Verkehrssimulation zur Stauerkennung auf Basis von OpenStreetMap Karten

— Ausarbeitung —

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Martin Poppinga, Marcel Ellermann und Miriam Freiin Heereman von Zuydtwyck
E-Mail-Adresse:	1popping@informatik.uni-hamburg.de 1ellerma@informatik.uni-hamburg.de 1heerema@informatik.uni-hamburg.de
Matrikelnummer:	6318650, 6315155, 6318979
Studiengang:	Informatik
Betreuer:	Julian Kunkel

Hamburg, den 13.10.2013

Abstract

In dieser Arbeit wird ein System zur Verkehrssimulation auf Basis von OpenStreetMap Daten implementiert und Möglichkeiten zur Parallelisierung aufgezeigt, sowie diese implementiert. Die verschiedenen Implementationen werden dann hinsichtlich ihrer Leistung und Skalierbarkeit verglichen.

Inhaltsverzeichnis

1. Einleitung

Auf dem Öffentlichen Markt gibt es eine Vielzahl von Programmen zur Simulation von Straßennetzen oder zum Erkennen und vermeiden von Staus, wie zum Beispiel das Tool der Universität von Florida TRANSIT[oF11]
Wir werden in dieser Arbeit eine ähnliche Simulation implementieren, wenn doch in deutlich geringerem Umfang. Unsere Simulation soll alleine auf der Basis von OpenStreet-Map Daten arbeiten, was den allgemeinen Umfang an Informationen zu den einzelnen Straßennetzen deutlich beschränkt und auch die Korrektheit nicht garantieren kann. Dennoch wollen wir eine Qualitativ gute Implementation erarbeiten, die in annehmbarer Zeit ausgeführt werden kann. Für die ausreichende Geschwindigkeit der Simulation, sollen daher verbreitete Standards wie OpenMP[MP13] und MPI [MPI13] verwendet werden um unsere Simulation zu parallelisieren. Welche Auswirkungen und Limitationen dieser Ansatz mit sich bringt wird in späteren Kapiteln weiter diskutiert.

2. Das Serielle Programm

In diesem Kapitel soll ein Überblick zu den einzelnen Bestandteilen der Simulation gegeben werden, dazu gehören Vorverarbeitung der Daten, die Implementation der eigentlichen Simulation, sowie die Nachverarbeitung bzw. Auswertung der Daten.

2.1. Die Vorverarbeitung

Die Vorverarbeitung ist in Python 2.7 geschrieben und ist für das Importieren und die Aufbereitung der Daten verantwortlich. Als Datenquelle dienen die Karten des OpenStreetMap Projektes im OSM/XML Format. Dabei durchläuft das Programm folgende Schritte:

1. **Einlesen:** des XML Dukumentes und erstes erstellen der Datentypen:
   a) **Knoten:** Punkte auf der Karte und ihere Kooridnaten
   b) **Straßen:** Gefilterte Wege als Liste von Punkten und weitere Eigenschaften (Anz. Spuren, Geschwindigkeit, Art, Name, Richtung)
   c) **Kreuzungen:** Knoten, bei denen sich mehrere Straßen Treffen oder sich Ampeln befinden
2. Erstellen von **„Spawnern“:** Bestehend aus Typ, Größe und Kreuzungen an denen die Autos ins system gesetzt werden können
   a) Extrahieren von Wohn-, Gewerbe, Industie und Bürogebieten
   b) Suchen von Knoten in dem Gebiet
   c) Bei Fehlen von Knoten, suche von Knoten in der Nähe
3. **Aussortieren und Reduzieren** Von nicht benötigten Daten
   a) Unbenutzte Knoten und Wege
   b) Kürzen von ids
   c) Aufteilen von Straßen für einheitliches Format
   d) Entfernen von mehrfachbesetzten Spawnpunkten
4. **Anreichern von Daten**
   a) Erkennen der Position von Straßen zueinandner

b) Berechnung von den Entfernungen der Punkten auf einer Straße

c) Min und Maximalwerte

d) Vorfahrtsregelungen an den Straßen

e) Erkennen von Kreisverkehren

f) Clustern von zusammenhängenden Ampelanlagen und deren Position zueinader

5. **Schreiben** von CSV und XML Datein für die weitere Verarbeitung mit unter Anderem folgenden Bedingungen:

   a) Eine Kreuzung liegt immer zwischen zwei Straßen (also nicht in einer Straße)

   b) Die Numerierung von Straßen und Kreuzungen ist durchgehend

   c) An Kreuzungen sind die Straßen gegen den Uhrzeigersinn sortiert

   d) Eine Ampel gehthört nur zu einem Ampelcluster

   e) Von jeder Straße auf jede Andere Straße einer Kruzung gibt es abbiegeregeln

   f) An jedem Punkt kann maximal ein Spawner vertreten sein.

2.2. Die Simulation

Die Simulation selbst ist in C++11 geschrieben unter Verwendung der Boost-Libraries [Lib13]. Die Grundstruktur der Simulation ist in Abbildung 2.1 dargestellt. Zuerst werden die Nodes über den NodeHandler in die Simulation geladen und anschließend die Straßen sowie die dazugehörigen Einmündungen und Vorfahrtsregeln vom MapHandler geladen. Dabei ist zu beachten das Straßen zur Vereinfachung nur aus relevanten Knoten bestehen, das heißt solchen Knoten die entweder als Einstiegspunkt für Autos genutzt werden oder an denen Kreuzungen zu anderen Straßen sind.

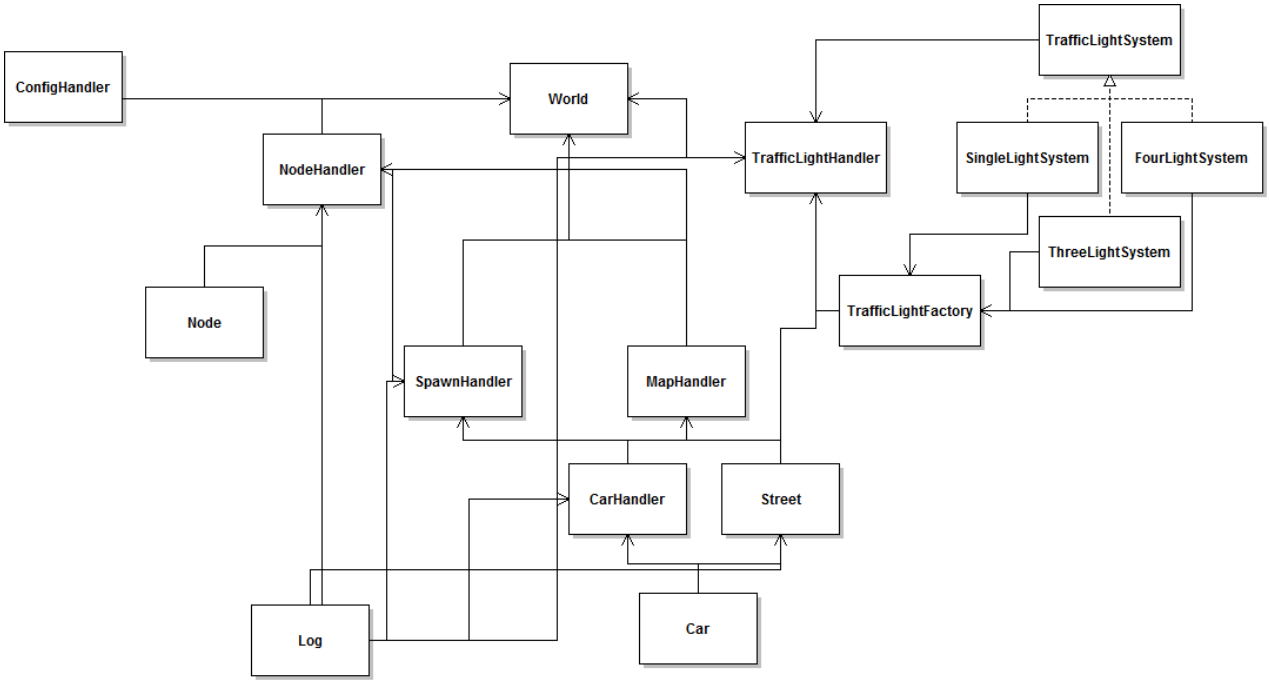

Abbildung 2.1.: Klassendiagramm der Simulation

Danach werden die Start und Zielgebiete vom SpawnHandler geladen. Als letztes werden die Ampeln geladen und Initialisiert, wobei beliebige Ampelsysteme geladen werden könnten, zur Zeit aber nur Ampelsysteme die aus einer, drei oder vier Ampeln bestehen implementiert sind. Grundlegend durchläuft die Simulation nun nach seiner Initialisierung eine Menge von logischen Zeitschritten, wobei jeder Zeitschritt einen realen Zeitraum darstellt. Der Standard für die Umrechnung ist, dass ein logischer Zeitschritt einer Sekunde entspricht, jedoch kann dieser Wert beliebig festgelegt werden. Ein Logischer Zeitschritt besteht aus verschiedenen Phasen:

1. Updaten des SpawnHandlers
   a) Entfernen von Autos die das Ende ihres Pfades erreichen
   b) Hinzufügen von neuen Autos an den Startpunkten
2. Updaten des MapHandlers
   a) Updaten der Straßen
      i. Update der Autos auf der Straße
      ii. Autos von einer Straße auf eine andere bewegen
3. Updaten des TrafficLightHandlers
   a) Umschalten von Ampeln

Das generelle Spawnverhalten wird über 2 Splines gesteuert, ein Spline beschreibt die Autopopulation im Tagesverlauf und der andere Spline beschreibt die Spawnverteilung,

siehe Abbildung 2.2. Wobei der Anfang des Graphen der Uhrzeit 03:00 Morgens entspricht.
Durch diese beiden Splines modelliert, wird jeden Zeitschritt eine nach oben hin begrenzte Menge an Autos der Welt hinzugefügt. Jedes Auto erhält dabei einen komplett vorgegebenen Pfad zu seinem Ziel, der durch einen A*-Algorithmus ermittelt wird, wobei nicht die kürzeste Route sondern die schnellste ermittelt wird. Die Autos werden alle einzeln im MapHandler-Update aktualisiert und folgen den Verkehrsregeln der deutschen Straßenverkehrsordnung. Sie lassen genug Sicherheitsabstand zu Fahrzeugen vor ihnen, falls es welche gibt, beachten Rechts vor Links, Geschwindigkeitsbegrenzungen und Ampeln. Jedes Auto besitzt ein Feld für den lokalen Zeitschritt, so das sichergestellt werden kann das ein Auto immer nur einmal pro Zeitschritt bewegt wird.

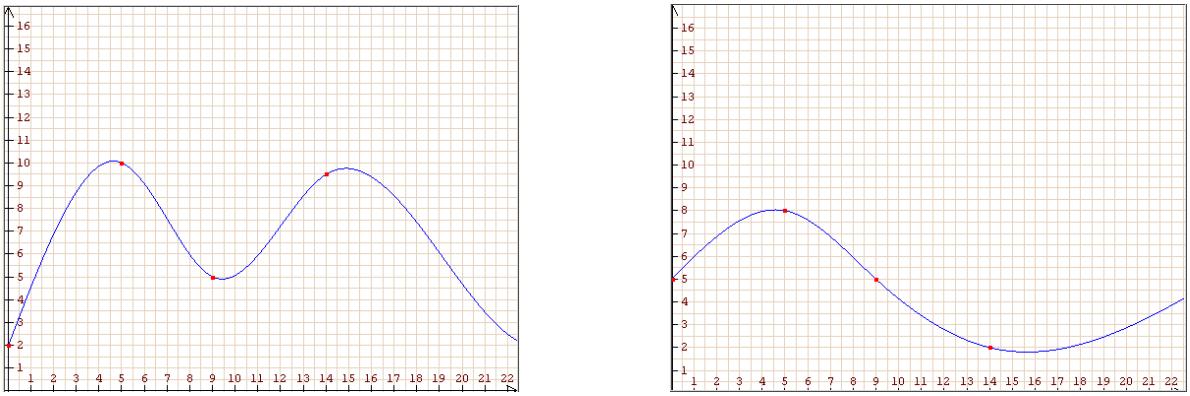

Abbildung 2.2.: Links: Graph für die Population. Rechts: Graph für die Spawnverteilung

2.3. Die Nachverarbeitung

Die Simulation gibt alle zehn Sekunden die Position alle Autos. Da aber nur die Relevanten Knoten den Straßen zugewiesen werden, kann die Ausgabe keine genauen Koordinaten der Autos beinhalten, da die Straßen in den meisten fällen zu Geraden vereinfacht wurden. Deshalb beinhaltet die Ausgabe nur die Straße und die Distanz die ein Auto vom Anfangspunkt der Straße zurückgelegt hat. Die Ausgabe ist in den cardump Dateien zu finden. Um eine genaue Position in Koordinatenform zu erhalten werden die Angaben zu Straße und Distanz im Nachhinein in die richtigen Koordinaten gerechnet und dann mittels eines von uns geschriebenen Tools daraus Bilder und Videos für die simulierte Karte erstellt. Für eine Beispiel Ausgabe für die Karte Halstenbek siehe 2.3.

Abbildung 2.3.: Beispielplot für einen Simulationsschritt auf Halstenbek

3. Die Parallelisierung und Leistungsanalyse

In diesem Kapitel werden unsere Parallelisierungsmaßnahmen beschrieben und die Unterschiedlichen Implementationen miteinander vergleichen.

3.1. Parallelisierung mit OpenMP

Den ersten Schritt zur Parallelisierung haben wir mit OpenMP bestritten, in dem wir die Kernmethoden unserer Simulation auf mehrere Threads aufgeteilt haben. Zwei große Punkte sind dabei die Parallelisierung der Wegfindung, siehe Code-Ausschnitt 3.1, sowie das Parallele Updaten der Karte, siehe Code-Ausschnitt 3.2.
In der OpenMP Variante für die Pfadfindung der Autos, wird in mehreren Threads gleichzeitig ein Pfad für jeweils ein Auto gesucht, dass dann zur Welt hinzugefügt wird. Die Pfade müssen also nicht länger Seriell gesucht werden. Da die Pfadsuche durchaus sehr Variable für größere Karten sein kann, haben wir für die schedule Option der omp Direktive in Zeile 1, dynamic verwendet damit die Zeit optimal auf die Threads aufgeteilt werden kann. Ein Weiterer Zeitfaktor den es bei der OpenMP Parallelisierung zu berücksichtigen galt, war der Zugriff auf Nodes. Da Theoretisch alle Threads gleichzeitig auf die Liste der Nodes schreibend zugreifen könnten, dabei aber nur die Werte der Elemente an sich verändern, war ein globaler Lock für die Nodeliste sehr uneffektiv, weshalb für jede Node ein eigener Lock verwaltet wird, siehe Code-Ausschnitt 3.3.

Listing 3.1: Code-Aussschnitt: Paralleles Pfadfinden - SpawnHandler.cpp

```
1 #pragma omp parallel for private(i) shared(points) schedule(dynamic)
2 for(i = 0; (ui32)i < max; i++)
3 {
4     std::pair<Node*,Street*> dPair;
5     std::pair<Node*,Street*> sPair(std::get<1>(points[i]),std::get<2>(points[i]));
6     getDestinationPoint(std::get<0>(points[i])->type, dPair);
7     std::list<std::pair<Node*, Street*>*>* path = MAP->getPath<__speedCost,
          ↪ __euclHeuristic>(sPair,dPair);
```

Listing 3.2: Code-Aussschnitt: Paralleles Update der Straßen - MapHandler.cpp

```
1 #pragma omp parallel for private(i) shared(max) schedule(guided)
2         for(i = 0; (ui32)i < max; i++)
3         {
4             _streets[i]->update(step);
5         }
```

Listing 3.3: Code-Aussschnitt: OpenMP Locks auf Node Ebene

```
#ifdef _OPENMP
        omp_set_lock(_lock_mutex[id]);
#endif
        if(_locks[id] == NULL)
        {
            _locks[id] = new std::list<nodeLock*>();
        }
        _locks[id]->push_back(nL);
#ifdef _OPENMP
        omp_unset_lock(_lock_mutex[id]);
#endif
```

3.2. Parallelisierung mit MPI

Die Parallelisierung mit MPI setzt an den gleichen Punkten an wie die Parallelisierung mit OpenMP. Dabei wird für jeden Prozess die gesamte Karte geladen, die Straßen aber nach ihrer Position in N Vektoren sortiert, wobei jeder Prozess genau für einen der N Vektor von Straßen zuständig ist. Das heißt, dass die Zielpunkte für die Autos die der Prozess generiert zwar außerhalb seines Zuständigkeitsbereichs liegen können, Startpunkte aber nicht. Auch Straßen-Updates werden nur für die Straßen ausgeführt für die er Zuständig ist. Die Aufteilung findet durch rastern der Karte statt, so kann die Karte beliebige quadratische Raster aufgeteilt werden, was die Prozesszahl auf eins, vier, neun, 16 oder andere Quadratzahlen limitiert.
Neben der Aufteilung und Zuständigkeit galt es allerdings noch ein anderes Problem zu lösen. Die Kommunikation zwischen den Prozessen. Dafür haben wir einen Message Handler geschrieben, der parallel zum Programmablauf Nachrichten empfangen und verarbeiten kann. Ein Schema zur Einbindung des Message Handlers im Programm findet sich in Abbildung 3.1.

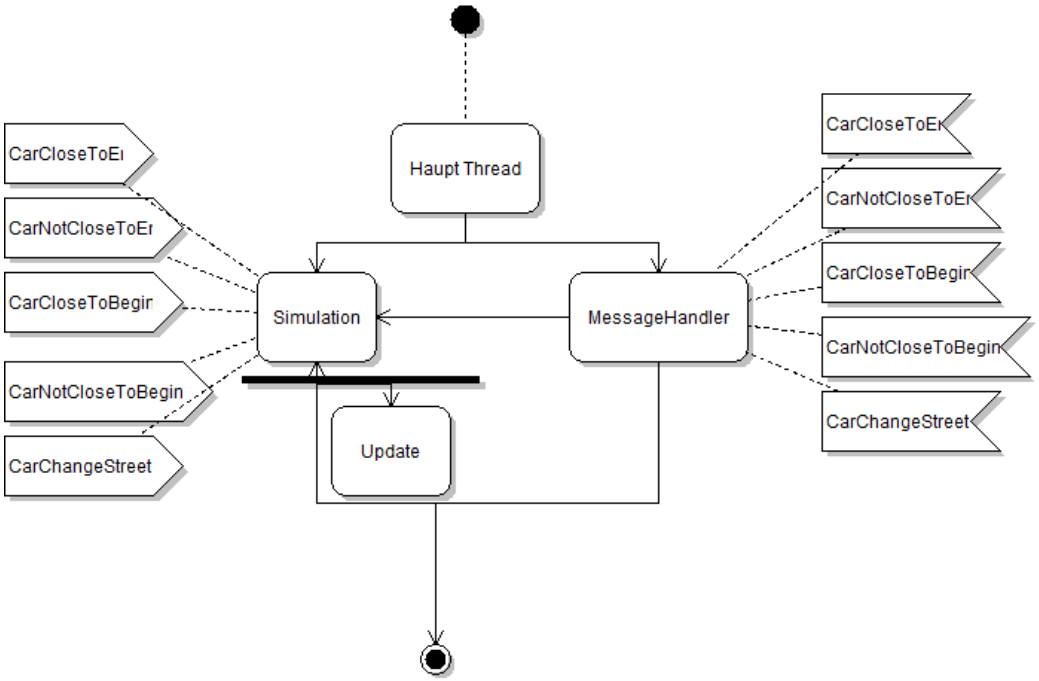

Abbildung 3.1.: Schema für den Message Handler

Dort ist zu sehen, dass Nachrichten während eines Updatezykluses jeder Zeit gesendet und empfangen werden können, wobei der Message Handler auf genau 5 Nachrichten hört. Zu beachten ist, das nach jedem Updatezyklus erst alle gesendeten Nachrichten verarbeitet werden müssen bevor die Prozesse mit dem nächsten Zyklus fortfahren können. Dies wird in Zeile 153 und 154 der World.cpp sicher gestellt. Siehen Code-Ausschnitt 3.4.

Listing 3.4: Code-Aussschnitt: Synchronisierung der Prozesse mittels des Message Handlers

```
1 void world::update(boost::uint8_t step)
2     {
3         [...]
4         MSG->WaitMsgsSend();
5         MSG->Synchronize();
```

4. Analyse und Vergleich der einzelnen Implementationen

In diesem Kapitel werden die unterschiedlichen Implementationen untersucht und untereinander vergleichen.

4.1. Strong Scaling

Für das Strong Scaling haben wir eine Maximale Anzahl von 75.000 Autos auf Hamburg für 10.800 Zeitschritte simuliert. Diese Simulation haben wir für einen, vier, neun und 16 Prozessoren durchlaufen lassen und das Ergebnis in 4.1 festgehalten.

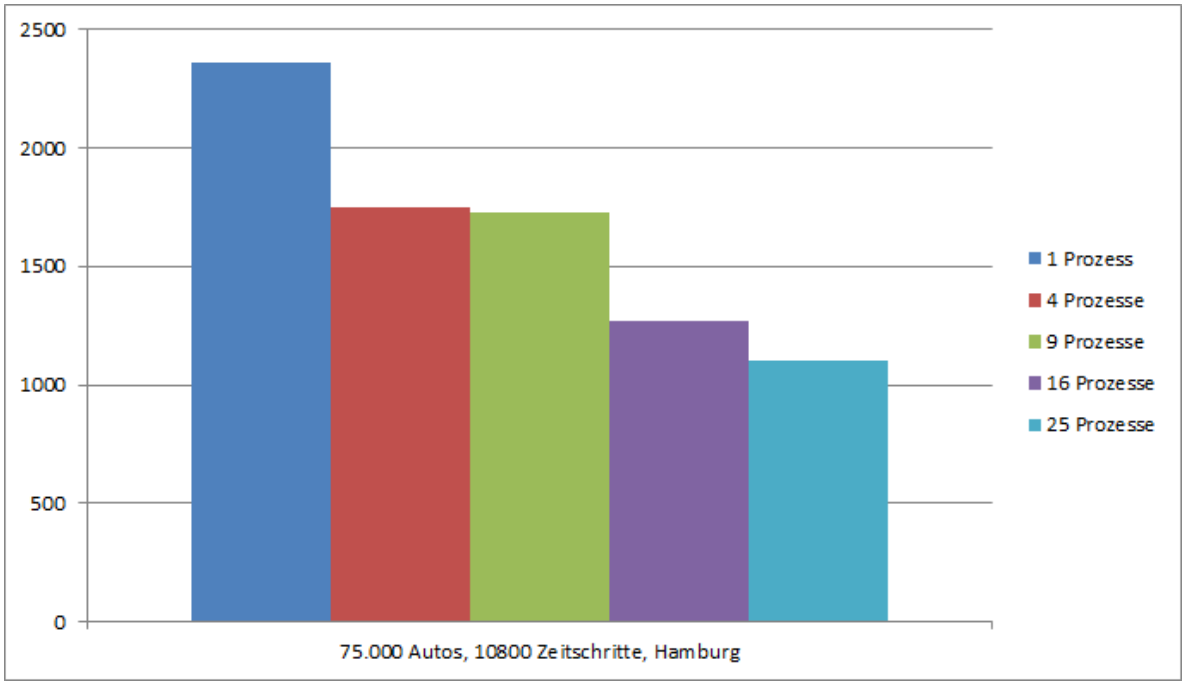

Abbildung 4.1.: Strong Scaling auf Hamburg. 75.000 Autos, 10.800 Zeitschritte

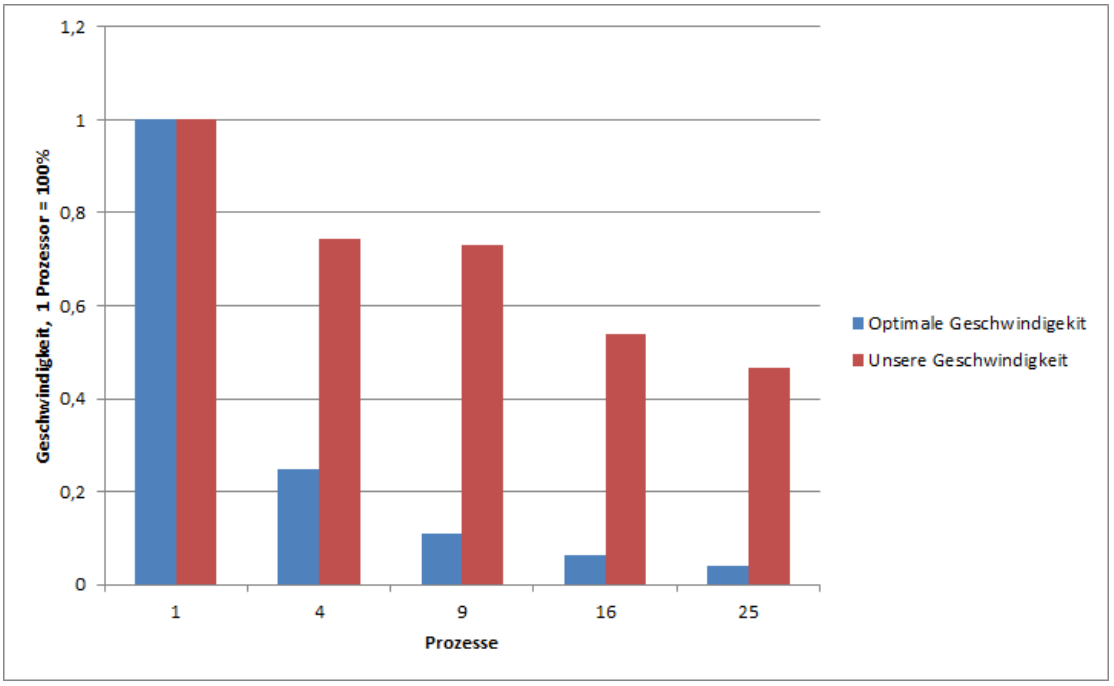

Abbildung 4.2.: Speedup unseres Programms, 1 Prozess = 100%

Wie an 4.1 und 4.2 zu sehen ist, erhalten wir zwar eine Laufzeitverbesserung bei steigender Prozess Anzahl, allerdings ist der Laufzeitgewinn nicht optimal. Dies liegt vor allem an der suboptimalen Nutzung von MPI, wie im Nächsten Kapitel noch genauer zu sehen sein wird. Denn Leider musste der Message Handler so konzipiert werden das nur serialisiert MPI Methoden aufgerufen werden, da MPICH2 für uns nicht auf dem Cluster zum laufen zu kriegen war. Trotz der vorgeschlagenen Veränderungen am Jobskript gab es nur Fehlermeldungen von uns, die auftreten während der MPI_Init Methode, also außerhalb unseres Codes. Für die Fehlermeldung inklusive des **hostname** Parameters im Aufruf von **mpiexec** siehe Anhang A - Ausschnitt A.1. Für die Fehlermeldung ohne den Parameter siehe Anhang A - Ausschnitt A.2. Das Jobskript befindet sich unter Anhang B.

4.2. Profiling

Im Pofil-Ausschnitt 4.1 ist deutlich zu erkennen das ein Großteil der Zeit für die Synchronisierung der OpenMP-Prozesse verloren geht. Dies geschieht an der Stelle an der die Pfadfindung an mehrere Threads delegiert wird. Da durch die Vorverarbeitung allerdings nicht ausgeschlossen wird, dass Pfad von einer Node zu einer anderen Existiert, kommt es vor das sämtliche Nodes der Karte überprüft werden müssen, was für Hamburg etwa 170.000 sind, wodurch die Laufzeit für die Pfadsuche deutlich höher ist als die Laufzeit für die Suche der anderen Pfade.

Listing 4.1: Code-Aussschnitt: VampirTrace Pofil der Simulation für 10 Zeitschritte

```
1                                          excl. time  incl. time
2 *excl. time  incl. time      calls       / call      / call  name
3   344.901s    344.901s         10     34.490s     34.490s   !$omp ibarrier
4    33.979s     76.915s       0.08    407.754s    922.974s   main
5     6.213s     14.564s  17455937.67      0.356us     0.834us [boost::list::iterator],
```

5. Fazit

Auch wenn sich die Last nicht gleichmäßig auf die verschiedenen Prozessoren verteilen lässt und so die Prozesse zum Teil auf andere warten müssen, war durch die Paralellisierung ein deutlicher Geschwindigkeitsgewinn gegenüber der Serriellen Variante zu verzeichnen.

Bisher Ungelöste Probleme:
Beim Preprocessing ist die Laufzeit bei großen Karten sehr hoch, dies liegt zu großen Teilen am Clustering der Ampeln, da hier bei entsprechnder Anzahl viele Positonsvergleiche notwendig sind und der Suche nach zugehörigen Punkten der Spawner. Durch den einsatz von Cython lässt sich die Laufzeit um ca. 1/3 reduzieren, was bei großen Karten wie einer Karte von ganz Deutschland aber immernoch sehr hoch ist. Problematisch wird hier auch in der jetztigen implemantation der Arbeitsspeicherverbrauch von ca. 30GB, Sollte man die Simulation auf ganz Deutschland laufen lassen wollen, muss dies noch optimiert werden. Weiterhin Problematisch ist die Laufzeit beim Postprocessing; das Gernerieren der Bilder wird mit einem einfachen plottingtool erledigt, was zwar gut funktioniert aber unnötig lange benötigt. So dauert das erstellen eines Frames in hoher Auflösung mehere Sekunden.

Die Darstellung im allgemejnen ist nicht sehr überisichtlich, so wäre mit einer weiteren Übersichtskarte indem farblich Stauschwerpunkte hervorgehoben werden, einfacher zu erkennen wo gerade Staus enstehen.

Ausblick: Die Verkehrsimmulation ist schnell sehr komplex geworden, das eine parallelisierung an vielen stellen erschwerte, dennoch gibt es einige Punkte in denen man die Simulation noch realitätsnäher gestalten könnte.

1. Autos die Automatisch auf Stau reagieren und ihn umfahren
2. Noch realistischere Start- und Zielpunkte
3. Autos mit unterschiedlichen Geschwindigkeiten
4. Intelligente Ampelschaltungen (z.B. Grüne Welle)
5. Realistischere Vorfahrtsregelungen

Literaturverzeichnis

[Lib13] Boost Libraries. Boost c++ libraries. 07 2013.

[MP13] Open MP. Open mp. 07 2013.

[MPI13] Open MPI. Mpi 1.6 documentation. 07 2013.

[oF11] University of Florida. Traffic network study tool. 05 2011.

Abbildungsverzeichnis

Listingverzeichnis

A. MPICH2 Fehlermeldungen

Listing A.1: Error File, beim Ausführen auf dem Cluster mit MPICH2, inklusive hostname Parameter

```
1  hostname: the specified hostname is invalid
2  hostname: the specified hostname is invalid
3  [proxy:0:1@west2] HYD_pmcd_pmip_control_cmd_cb (./pm/pmiserv/pmip_cb.c:883): assert (!closed) failed
4  [proxy:0:1@west2] HYDT_dmxu_poll_wait_for_event (./tools/demux/demux_poll.c:77): callback returned error status
5  [proxy:0:1@west2] main (./pm/pmiserv/pmip.c:210): demux engine error waiting for event
6  [proxy:0:3@west4] HYD_pmcd_pmip_control_cmd_cb (./pm/pmiserv/pmip_cb.c:883): assert (!closed) failed
7  [proxy:0:3@west4] HYDT_dmxu_poll_wait_for_event (./tools/demux/demux_poll.c:77): callback returned error status
8  [proxy:0:3@west4] main (./pm/pmiserv/pmip.c:210): demux engine error waiting for event
9  srun: error: west2: task 1: Exited with exit code 7
10 srun: error: west4: task 3: Exited with exit code 7
11 [mpiexec@west1] HYDT_bscu_wait_for_completion (./tools/bootstrap/utils/bscu_wait.c:76): one of the processes
       ↪ terminated badly; aborting
12 [mpiexec@west1] HYDT_bsci_wait_for_completion (./tools/bootstrap/src/bsci_wait.c:23): launcher returned error
       ↪ waiting for completion
13 [mpiexec@west1] HYD_pmci_wait_for_completion (./pm/pmiserv/pmiserv_pmci.c:216): launcher returned error waiting for
       ↪ completion
14 [mpiexec@west1] main (./ui/mpich/mpiexec.c:325): process manager error waiting for completion
```

Listing A.2: Error File, beim Ausführen auf dem Cluster mit MPICH2

```
1  *** glibc detected *** /home/ellermann/papo/papo/Software/Traffic_Simulation_Seriell/src/Simulation: double free or
       ↪ corruption (fasttop): 0x00000000016df140 ***
2  ======= Backtrace: =========
3  /lib/x86_64-linux-gnu/libc.so.6(+0x7eb96)[0x7fc4a2ad8b96]
4  /opt/mpich2/1.5/lib/libmpich.so.8(MPIDI_Populate_vc_node_ids+0x3f9)[0x7fc4a3afddd9]
5  /opt/mpich2/1.5/lib/libmpich.so.8(MPID_Init+0x136)[0x7fc4a3af86c6]
6  /opt/mpich2/1.5/lib/libmpich.so.8(MPIR_Init_thread+0x202)[0x7fc4a3ba82f2]
7  /opt/mpich2/1.5/lib/libmpich.so.8(MPI_Init_thread+0x73)[0x7fc4a3ba8533]
8  /home/ellermann/papo/papo/Software/Traffic_Simulation_Seriell/src/Simulation[0x406f80]
9  /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xed)[0x7fc4a2a7b76d]
10 /home/ellermann/papo/papo/Software/Traffic_Simulation_Seriell/src/Simulation[0x407041]
11 ======= Memory map: ========
12 00400000-0045f000 r-xp 00000000 00:14 8530651
       ↪ /home/ellermann/papo/papo/Software/Traffic_Simulation_Seriell/src/Simulation
13 0065f000-00660000 r--p 0005f000 00:14 8530651
       ↪ /home/ellermann/papo/papo/Software/Traffic_Simulation_Seriell/src/Simulation
14 00660000-00661000 rw-p 00060000 00:14 8530651
       ↪ /home/ellermann/papo/papo/Software/Traffic_Simulation_Seriell/src/Simulation
15 016dd000-016fe000 rw-p 00000000 00:00 0
```

B. Job Batch File

Listing B.1: Batchfile für die Ausführung auf dem Cluster

```
1  #!/bin/sh
2
3  #SBATCH --time=100
4  #SBATCH  --error=job.err --output=job.txt
5  #SBATCH -N 4 -n 4
6
7  alias module="/usr/bin/modulecmd bash"
8  which mpiexec
9  module load mpich2/1.5
10 which mpiexec
11
12 mpiexec -np 4
       ↪ /home/ellermann/papo/papo/Software/Traffic_Simulation_Seriell/src/Simulation
```