

Smart Pathing

Ausarbeitung im Zuge des Praktikums „Parallele Programmierung“

Rami Aly und Christoph Hüter

Agenda



- I. Kontext
- II. Problem- und Modellbeschreibung
- III. Sequenzielles Programm und Lösungsansatz
 - I. Preprocessing
 - II. Routentabelle und Simulation
 - III. Visualisierung
 - IV. Laufzeiten
- IV. Parallelisierung
 - I. Routingtabelle
 - I. Methode a)
 - II. Methode b)
 - II. Bewegung der Autos/Simulation
- V. Performanceanalyse und Skalierbarkeit
- VI. Verbesserungen und Ausblick

Kontext



Gegeben: Karte eines Straßennetzes, Autos mit Start- und Zielorten

Ziel: Konstruktion von optimalen Wegen und intelligente Berechnung eines möglichst optimalen Weges von Fahrzeugen zu einem Ziel in komplexen Straßennetzen

Problem- und Modellbeschreibung

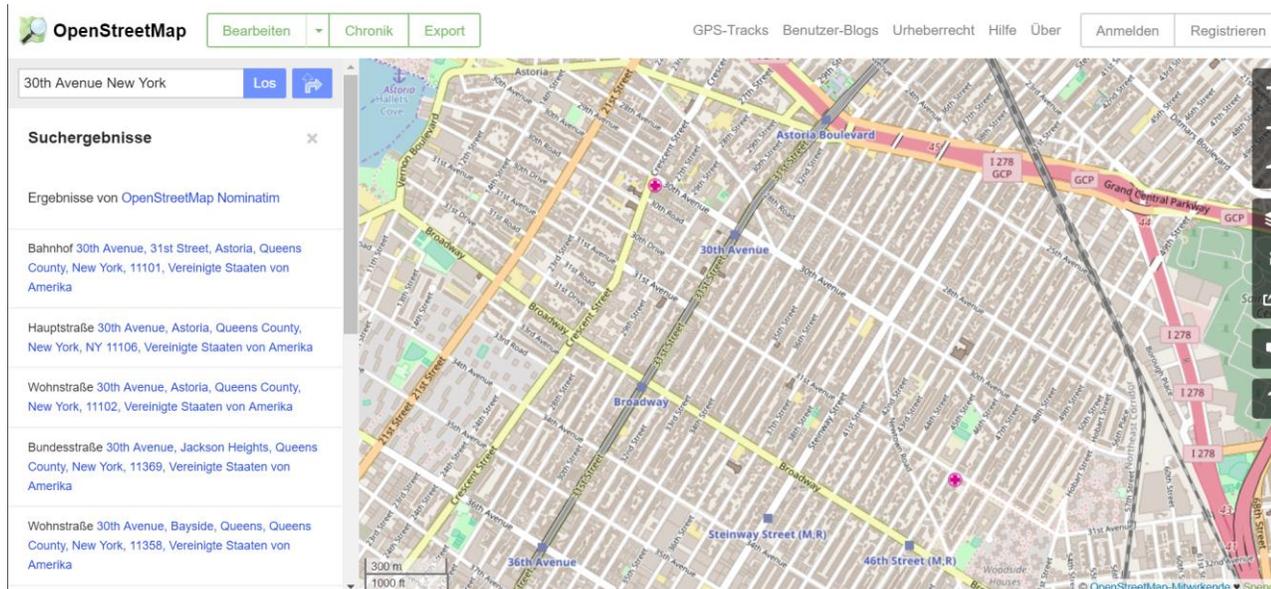
- Gerichteter Graph (V, E)
- Kante hat Kapazitätsfunktion e
- Kantengewichtsfunktion f , abhängig von e
- Spawner $S \subseteq V$ Eintrittspunkt für Marken
- Ampelfunktionalität von Knoten übernommen



Sequentielles Programm - Preprocessing



- Kartendatenbank OpenStreetMap(Geofabrik)
- Pythonscript zum Extrahieren der Knoten und relevanten Straßen
- Anpassung und Normierung der Koordinaten und Längen



VertexID	PosX	PosY
...

VertexID start	VertexID end
....

Lösungsansatz

1. Routentabelle und Timetable
2. Bewegung und Simulation der Autos



Sequentielles Programm - RoutingTable

- Erzeuge Graph aus extrahierten Informationen im C++ Programm
- Randknoten werden als Spawner ausgewählt (beliebig änderbar)
- Benutze A* Start Algorithmus und Luftlinie als Heuristik zur Berechnung (Boost Graph Library [BGL])



Pfade:

ID	5	18
5	()	(.....cba)
18	(abc....)	()

Kosten:

ID	5	18
5	0	Distanz({1,2})
18	Distanz({1,2})	0

Nachbarn:

ID	K_nearest
5	{n ₁ ,...,n _k }
18	{n ₁ ,...,n _k }
19	{n ₁ ,...,n _k }

Sequentielles Programm - Simulation

- Spawner erzeugen Marken mit Ziel
- Suche Pfad mit niedrigstem Wert:

$$\theta(v, w) = c[v, w] + \sum_{e \in RT[v, w].E} f_e(t_e) + dist(w_m, w)$$

- Jede Kante speichert in einer Timetable für jeden Zeitpunkt voraussichtliche Belegung
- Nach Wahl der Route aktualisiere die Timetable für alle betroffenen Kanten mit Heuristik



Sequentielles Programm - Simulation



- Bewegung von Marke innerhalb der Kante wird über eine Schlange geregelt
- Das Wechseln der Kante wird von dem verbundenen Knoten gesteuert
 - Bedingung: Ampel grün und noch Platz auf der Kante
- Bewegung der Autos geschieht in zwei Schritten:
 - Bewege Marke und merke falls es Edge überschreiten möchte
 - Prüfe ob Marke Edge überschreiten kann und bewege es um verbleibende Strecke
- Verlässt Marke Kante, aktualisiere Gewicht und zerstöre sie ggf.
- Innerhalb jedes Schrittes, mehrere Update
- Sequentiell → alte Zustände werden verwendet
- Speichere Belegung und Kapazität jeder Kante nach jedem Step in Datei ab

Sequentielles Programm - Simulation



Queue für Kante

Queue_3_1
car#0001
car#0002
car#1234
...
...
car#9876

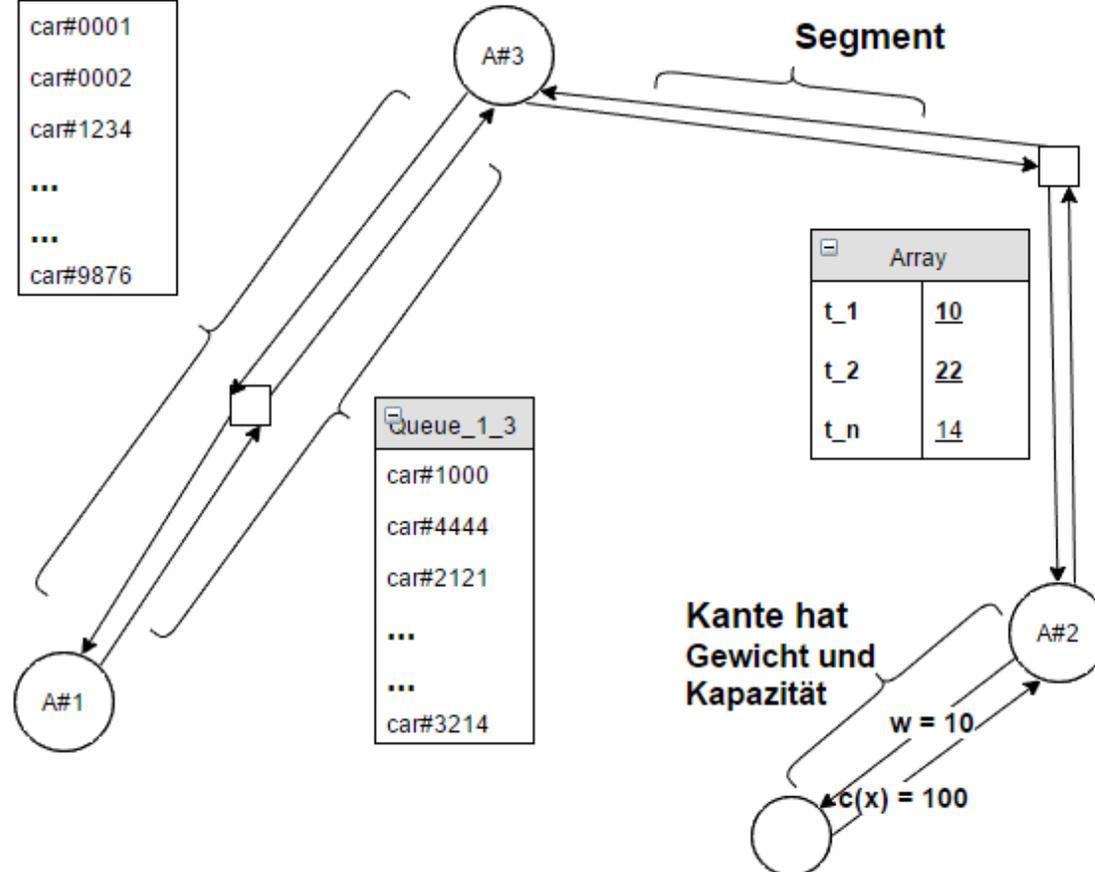
Ampel/Kreuzung

Segment

Belastung zu Zeitpunkt t

Array	
t_1	<u>10</u>
t_2	<u>22</u>
t_n	<u>14</u>

Array	
t_1	<u>17</u>
t_2	<u>10</u>
t_n	<u>30</u>



Kante hat Gewicht und Kapazität

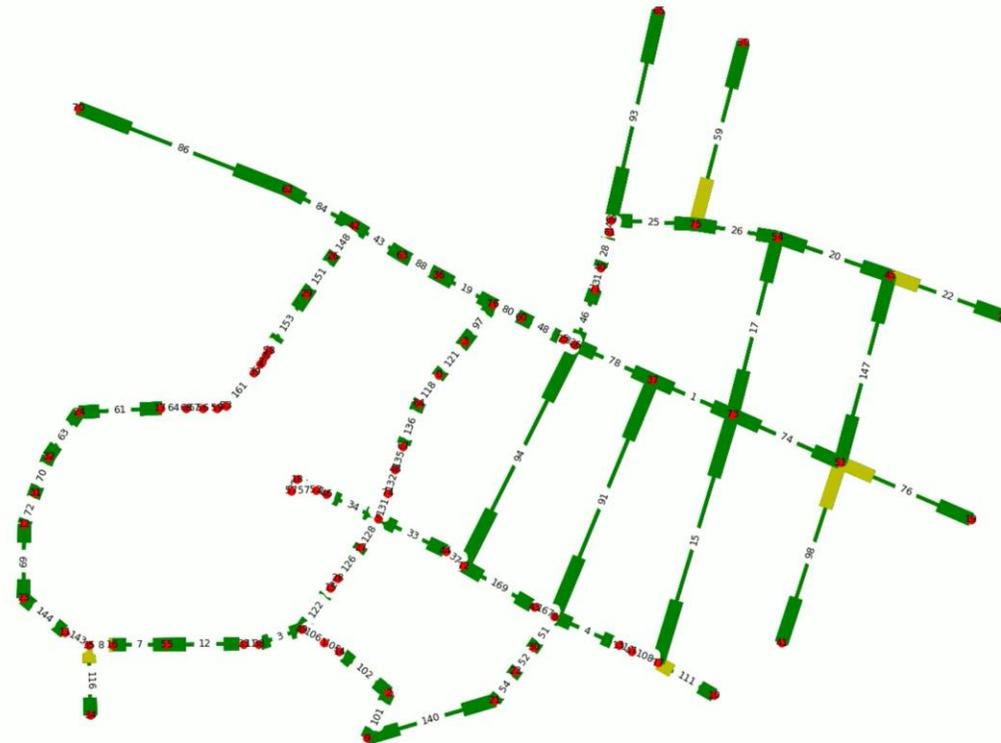
$$w = 10$$

$$c(x) = 100$$

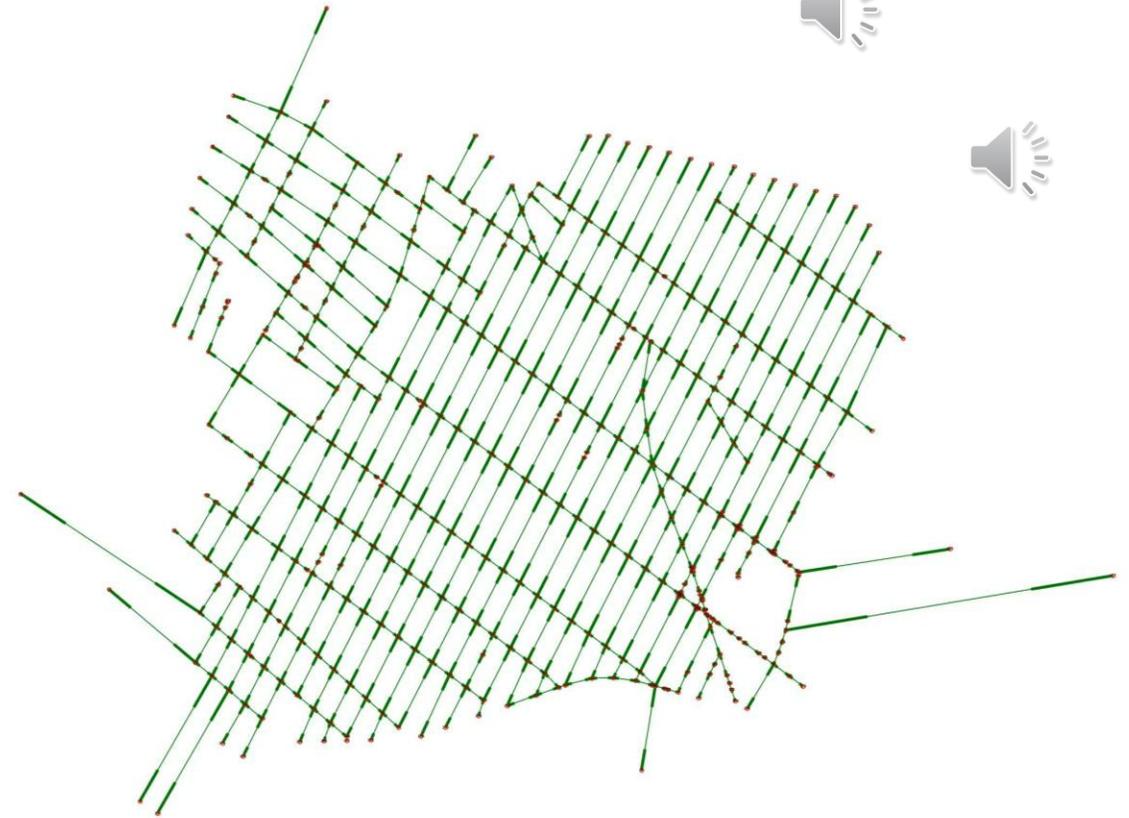
Sequentielles Programm - Visualisierung



- Visualisierung des Graphens und der Dichte der Autos auf den Kanten.
- Visualisierung mit Python Script, Networkx und Mathplot
- Erzeuge Animation aus Bilder



Sequenzielles Programm - Visualisierung



Sequentielles Programm - Laufzeiten



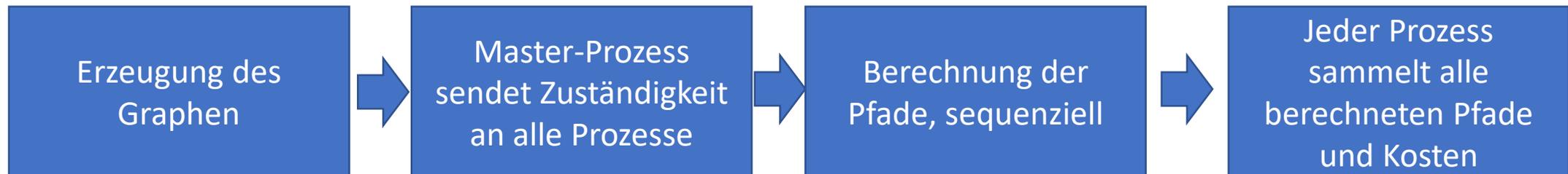
10 Durchläufe Durchschnitt:

- Knoten: 78, Kanten: 170, Steps: 200
 - RoutingTabelle: 0:00:857, Optimiert: 0:00:75
 - Simulation: 00:41

- Knoten: 424, Kanten: 1244, Steps: 200
 - RoutingTabelle: 1:37, Optimiert: 1:06
 - Simulation: 27:23

Parallelisierung – RoutingTable, Methode a)

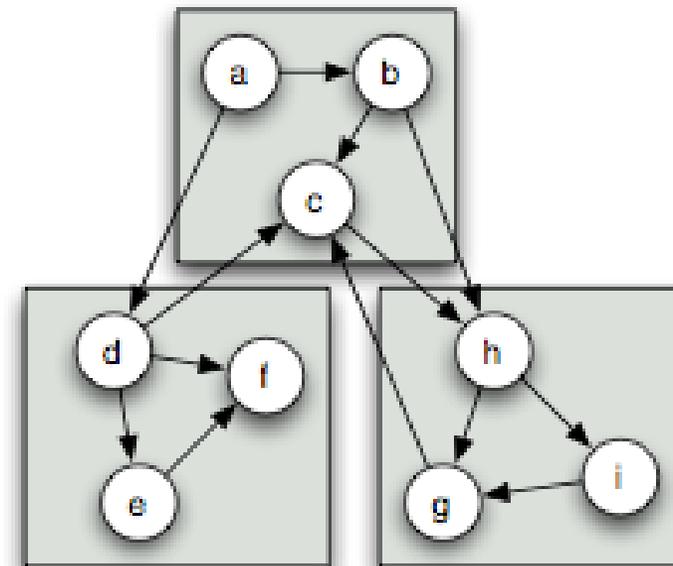
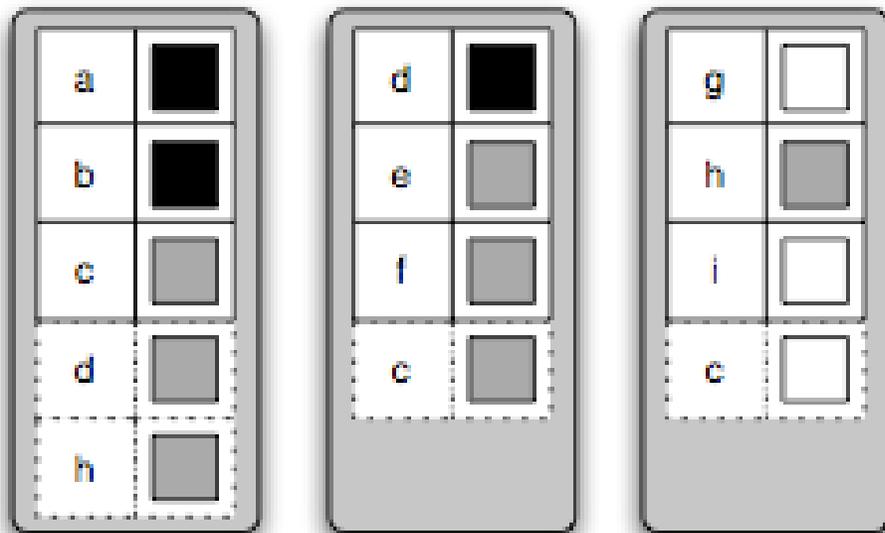
- Berechnung des Pfades sequenziell, parallel für mehrere Spawner durchführen
- Scatterv zur Aufgabenverteilung
- Gatherv für alle Prozesse, zum sammeln der Pfade



Parallelisierung – RoutingTable, Methode b)



- Nutzung der Parallel BGL
- Berechnung des kürzesten Pfades parallel, nacheinander für alle Spawner

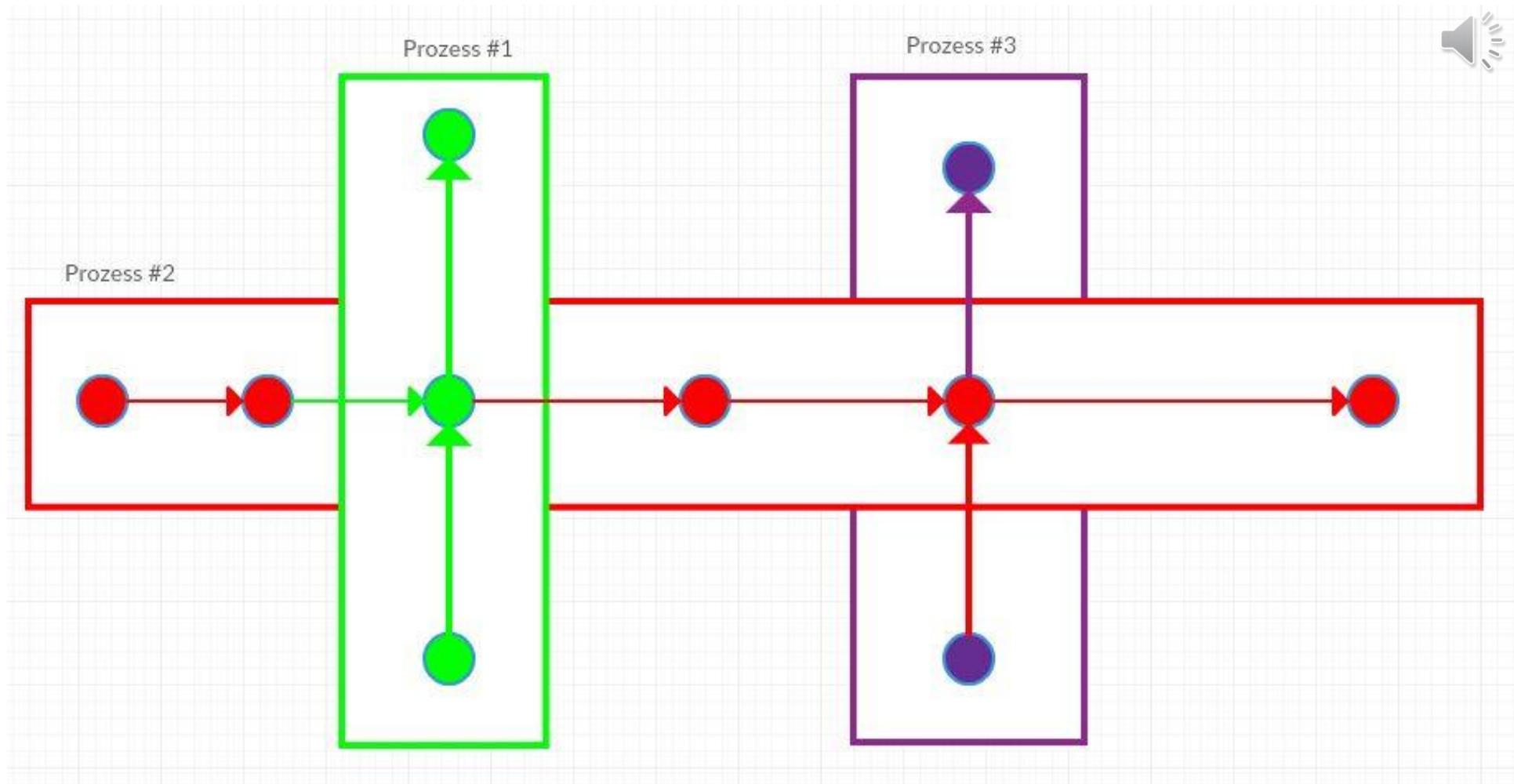


Parallelisierung - Simulation



- Aufteilung des Graphens auf Prozesse
- Jede Route wird einem Prozess zugeordnet
 - Jeder Knoten auf Route wird Prozess zugeordnet
- Bei Konflikt zwischen mehreren Prozessen auf Vertex → eindeutige Zuordnung
- Knoten im Graphen merken sich Schnittstellen mit Kanten
 - Möglich da Konfliktlösung lokal geschieht

Parallelisierung - Simulation



Parallelisierung - Simulation

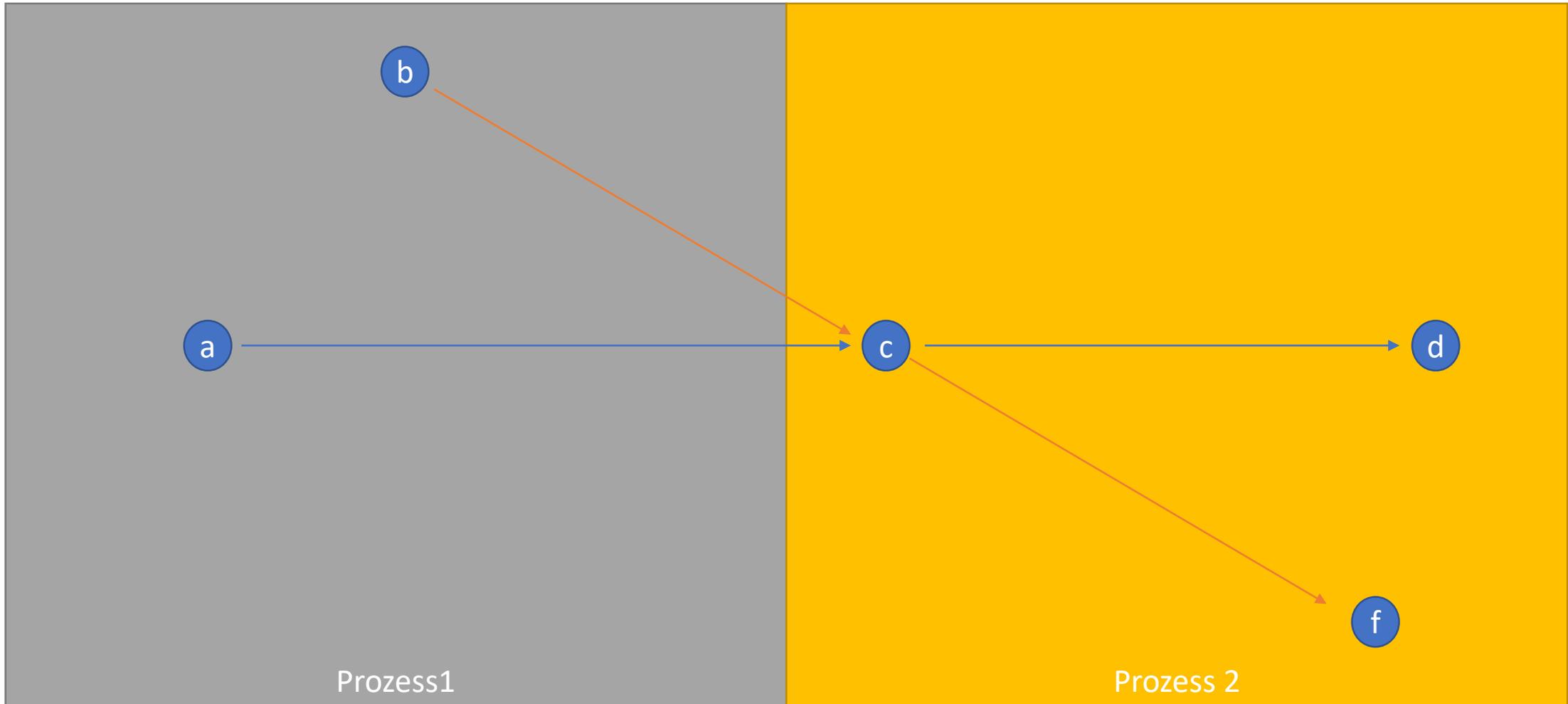


Gedachter Vorteil gegenüber Sektorenverteilung:

- Weniger Schnittstellen zwischen Prozessen
- Nur benötigte Schnittstellen
- Wenig Kommunikation bei Aktualisierung des Kantengewichtes, da Route möglichst Konsistent einem Prozess zugeordnet wird.(WIP)



Parallelisierung - Simulation



Parallelisierung - Simulation

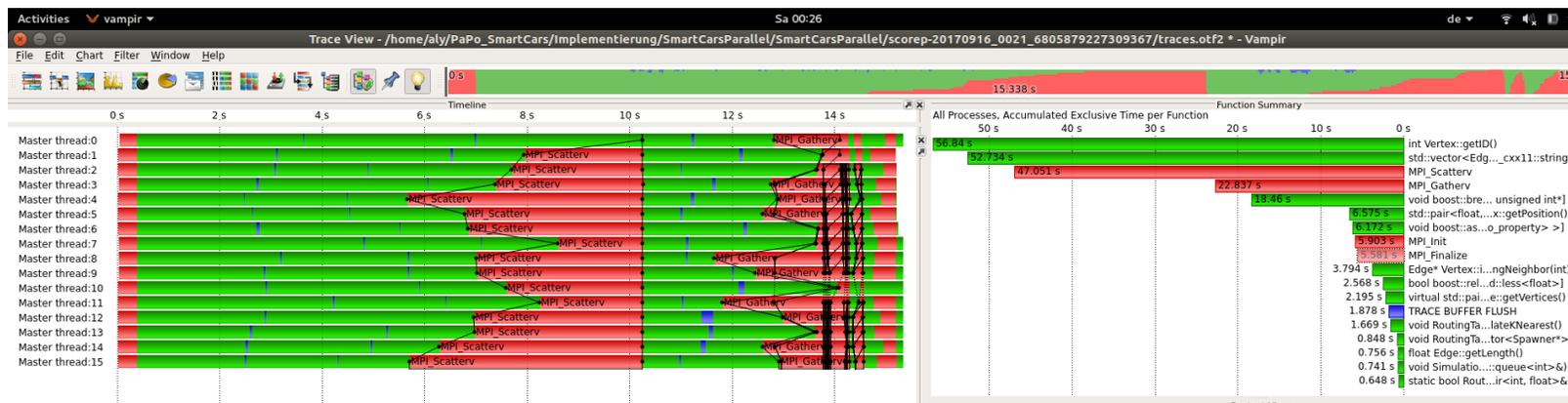
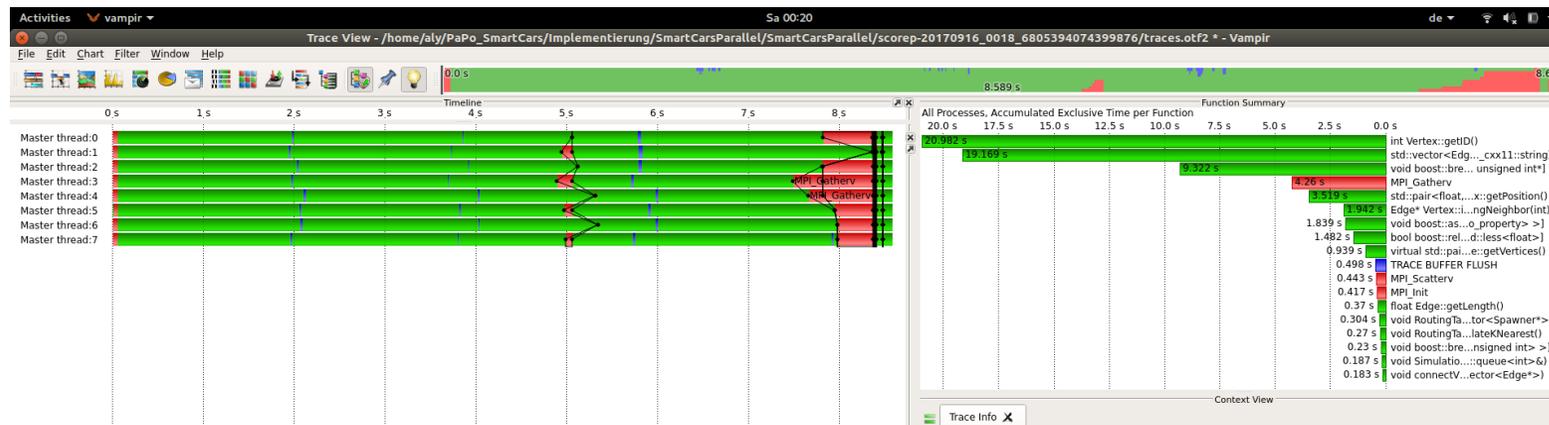


- Simulation in Prozessen erfolgt sequentiell
- Vor jedem Update: Austausch von Kapazitäten
- Transfer zwischen Prozessen nach jedem Update:
 - Auto wird in Buffer auf Vertex gespeichert
 - Buffer im Prozess werden gesammelt
 - `MPI_Send(buffer,...,...)` an jeden verbundenen Prozess und `MPI_Recv(...)`



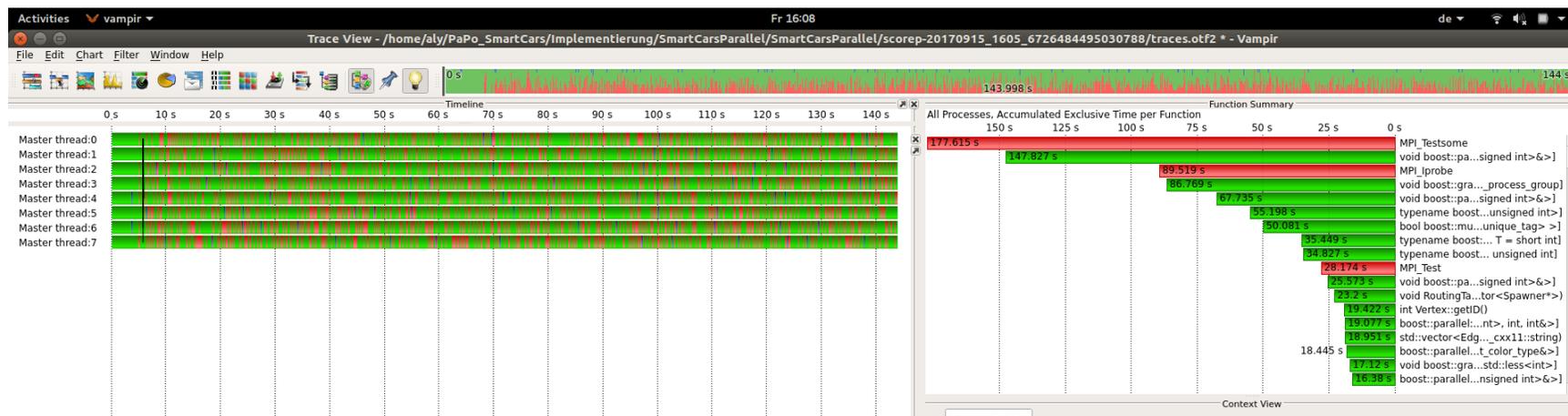
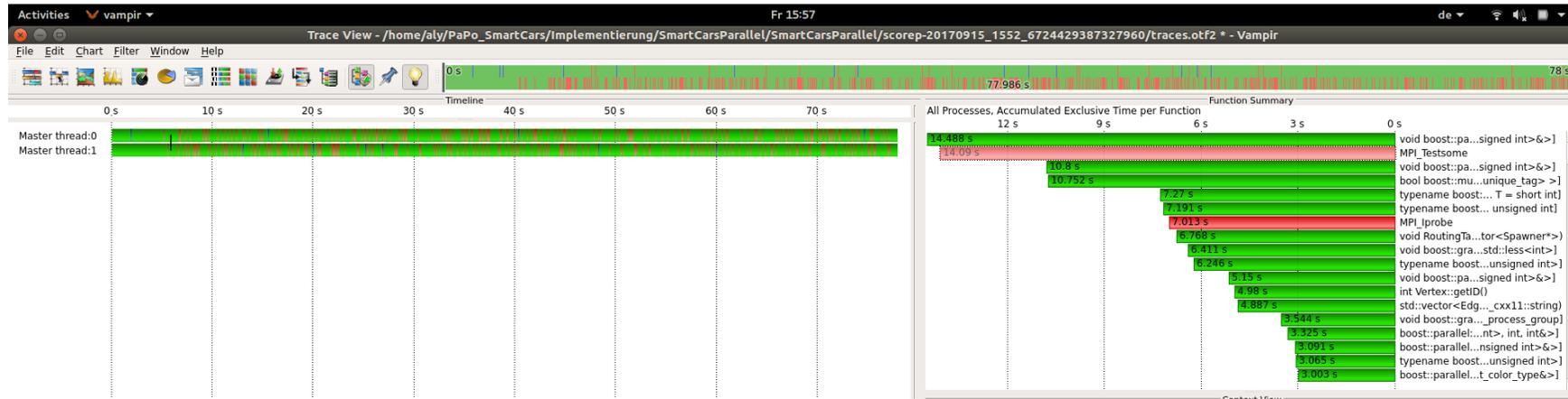
Performanceanalyse und Skalierbarkeit

RoutingTable a), Analyse mit Vampir und Oprofile:

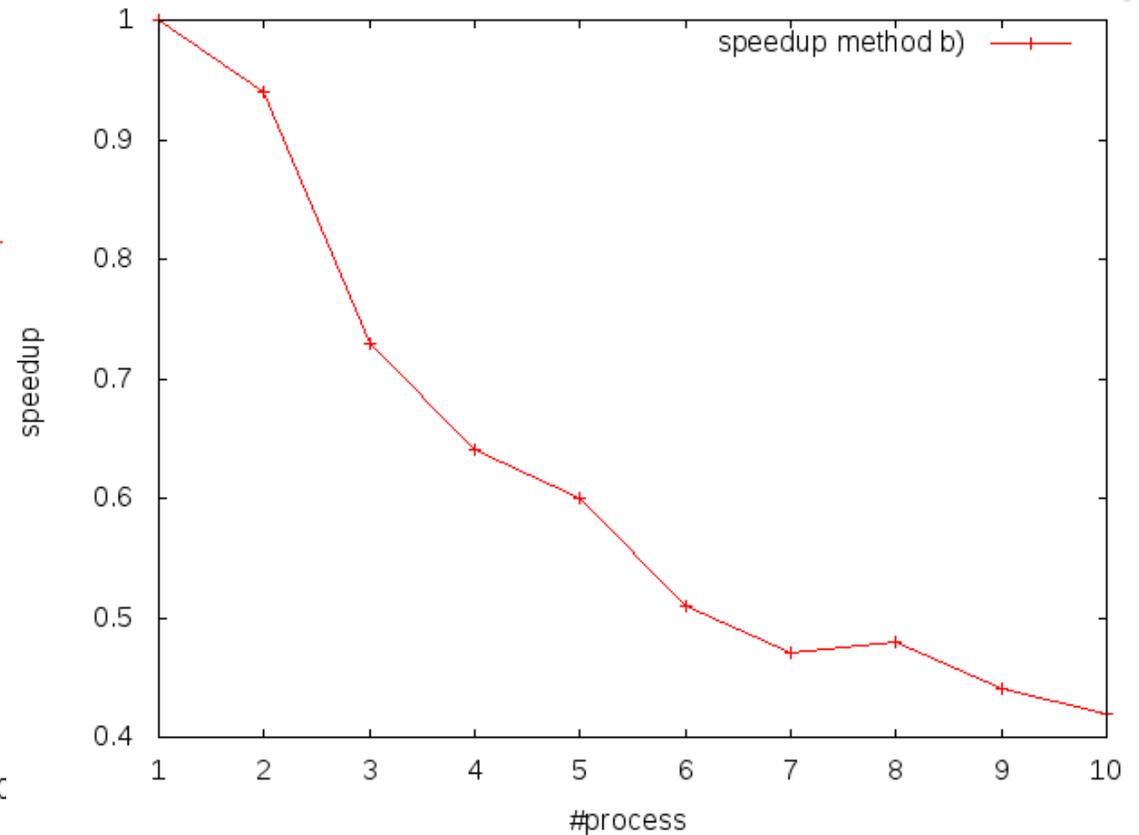
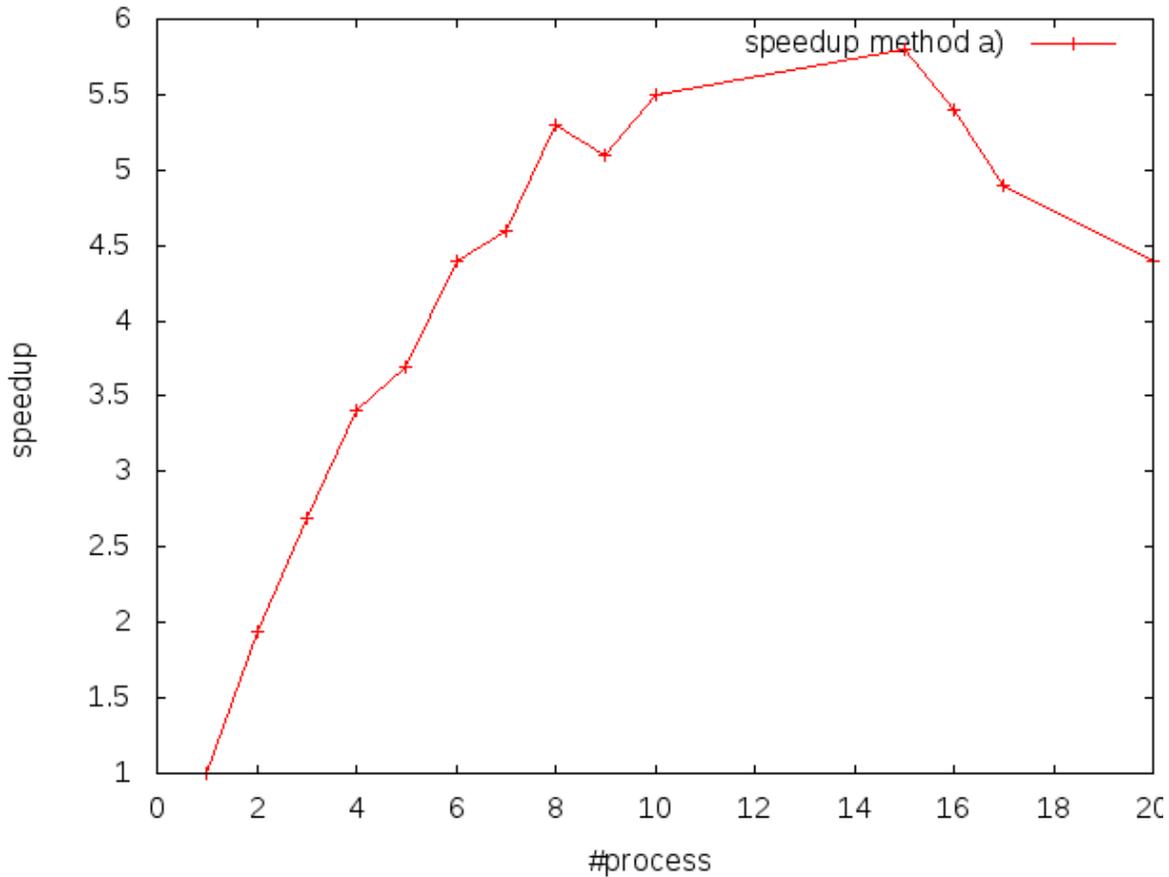


Performanceanalyse und Skalierbarkeit

RoutingTable b), Analyse mit Vampir und Oprofile:



Performanceanalyse und Skalierbarkeit



alternative Karte: 16700 Knoten, 39982 Kanten bis 3 Prozesse getestet, bei $n = 3$ Speedup von **0.984**

Verbesserungen und Ausblick



- Bessere Heuristik für TimeTables finden
- Funktion einbauen, dass alle n-ten optimalen Routen gespeichert und verwendet werden
- Parallel BGL: Eager Dijkstra oder Delta Routing verwenden, da besseres strong scaling
- Wiederholte Updates optimieren

Quellen

- [1] Wikipedia, Routing table (17.09.2017) https://en.wikipedia.org/wiki/Routing_table
- [2] OpenStreetMap <https://www.openstreetmap.de/>
- [3] Boost c++ Libraries: <http://www.boost.org/>
- [4] Parallel Boost Graph Library <http://www.osl.iu.edu/research/pbgl/>
- [5] Parallel Boost Graph Library Overview
http://www.boost.org/doc/libs/1_65_1/libs/graph_parallel/doc/html/overview.html
- [6] Zbigniew J. Czech (2016): 'Introduction to Parallel Computing', S. 69f.
- [7] Dijkstra shortest paths, Parallel BGL
http://www.boost.org/doc/libs/1_65_1/libs/graph_parallel/doc/html/dijkstra_shortest_paths.html