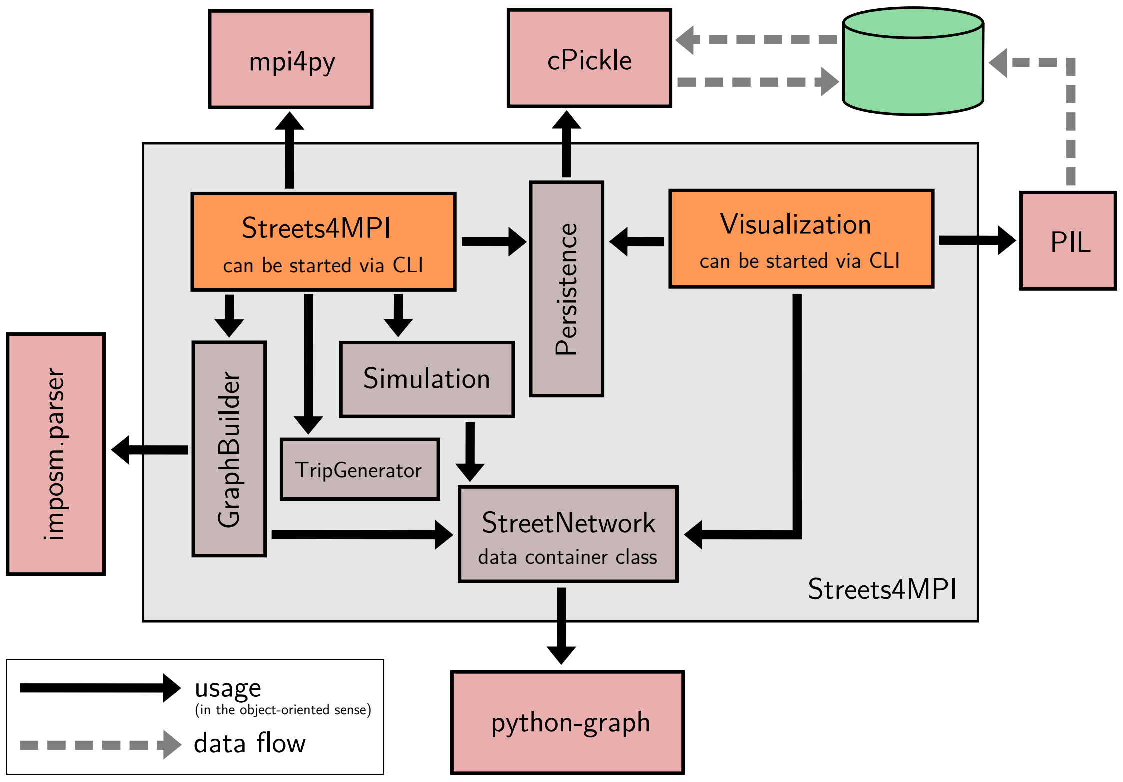


Figure 3: Overview of the software architecture of *Streets4MPI*. This diagram does not show every single class relation, but divides *Streets4MPI* into meaningful components and displays their interaction with external modules.

testing (approximately encompassing the Stellingen district of Hamburg). Recent data dumps sorted by country or state are available e.g. at *Geofabrik*[6].

For the sake of brevity, we will not discuss the OSM data format in this document. Suffice to say that it is XML based and describes a graph structure.

3.3. Software Architecture

Since we use external packages to do most of the "heavy lifting", the main task of our code is to pass data between the different parts and encapsulate interfaces accordingly. The overall architecture is visually summarized in figure 3.

The main application is represented by the class `Streets4MPI` and can be roughly seperated in three phases:

In the first phase the program creates an instance of the `GraphBuilder` class that reads the OSM data into memory and fills it into a `StreetNetwork` instance which represents our main data structure containing all the information about the street network. Internally it is built around the `Graph` data structure from the `python-graph` library.

[6]`http://download.geofabrik.de/osm/`

During the second phase a `TripGenerator` instance generates the trips. It randomly selects nodes from the street network as start nodes and assigns one or more destination nodes to them. We also experimented with selecting nodes based on their land use type to create trips that lead from residential areas to industrial or commercial areas. But since the OSM data provides only a few nodes with such a land use type assigned we had to discard that approach and decided to implement a completely random trip generation.

The third phase contains the execution of the `Simulation` and is the most important one. As described in section 2 our simulation runs in steps. During each step we first update all the driving times respectively the edge weights in the street network graph based on the traffic load calculated in the previous simulation step. Then the simulation calculates the shortest paths for all the trips and sums up the traffic load data. Within the shortest path calculation lies the most computation complexity. Here we use an implementation of Dijkstra's algorithm from the `python-graph` library.

We decided to completely decouple the visualization from running the simulation. Therefore the street network data and the traffic load data are serialized to disk as files with an `s4mpi` extension (internally, street network data is saved as pickle[7] output and traffic load as a numerical array, both *gzip*ped before being written to disk). Then – after the simulation is finished – the `Visualization` class is started as a seperate application that reads the previously generated `s4mpi` files from disk and renders them as PNG images. It may even be run on another machine, provided that all the files are transferred.

3.4. Visualization

The visualization component of *Streets4MPI* supports several different metrics, the most important one being the overall traffic load – an approximate metric for how many cars are on a certain street at any given point in time. Other than that, it can also visualize the speed limits, some related metrics like an approximation for the ideal speed based on total number of cars passing, and the connected components of the street graph.

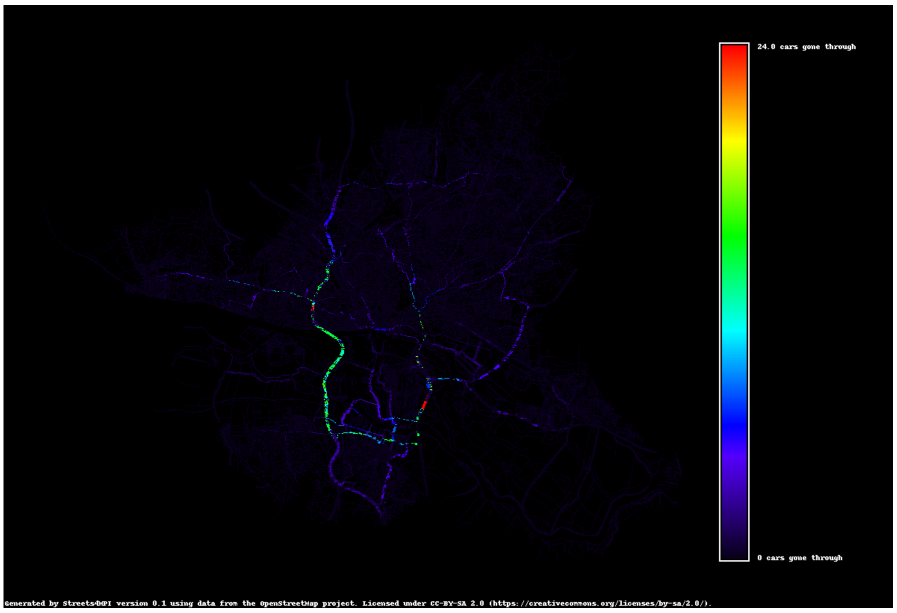
Furthermore, it supports two color modes: a heatmap mode where the value scale is mapped to colors in the way that is commonly used in heatmaps (bright red for highest values, dark blue for lowest) and a grayscale mode that is better suited for printing in black and white. An example output for each of these two color modes using Hamburg (Germany) for the street network is available in figure 4.

The visualization shares some components with the main *Streets4MPI* software, but is launched independently. It is not parallelized.

4. Parallelization

Since the parallel execution was one of our main project tasks we decided to dedicate a seperate chapter to this aspect of *Streets4MPI*. According to the methodology described in [Mattson et al. 2004] we will first have a look at the main computation tasks that our

[7]Python's versatile object serialization engine – `http://docs.python.org/library/pickle.html`


24.0 cars gone through
0 cars gone through
Generated by Streets4MPI version 0.1 using data from the OpenStreetMap project. Licensed under CC-BY-SA 2.0 (https://creativecommons.org/licenses/by-sa/2.0/).

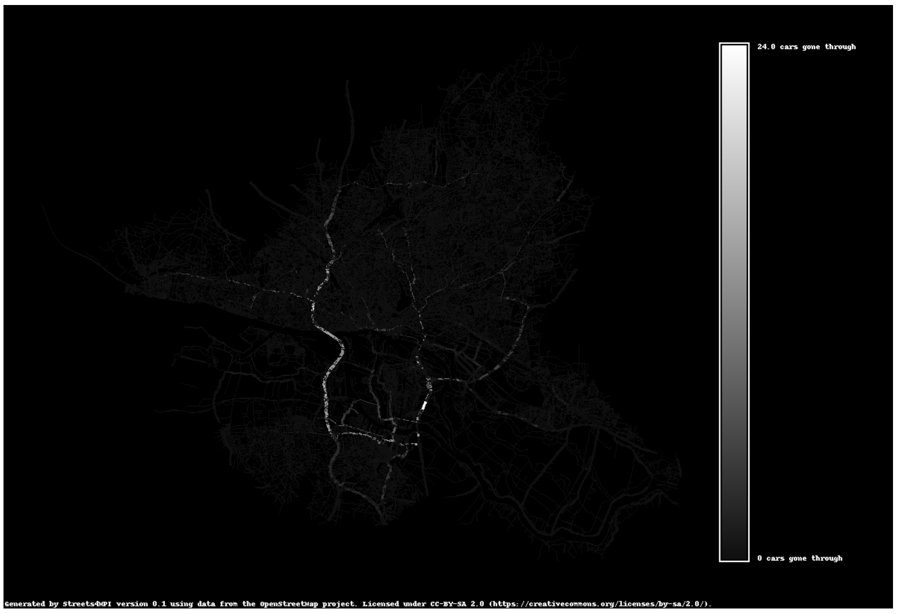

24.0 cars gone through
0 cars gone through
Generated by Streets4MPI version 0.1 using data from the OpenStreetMap project. Licensed under CC-BY-SA 2.0 (https://creativecommons.org/licenses/by-sa/2.0/).

Figure 4: *(a)* Simulated traffic load for the street network of Hamburg, Germany, in heatmap color mode. *(b)* The same traffic data, visualized in grayscale color mode.

program consists of, the main data they operate on and how both can be decomposed for parallelization before we explain our final parallelization strategy in section 4.2.

4.1. Task and Data Decomposition

In section 2 and 3.3 we described the main application flow. In doing so we already pointed out the tasks that our application consists of:

1. Create street network from OSM data
2. Generate trips
3. Simulation steps (repeat until simulation is finished)
   a) Update edges in street network graph
   b) Calculate the shortest path

Since we decided to use external libraries for parsing the OSM data and calculating the shortest paths we had to handle task 1 and 3.2 as atomic. Otherwise we could have splitted them further into smaller subtasks.

By listing the tasks we already did some grouping and ordering. In fact task 2 and 3 represent groups containing multiple tasks: We have to generate n trips and for each of these n trips we need to execute the tasks inside a simulation step namely task 3.1 and 3.2. All the tasks inside a group are independant from each other, so they can be handled as one task in the order that is given through the enumeration: At first we have to create the street network, only afterwards can we select start and destination nodes during the trip generation in task 2. And only after the trips have been created we can start the first simulation step containing task 3.1 and 3.2. Then every simulation step depends on the results from the previous step.

The reason why we listed task 3.1 and 3.2 as internal tasks of a simulation step is the traffic jam tolerance that we introduced in section 2.3. Since every resident has its own traffic jam tolerance we theoretically have to handle both tasks as one task: Every time we calculate a trip's shortest path we need to update all the edge weights in advance. On the other hand since large cities contain a lot of streets this is very cost-intense. We will handle this problem later in the next section.

First we will take a look at the data that the tasks operate on. Beside some small data that is exchanged between our application's components the main data structure is the street network graph. It is used by all of the tasks: Task 1 creates it, task 2 reads it, task 3.1 updates it and task 3.2 again reads it. If such a central data structure is involved a data driven decomposition would make sense. However our random trip generation presents a major problem: As trips can lead all across the city it is impossible to find a decomposition into parts that the tasks in group 3 can operate on more or less independently.

4.2. *Streets4MPI* Solution

Coming from the decomposition analysis that we described in the previous section we will now present our final parallelization strategy. The result of this analysis was that all the trip based tasks – thus their generation and the operations during each simulation step – can be executed independant from each other. Therefore we decided to organize our parallelization around the trips.

Since we use Dijkstra's algorithm that calculates the shortest path from one node to all other nodes the computation complexity related to each trip remains relatively constant. Hence we decided on a static scheduling: At the beginning we divide the number of residents respectively the number of trips by the number of processes. Then each process generates its part of the trips and runs the simulation with these trips.

In the previous section we already abandoned a decomposition of the street network graph. As we need the whole graph for calculating the shortest paths we let each process create its own copy of the street network graph. Since they are all based upon the same OSM data and a deterministic parsing process the different copies are indentical. Now there is one remaining dependency: At the beginning of each simulation step we update all the edge weights in the graph based on the complete traffic load data from the previous step. Here we need to do some synchronization: Every process saves its results seperate from the graph inside an array. After each simulation step we merge these arrays into one array and distribute it over the processes. Figure 5 shows the whole parallelization strategy.

In the previous section we pointed out that the traffic jam tolerance which is assigned to each trip presents a problem: According to this we would have to update all the edge weights in the street network graph before each shortest path calculation. To avoid this we made a simplification that exploits the fact that each process has its own copy of the entire street network graph. Instead of giving an individual tolerance value to each trip we gave one to each process respectively all the trips that are managed by it. In doing so we accepted the fact that our simulation's level of detail depends on the number of processes. On the other hand we saved a lot of computation because now we need to update the edge weights only once on each process.

Technically the application is written according to the single program multiple data principle which MPI is based on. Thus the same program runs on each process, the differences in execution only depend on the process ID. The whole MPI related code is encapsulated in the main class `Streets4MPI`. To merge the traffic load data after each simulation step we use the MPI operation allreduce. A more detailed description of the merging process is given later in section 5.2.

5. Speed and Efficiency

As mentioned in section 3.1, we expected our software to be relatively slow in comparison to a hypothetical version written in a lower-level programming language than Python. Still, we had high hopes towards the efficiency (scalability when adding more MPI processes). In this chapter, we explain what we found out.

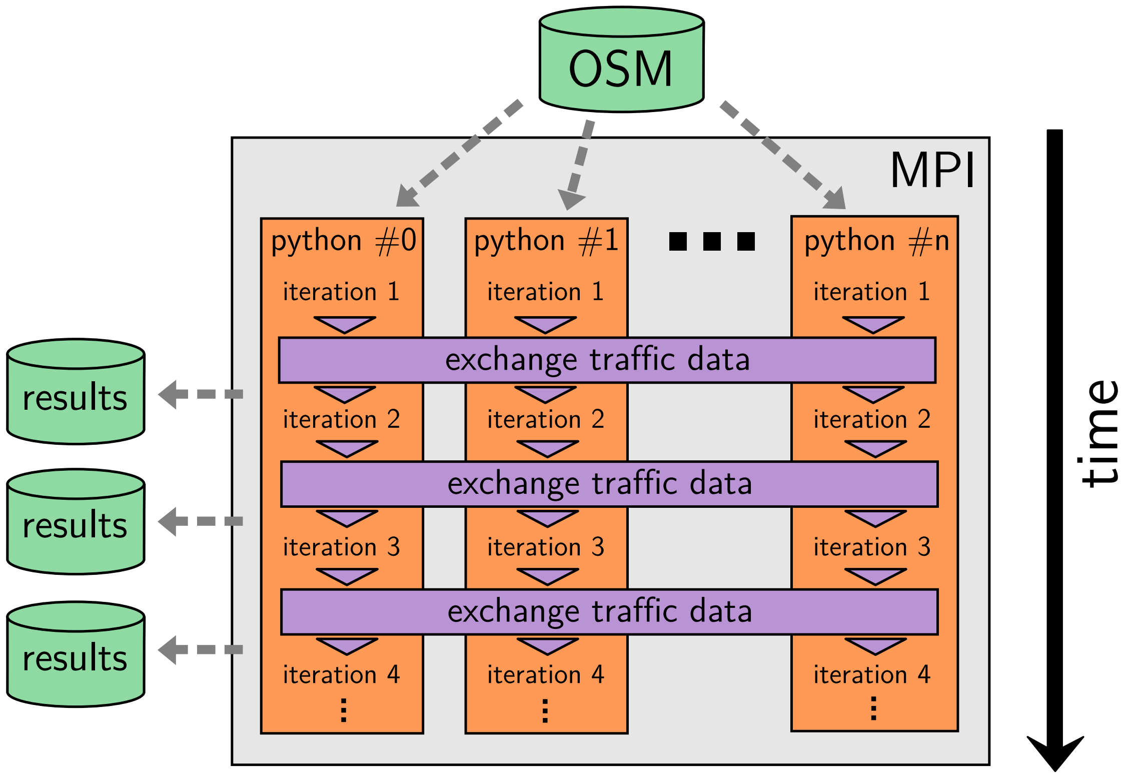


Figure 5: Diagram visualizing the parallelization strategy of *Streets4MPI*. The desired number of Python processes is launched through MPI, each process reads the OSM data, synchronization of the traffic load occurs after each iteration, and process #0 writes the results to disk.

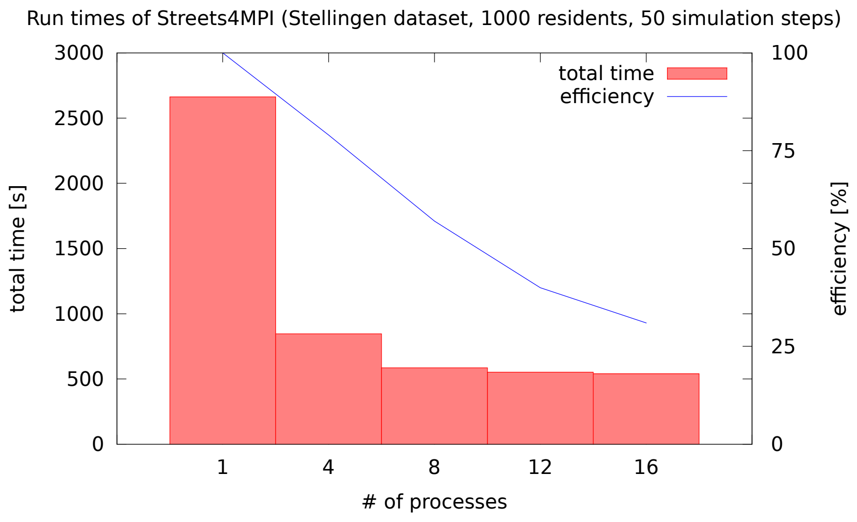


Figure 6: Speed test diagram for our first test, conducted April 3rd, 2012. Regrettably, the efficiency of the parallelization decreases rather quickly as we add more MPI processes.

We did most tests using a relatively small dataset, so that they would finish within several minutes to an hour. We also did some viability testing with bigger than default datasets (hundreds of thousands of streets/trips). There are no hard limits for the size of the street network or the number of trips, but it should be noted that the RAM usage can increasy sharply as more MPI processes are added.

5.1. Measurements

We did performance measurements on a testing machine that belongs to the DKRZ[8]. It is a machine powered by 12 processors with 4 cores each, running a total of 48 cores at 1.9 GHz. Detailed information about the system is collected in the appendix on page 28. Normally we always used the most recent version of *Streets4MPI* for all tests, making sure that all conditions remained consistent for tests that would be compared for efficiency.

To start off, we tested the efficiency by running a simulation with 1000 residents through 50 simulation steps in the Stellingen street network. The results can be seen in figure 6. This test reveals that, while our parallelization is not terrible for a first effort, it also seems to offer plenty of room for improvement. The total run time seems to consist of a large portion that is parallelized really well, as well as a constant part that results in little to no improvement as the number of MPI nodes increases into the double digits. The efficiency also decreases sharply.

These results are generally in line with what could be expected from our software.

[8]Deutsches Klimarechenzentrum – `http://dkrz.de/`

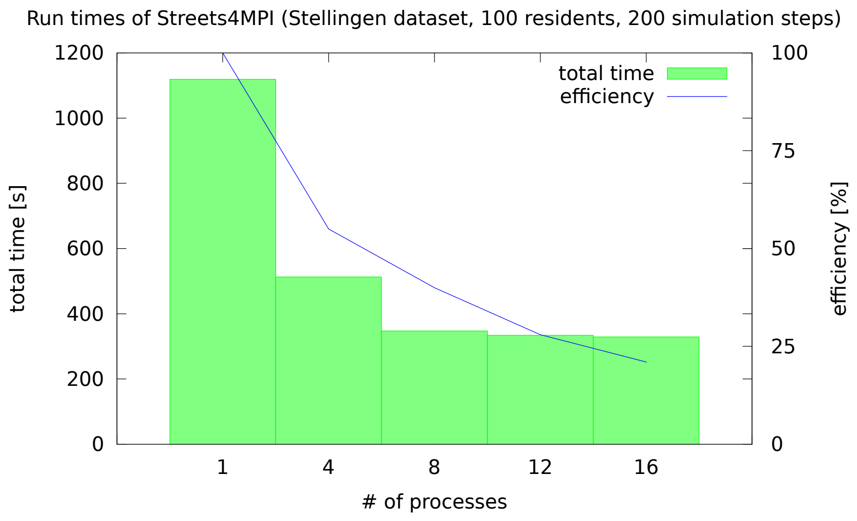


Figure 7: Speed test diagram for our second test, also conducted April 3rd, 2012. With fewer residents and more steps compared to the first test, *Streets4MPI* spends less time calculating paths and more time coordinating processes, resulting in a loss of efficiency.

The simulation can be cleanly divided into two parts:

1. The calculation of the shortest paths and the traffic load for the individual roads: This is what would be called the "main" part of our simulation and scales extremely well, because for every simulation step, every MPI process gets a certain number of trips and does all related calculations autonomously.

2. The summation of the process local traffic loads, saving the results to disk and starting the next simulation step: This part of the simulation would be very tricky to parallelize, if not outright impossible. As a result, it does not scale at all and takes an approximately constant time no matter the number of processes.

To confirm whether the above ideas are correct, we executed a second test run with different parameters: The number of residents was reduced from 1000 to 100 and the number of simulation steps was increased from 50 to 200, thereby forcing the software to spend less time calculating paths and more time coordinating the processes. We expected to see a proportionally bigger constant part to the total run time, resulting in a further drop in efficiency. The results of this test run can be seen in 7. They confirm our expectations.

The conclusion from our tests is that we should do our best to reduce the time spent on organizational tasks. We did a few experiments – some only in thought, some in code – which we will detail in the next section.

5.2. Lessons and Improvements

As mentioned above, a good way to improve the efficiency of our *Streets4MPI* software would be to reduce the time spent on non-parallelized tasks, such as disk I/O and communication.

Every step we need to save the traffic load data to disk, as well as the street network in case it has changed. Originally, we simply used Python's `pickle` module to serialize our data structure and save that to a file. The format used by `pickle` is not very space efficient, since it is a universal serializer that can handle any Python object.

Optimally, we would write our own serialization method for our data structures to save space and time. However doing so proved impractical for the following reasons:

1. Both the street network and the traffic load data use OpenStreetMap element IDs as the index variable. This element ID is numeric, but otherwise apparently poorly specified. Not much is known about it other than it being globally unique (except in some rare edge cases) and non-negative (except in some other rare edge cases), and that at least one implementation posits that many element IDs are 32 bits wide while others need 64 bits. Python transitions fluently between different numeric types even into BigNumber implementations, but if we were serializing it, we would have to decide how to encode it so that its length and value are self-evident *and* it can be indexed in a byte stream, which would be possible but complicated.

2. In addition, the street network uses an underlying data structure that is imported from the `python-graph` module, that is already quite sophisticated. Even if we found a good way to serialize the graph including all its nodes, edges, attributes and properties, we would have to rebuild the data structure essentially from scratch whenever we would deserialize a file.

We did not want to give up easily, but decided that these problems were not likely to be solved within our timeframe. We did however – so to speak – tackle the problem from the other side.

The first thing we did was to change the serialization pipeline so that the `pickle` output would be compressed (we decided on `gzip`) before it is written to a file. This reduces the file size (and thus, it is sensible to assume, also the I/O time) by about 70%, at the cost of slightly increased processor usage. We tested the speed-up using the same parameters as in the first test we did (cf. section 5.1). The results can be seen in figure 8. This change leads to an overall speedup of approximately 1%, almost constant even as the number of processes changes. We were expecting more of a positive change in efficiency, but are also glad that the speed has increased overall.

Next, we switched out `pickle` in favor of `cPickle`, a `pickle` reimplementation written in C rather than Python, promising a speed boost for the serialization and deserialization process. Indeed, this speedup has produced another overall speedup of about 1%. The relevant data is visualized in figure 9.

Under the assumption that the serialization/deserialization process and the disk I/O can not easily be optimized further, we decided to apply the above principles to the inter-process communication via MPI.

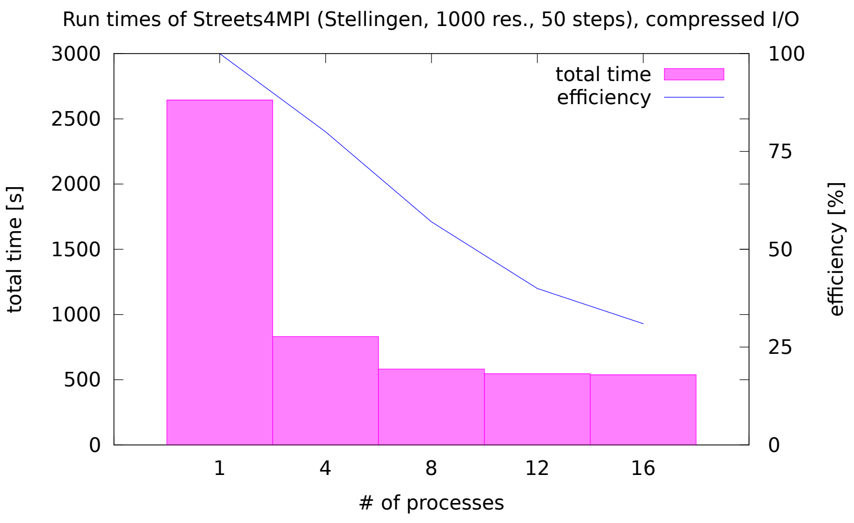


Figure 8: Speed test diagram for our third test, conducted April 18th, 2012. This is the first test using compressed serialization. It uses the same parameters as the first test (cf. figure 6) and as such should be compared to that specific test. The difference is not immediately visible to the naked eye, but there's an overall reduction of total run times of about 1%.

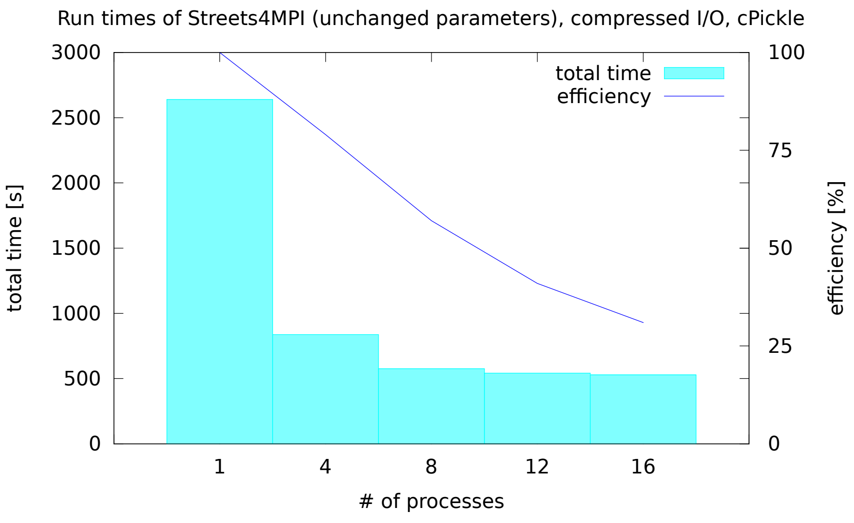


Figure 9: Speed test diagram for our fourth test, conducted April 18th, 2012. This is the first test using `cPickle` rather than `pickle`, yielding no change in program semantics, but another overall speedup of 1%.

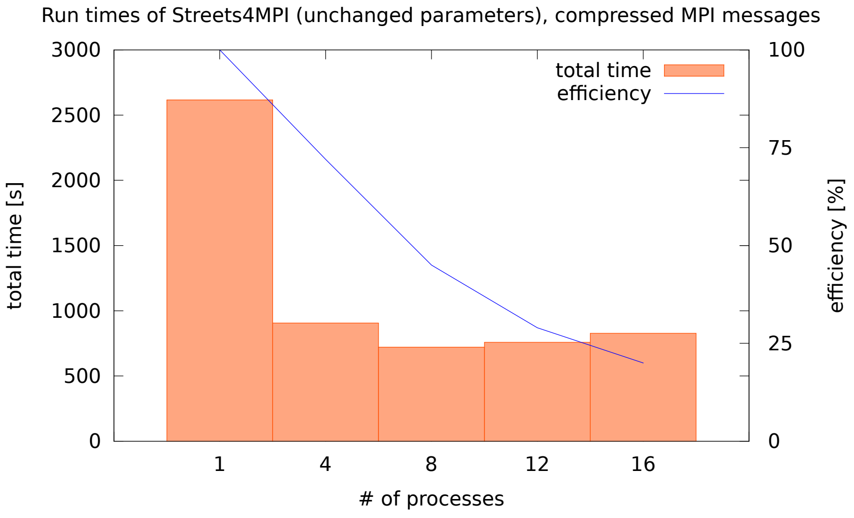


Figure 10: Speed test diagram for our fifth test, conducted April 18th, 2012. In this test, we handled the serialization of traffic data on our own. This proved to be fruitless, as the overhead of the conversion further destroyed the efficiency and actually lead to *increased* total runtime as the number of processes was increased.

Streets4MPI uses `mpi4py`, which is a Python layer on top of standard `mpich2`. Its documentation reveals [MPI for Python v1.3 documentation 2012] that `mpi4py` typically transmits objects serialized in `pickle` format as well, unless they are of a type that offers a special interface for buffer-based communication, e.g. a string or a numerical array. To offer this interface, a class must be implemented in C rather than Python. Python's native strings and arrays fulfill this requirement.

The only thing we send via MPI is the traffic load data, which we stored as a native Python `dict`. With the idea of implementing our own data type in C dismissed due to the complexity of the task, the only remaining option was conversion into a string or an array. Due to the aforementioned issues surrounding the OpenStreetMap element IDs, we were hesitant about creating our own array structure, since we wouldn't be able to assure it to be future-proof. We did however attempt to serialize the traffic data ourselves and transmit them as strings via MPI, in hopes of achieving a better run time efficiency. The results of this attempt can be seen in figure 10. It turned out to be quite inefficient and introduced a lot more overhead into the communication.

This version of the code has been committed to our public version control system and can be restored from the archive, but we were quick to replace it with what we had before and strongly advise not to use it.

Taking a step back, we considered that another possible venue for improvement would be the communication strategy. Up to that point, we used `mpi4py`'s implementation of `mpi_allgather` to exchange traffic load data between processes in such a way that every process has access to the data of every other process. However this traffic load data was then merely summed up. To clarify: Every process had a dictionary with graph edges as the index variable, holding the amount of times any specific edge was used by a trip. If this number was 0, the edge was not included in the dictionary. After gathering all dictionaries from all other nodes, every process then added up all the usage values for every street until it had a complete street usage dictionary for the current simulation step, which was then identical for every process.

This communication scheme intuitively looked like it should be able to be simplified. Under optimal conditions, one would use something like `mpi_allreduce` in situations where things have to be summed up, to save processing time. Unfortunately, `mpi4py`'s `mpi_allreduce` can only handle arrays for summation, which are then summed up for every index.

The `mpi4py` package offers two different implementations for the `allreduce` operation: A buffer-based one (`Allreduce` with a capital *A*) and a generalized one for other Python objects (`allreduce` in small letters). The `mpi4py` documentation suggests that the generalized variant of the reduction operation might not bring the optimization that one would hope for: “the actual required reduction computations are performed sequentially at some process” [MPI for Python v1.3 documentation 2012]. To support the buffer-based variant, we would have to save the traffic load data as arrays, which would require further modifications to our data structure.

Since we could not rely on the OSM element ID as a consistent identifier, we quickly decided that it would not be viable to use it as an array index. But for MPI `reduce` operations, the traffic load would have to be saved as an array, with only traffic load data values as elements, and the corresponding streets coded as the array index. Thus, we introduced our own street ID which is assigned at street network loading time. Whenever a street network is newly created from OSM data, every street gets a numerical ID starting from 0. Streets can be accessed by this ID. It is a perfect array index for the traffic load data.

We reimplemented the traffic load as numerical arrays instead of dictionaries. It took some small changes scattered across the software, but overall it was not too hard once we had the concept cleared. This new version of the software (at that point still using MPI's `allgather`) was then tested using the same parameters as before. The results of this test can be seen in figure 11 – and they surpassed our hopes by far. We were able to reduce the overall runtime for the non-parallelized case by about 30%, which is remarkable even by itself. More incredible still was the improvement of the efficiency as the number of processors increased: From a rather disappointing 31% efficiency for the 16-processes-case to a rather more pleasant 78%.

Although our dataset for these tests was too small for it to make any kind of huge impact, we were still committed to transmit the traffic load data via MPI's `Allreduce` operation (using the buffer-based interface), as we had originally planned. With the groundwork done, this was relatively reasy to do. The results of the final speed test can be seen in figure 12. Compared to the previous test, there are no obvious changes. We assume that the benefits of this method are simply not visible using our testing dataset.

Overall, we are very content with the improvements we were able to make.

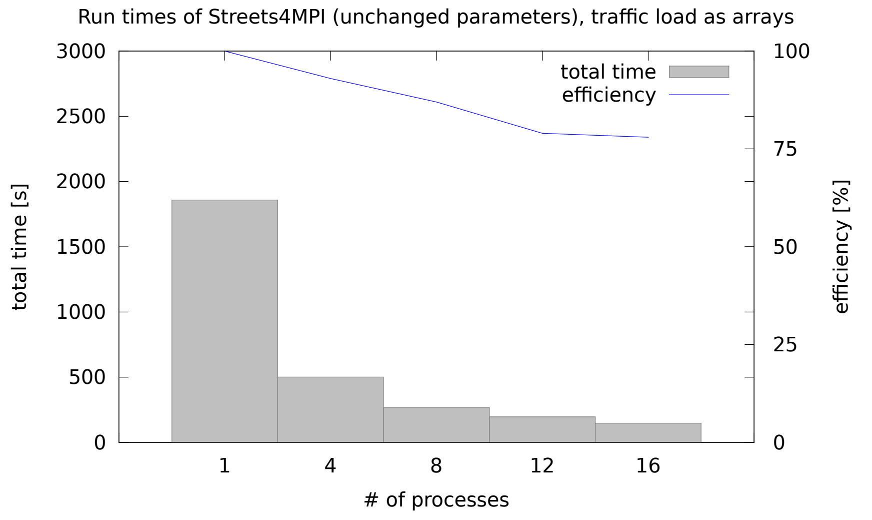


Figure 11: Speed test diagram for our sixth test, conducted May 2nd, 2012. In this test, we introduced a new method for saving and transmitting traffic load data, which proved to be extremely beneficial. We note a significant boost in overall speed and, more importantly, an extraordinary improvement in efficiency.

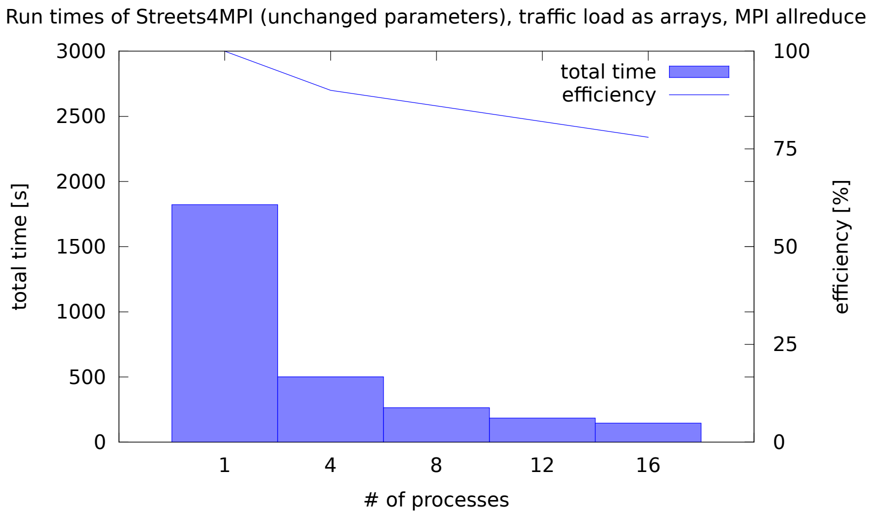


Figure 12: Speed test diagram for our seventh and final test, conducted May 2nd, 2012. In this test, we switched to the `allreduce` method for the traffic load. Improvements over the previous test are incremental at most. It is likely that the benefits of the `allreduce` usage would be more visible using a bigger dataset and more residents than we used here.

6. Potential for Future Improvements

In this chapter we propose a few approaches for improving *Streets4MPI* that might be pursued in the future maybe by interested individuals or in the form of further university projects like ours. The first approaches refer to the underlying simulation concept, the next ones improve the technical realisation, mainly its performance.

6.1. Improved Simulation Model

In chapter 2 we already named some of the limitations that our simplified model implicates.

One of them was the completely random trip generation. In most cities one can observe that the greater part of the residents lives in the outer areas and drives to work somewhere near the city center. This can be modeled by adding an attractiveness to each node that in- or decreases the possibility to be selected as a start or destination node. This attractiveness could then depend on the distance to the city center. In 3.3 we described that a few nodes in the OSM data have an assigned land use type which marks them as a “residential” or “industrial” node. This attribute could also be considered when rating the attractiveness.

Further improvements could be done when adapting the street network as described in section 2.4. We could think of blocking streets for some simulation steps while the construction is in progress or residents moving away from widened streets respectivly lowering these streets’ attractivness.

6.2. Dynamic Shortest Path

The most computation complexity lies within the calculation of the shortest paths. Since we update all our edge weights at the beginning of each simulation step we need to recalculate the shortest paths again and again. But although the traffic changes in general during a simulation step it might stay the same for some of the streets. Under some circumstances it wouldn’t be necessary to recalculate all shortest paths.

This is the idea of dynamic shortest path algorithms. Given a graph they initially compute all shortest paths. Then if the graph is changed locally they update only those shortest paths that are influenced by the changes. By keeping an initial set of shortest path up-to-date computation time is saved: For example if the weight of an edge is decreased all the shortest paths containing that edge don’t need to be recalculated because they become even shorter. On the other hand if an edge weight is increased only those shortest paths containing the edge need to be recalculated because other paths won’t become shorter by using the changed edge.

There are several different algorithms available that use different mechanisms to determine which paths need to be updated. Some of them are presented in [Demetrescu and Italiano 2006].

6.3. Improved Parallelization

While *Streets4MPI* has made quite some strides to improve its efficiency, it is certainly not perfect – there must still be ways to improve the communication and reduce the overhead.

An emergent candidate for improvement would be I/O handling: At the moment, the street network must be read from disk by every single process, which seems wasteful until one realizes that reading it at some process and then broadcasting it, leaving all others waiting pointlessly, would be even worse. Maybe there are parallelization opportunities there.

Similiarly, all persisted data (street network revisions and traffic loads) are written to disk by the process with the MPI rank of 0, which causes it to block and lag behind the others when calculating the paths for the iteration.

Additionally, maybe the `PyMPI` project[9] can offer technical advantages that `mpi4py` doesn't.

7. Summary

With the project task and scope in mind we are satisfied with *Streets4MPI*, its capabilities and the progress it has made. We were able to deliver a software that can simulate street traffic in any street network and adapt the streets to the traffic demands. Furthermore we invested a lot of effort into ensuring that it remains scalable and efficient across parallel processors.

Streets4MPI is free software. Information on how to download and run it are available at the project's website[10].

[9] `http://pympi.sourceforge.net/`
[10] `http://jfietkau.github.com/Streets4MPI/`

Appendix

Run Time Data

This is the run time data collected during our speed up tests, running *Streets4MPI* on our testing machine (see next section for detailed hardware information). Example call: `time mpiexec -n 16 python streets4mpi.py`

```
=== 2012-04-04: pickle-based serialization ===

test.osm, 1000 res, 50 steps, 1 node:
real 44m23.046s
user 44m17.822s
sys 0m3.680s

test.osm, 1000 res, 50 steps, 4 nodes:
real 14m6.164s
user 54m35.741s
sys 1m40.930s

test.osm, 1000 res, 50 steps, 8 nodes:
real 9m46.343s
user 75m37.056s
sys 2m20.957s

test.osm, 1000 res, 50 steps, 12 nodes:
real 9m11.719s
user 105m35.460s
sys 4m30.217s

test.osm, 1000 res, 50 steps, 16 nodes:
real 9m0.204s
user 136m50.473s
sys 6m48.298s



test.osm, 100 res, 200 steps, 1 node:
real 18m39.199s
user 18m28.369s
sys 0m0.356s

test.osm, 100 res, 200 steps, 4 nodes:
real 8m32.741s
user 31m30.498s
sys 2m26.165s

test.osm, 100 res, 200 steps, 8 nodes:
real 5m47.401s
user 41m54.001s
sys 4m7.887s

```

```
test.osm, 100 res, 200 steps, 12 nodes:
real 5m34.219s
user 59m47.856s
sys 6m37.881s

test.osm, 100 res, 200 steps, 16 nodes:
real 5m29.153s
user 78m23.314s
sys 8m51.493s



=== 2012-04-18: compressed serialization ===

test.osm, 1000 res, 50 steps, 1 node:
real 44m4.473s
user 44m0.073s
sys 0m0.392s

test.osm, 1000 res, 50 steps, 4 nodes:
real 13m51.084s
user 53m44.994s
sys 1m29.678s

test.osm, 1000 res, 50 steps, 8 nodes:
real 9m41.537s
user 74m32.992s
sys 2m44.930s

test.osm, 1000 res, 50 steps, 12 nodes:
real 9m5.765s
user 104m15.907s
sys 4m33.449s

test.osm, 1000 res, 50 steps, 16 nodes:
real 8m56.532s
user 136m41.953s
sys 5m54.334s



=== 2012-04-18: compressed serialization, cPickle ===

test.osm, 1000 res, 50 steps, 1 node:
real 43m59.562s
user 43m55.757s
sys 0m0.300s

test.osm, 1000 res, 50 steps, 4 nodes:
real 13m58.493s
user 54m17.048s
sys 1m29.246s

test.osm, 1000 res, 50 steps, 8 nodes:
```

```
real 9m35.552s
user 74m12.482s
sys 2m19.521s

test.osm, 1000 res, 50 steps, 12 nodes:
real 9m2.020s
user 103m59.690s
sys 4m5.683s

test.osm, 1000 res, 50 steps, 16 nodes:
real 8m49.439s
user 135m49.861s
sys 4m58.503s



=== 2012-04-18: compressed serialization, MPI using serialized data ===

test.osm, 1000 res, 50 steps, 1 node:
real 44m21.334s
user 44m18.126s
sys 0m0.280s

test.osm, 1000 res, 50 steps, 4 nodes:
real 15m5.995s
user 58m30.255s
sys 1m50.927s

test.osm, 1000 res, 50 steps, 8 nodes:
real 12m0.814s
user 92m37.643s
sys 3m17.468s

test.osm, 1000 res, 50 steps, 12 nodes:
real 12m37.881s
user 144m7.204s
sys 6m49.054s

test.osm, 1000 res, 50 steps, 16 nodes:
real 13m47.411s
user 208m29.014s
sys 9m9.486s



=== 2012-05-02: traffic load as arrays ===

test.osm, 1000 res, 50 steps, 1 node:
real 30m58.271s
user 30m54.996s
sys 0m0.240s

test.osm, 1000 res, 50 steps, 4 nodes:
real 8m21.201s
```

```
user 32m35.330s
sys 0m42.775s

test.osm, 1000 res, 50 steps, 8 nodes:
real 4m25.548s
user 33m53.327s
sys 1m19.737s

test.osm, 1000 res, 50 steps, 12 nodes:
real 3m15.611s
user 36m20.500s
sys 2m27.817s

test.osm, 1000 res, 50 steps, 16 nodes:
real 2m27.817s
user 36m52.994s
sys 1m41.966s



=== 2012-05-02: traffic load as arrays, using MPI allreduce ===

test.osm, 1000 res, 50 steps, 1 node:
real 30m22.387s
user 30m19.450s
sys 0m0.276s

test.osm, 1000 res, 50 steps, 4 nodes:
real 8m20.850s
user 32m42.283s
sys 0m34.670s

test.osm, 1000 res, 50 steps, 8 nodes:
real 4m23.734s
user 33m49.991s
sys 1m7.928s

test.osm, 1000 res, 50 steps, 12 nodes:
real 3m5.335s
user 35m15.804s
sys 1m30.758s

test.osm, 1000 res, 50 steps, 16 nodes:
real 2m26.309s
user 36m30.049s
sys 2m4.456s
```

runtimedata.txt

Test Machine Hardware/Software Info

This is the detailed hardware and software information for the machine on which we ran our performance tests.

```
Architecture:    x86_64
CPU op-mode(s):  32-bit, 64-bit
Byte Order:      Little Endian
CPU(s):          48
On-line CPU(s) list: 0-47
Thread(s) per core: 1
Core(s) per socket: 12
CPU socket(s):   4
NUMA node(s):    8
Vendor ID:       AuthenticAMD
CPU family:      16
Model:           9
Stepping:        1
CPU MHz:         1900.206
BogoMIPS:        3800.12
Virtualization:  AMD-V
L1d cache:       64K
L1i cache:       64K
L2 cache:        512K
L3 cache:        5118K
NUMA node0 CPU(s): 0-5
NUMA node1 CPU(s): 6-11
NUMA node2 CPU(s): 12-17
NUMA node3 CPU(s): 18-23
NUMA node4 CPU(s): 24-29
NUMA node5 CPU(s): 30-35
NUMA node6 CPU(s): 36-41
NUMA node7 CPU(s): 42-47
```

Output of `lscpu`

```
            total    used     free   shared  buffers   cached
Mem:      129130    3100   126029      0       48      1623
-/+ buffers/cache: 1428   127701
Swap:          0       0        0
```

Output of `free -m` (after some usage)

```
Linux magny1 3.0.0-14-server #23-Ubuntu SMP Mon Nov 21 20:49:05 UTC 2011 x86_64
x86_64 x86_64 GNU/Linux
```

Output of `uname -a`

```
Python 2.7.2+
```

Output of `python --version`

```
mpiexec (OpenRTE) 1.4.3

Report bugs to http://www.open-mpi.org/community/help/
```

Output of `mpiexec --version`

How to use MPI in Python

Motivation

MPI is a well established standard for distributed parallel programming, capable of extending the SPMD paradigm to a cluster network of computing nodes. Therefore, it is one of the essential tools in high performance programming.

MPI implementations are available and well-supported for code written in C or Fortran. Since performance is one of the most important aspects in parallel processing those low-level languages are often chosen when teaching parallel programming concepts. However, if students have a more object-oriented or high-level programming background learning parallel programming concepts and C paradigms at the same time is a huge challenge. In this case dynamic high-level languages (such as Python or Ruby) tend to be better-suited for quickly producing working results, requiring less time fiddling with details such as explicit memory management.

We do not dispute that a low-level language might very well be the most sensible option for large-scale parallel software, since in those cases the costs from time and energy spent running the program can quickly eclipse development costs. However, especially in practice scenarios or student exercises, it might be a better alternative to use a language that offers less overall efficiency if it can significantly reduce unnecessary distractions during development.

To that end, we explain how one can harness MPI when programming in Python.

MPI in Python

There are mainly two MPI implementations in Python available: pyMPI[11] and mpi4py[12]. The latter is the one that we will present here.

Getting started with mpi4py requires only a cursory look into the documentation[13]. As can be seen, the established MPI idioms are translated more or less directly to Python. There are plenty of ways to structure and manage MPI communication, but the simplest case requires only the concept of the *MPI communicator* (the predefined `world` communicator is a collection of all available nodes) and of simple, blocking, point-to-point communication methods like `send` or `receive`. Here's a small code example taken from the mpi4py documentation[14]:

[11]`http://pympi.sourceforge.net/`
[12]`http://mpi4py.scipy.org/`
[13]`http://mpi4py.scipy.org/docs/usrman/index.html`
[14]`http://mpi4py.scipy.org/docs/usrman/tutorial.html#point-to-point-communication`

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
  data = {'a': 7, 'b': 3.14}
  comm.send(data, dest=1, tag=11)
elif rank == 1:
  data = comm.recv(source=0, tag=11)
```

In this example each node determines its own MPI rank. If it is 0, it creates a `dict` object containing some data and sends it to node 1. Node 1 takes care to receive the object from node 0. All other nodes do nothing.

A python script that uses mpi4py may be executed (example: 12 processes) like this:

```
mpiexec -n 12 python example.py
```

If an MPI backend is available and properly configured, the script may also be launched directly for testing:

```
python example.py
```

This executes it in an MPI context where there is only one process on one node.

Besides simple, blocking, point-to-point operations like `send` and `receive`, mpi4py also offers collective communication operations, and non-blocking operation variants. Furthermore, most communication operations are provided with a capital first character (e.g. `Send`) as well as all lower case (e.g. `send`). There is an important difference between these two operations: The latter can communicate generic Python objects of any kind, but they are serialized and deserialized using Python's `pickle` module, introducing a sizable overhead to most communication operations, especially if large data structures are involved. The versions with the capital letters make use of the so-called *single-segment buffer interface*, a way for Python types to make their data available through a contiguous memory buffer. In mpi4py, this buffer interface can be used to send and receive the objects much more quickly. The caveat is that this interface can not be provided by custom Python classes, unless they are internally written in C. However, Python's native strings and numerical arrays provide this interface, as do `numpy` arrays. To guarantee good results for the runtime efficiency, usage of the generic operations should be avoided and the buffer-based ones should be used wherever possible.

Due do Python being an interpreted, dynamic language, it will in the general case be slower than a structurally comparable program written in C. Nevertheless, it is very possible to look at the relative efficiency of the software as MPI nodes are added. Achieving a high degree of efficiency is definitiely possible using Python and mpi4py.

References

Demetrescu and Italiano 2006
Demetrescu, C. ; Italiano, G.F.: Experimental Analysis of Dynamic All Pairs Shortest Path Algorithms. In: *ACM Transactions on Algorithms (TALG)* 2 (2006), no. 4, pp. 578–601

Mattson et al. 2004
Mattson, T. ; Sanders, B. ; Massingill, B.: *Patterns for Parallel Programming.* Addison-Wesley Professional, 2004

MPI for Python v1.3 documentation 2012
MPI for Python v1.3 documentation: *Design and Interface Overview.* `http://mpi4py.scipy.org/docs/usrman/mpi4py.html`. Version: May 2012, Last checked: May 2012

OpenStreetMap Wiki 2012
OpenStreetMap Wiki: *OpenStreetMap License.* `http://wiki.openstreetmap.org/wiki/OpenStreetMap_License`. Version: May 2012, Last checked: May 2012

Python Project 2012
Python Project: *Global Interpreter Lock.* `http://wiki.python.org/moin/GlobalInterpreterLock`. Version: May 2012, Last checked: May 2012

Weber et al. 2009
Weber, B. ; Müller, P. ; Wonka, P. ; Gross, M.: Interactive Geometric Simulation of 4D Cities. In: *Computer Graphics Forum* Bd. 28 Wiley Online Library, 2009, pp. 481–492