A Parallel Implementation of a Schedule-Based Transit Assignment Algorithm for Large Networks

Markus Friedrich, Gerd Schleupen
{friedrich | schleupen}@isvs.uni-stuttgart.de
Institut für Straßen- und Verkehrswesen, Universität Stuttgart
Seidenstraße 36, D-70174 Stuttgart

Michael Moltenbrey
michael.moltenbrey@ipvs.uni-stuttgart.de
Institut für Parallele und Verteilte Systeme, Universität Stuttgart
Universitätsstraße 38, D-70569 Stuttgart

Hans-Joachim Bungartz
bungartz@in.tum.de
Institut für Informatik, Technische Universität München
Boltzmannstraße 3, D-85748 Garching

Abstract

Handling traffic requires proper planning and simulation. Therefore, one needs to model the journeys of travellers. Assignment procedures are used for that. However, obtaining high quality results in large networks is computationally intensive and requires a sophisticated implementation of algorithms. We describe and analyze a parallel implementation of an transit assignment algorithm.

1 Introduction

Assignment procedures model and simulate the journeys of travellers in transport networks from origin to destination. As a result the procedures provide not only traffic volumes for links and transit lines but also indicators describing the service quality of a network.

Assignment procedures for transit networks need to model the spatial and temporal structure of traffic supply as well as travel costs. Line routes and timetables must be examined in great detail in order to mirror exact transfer times and timetable co-ordination. This type of *schedule-based* search algorithm can be found in passenger information systems provided by an increasing number of transit operators (e.g. www.bahn.de). Here, the search algorithms are applied and optimized for a one-to-one type of problem. Assignment in transport planning, however, needs to examine an entire network and the connections between all

traffic zones. This extends the search to a many-to-many problem, which can produce long computing times for big nation-wide networks.

Obtaining high quality results (e.g. identification of fast but expensive and/or slow but cheap alternatives) and coping with the resulting problem size, requires specific algorithms and a sophisticated implementation. Objective of the proposed paper is to describe the implementation of an existing schedule based assignment algorithm using parallel implementation techniques.

2 Algorithm

The algorithm for the schedule based transit assignment (see [1]) consists of three successive steps, which are described below:

2.1 Preprocessing

The objective of the preprocessing step is the generation of *connection segments*, which describe a part of a journey. These segments are the building blocks of connections in the search process. A connection segment represents a transfer-free ride on one transit line. A connection made up of an access walk, a bus ride and an egress walk, for example, consists of three connection segments.

From the line route data, an initial step generates the network's *route segments*. A single route segment describes a segment of a transit line between a boarding stop and an alighting stop. It has an initial stop node, a terminal stop node, a length and a running time.

Subsequently, *connection segments* are calculated from the set of route segments. It additionally carries information on the departure and arrival time.

2.2 Connection Search

A dynamic, i.e. time-dependent, multi-path algorithm is applied to determine all potential connections. This algorithm builds a connection tree which may contain several paths (connections) from the origin to a destination node. Since the tree's width largely depends on the number of runs of the transit lines, it may be much wider than a usual shortest-path tree. On the other hand, the use of entire connection legs as tree edges simplifies the tree's structure to a great extent and limits its depth by the maximum number of transfers. A typical value of this user-defined constant is 4 or 5.

Figure 1 outlines the structure of the connection tree. The root of the tree is the centroid node of the origin zone and the only outgoing branch represents the access from the origin to the first transit stop.

Starting from the tree's origin node, a branch & bound strategy is employed. Given a connection segment from the current tree level, all possible successors are considered. Every connection segment s is described by means of the following attributes:

- departure time x_s and arrival time y_s

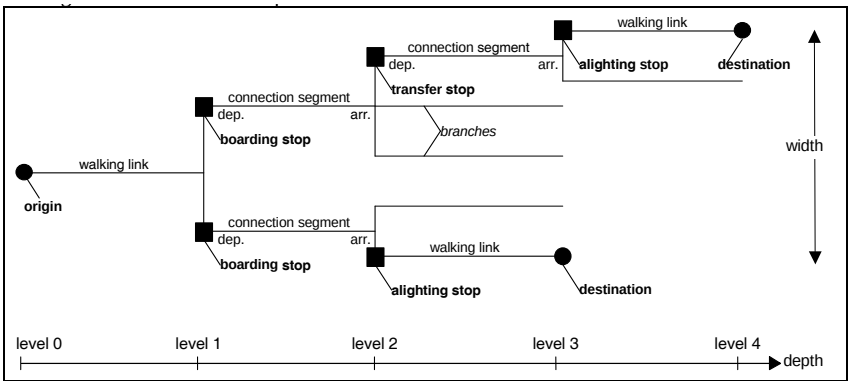

Figure 1: Structure of the connection tree

- travel time t_s
- fare f_s

Naturally, these quantities are also defined for connections (i.e. *sequences* of connection segments). Furthermore, if a connection c is composed of n connection segments, the number of transfers is $n_c = n - 1$.The resulting search impedance of a connection c is then defined as:

$$\nu_c = \chi^t t_c + \chi^n n_c + \chi^f f_c \;\; (\chi^t \geq 0, \chi^n \geq 0, \chi^f \geq 0)$$

where the coefficients are user-defined parameters.

Now, let s be the currently processed connection segment from network node a to network node b. Let c^* be the new connection between the origin node and b formed by adding s to some connection c arriving at a. Finally, let C_b be the set of all known connections to b.

Connection segment s is inserted into the tree as an successor of c, if the following conditions hold:

- *Temporal suitability:* The connection segment s departs from node a only after the arrival of connection c plus a minimum transfer wait time and before a user-defined maximum transfer wait time has elapsed.
- *Dominance:* There is no known connection $c' \in C_b$ that is faster, contains less transfers and is cheaper.
- *Tolerance constraints:* User-defined tolerances compared to the best known connections are not exceeded.
- *Loops:* Transfers *within* a transit line are only allowed for lines containing a loop, provided that the passenger can save time by boarding an earlier vehicle trip (run) at the intersection node of the route loop.

The resulting set of connections is the input for a connection choice model. The choice model distributes the travel demand onto the set of connections determined in the search process. It reflects interaction effects between competing alternatives. For a description of the choice model refer to [1, 2].

3 Use Cases

The algorithm is ideal for railway networks where headways are long and the coordination of the timetable is important. One use case is the application of the algorithm to the German railway network. An interface to the data source provided by passenger information systems is used to build a railway network model which contains all long distance trains and relevant local trains. This railway network is integrated with a digital road network to provide a solid basis for multi-modal network analysis and traffic flow simulation.

The common ground for the railway and road network are 6,929 traffic-zones where trips start and terminate. Except for the zones there is no coherence between these 2 parts of the network. Each network has its own links and nodes.

The railway network is based on approximate 15,000 nodes and 32,500 links. 13,700 of the nodes serve as stations, the remaining nodes represent junctions. The daily timetable is described by 34,385 vehicle journeys which were transferred from the passenger information system.

- 734 high-speed trains (e.g. ICE, Thalys, TGV, Eurostar, Cisalpino)
- 1,302 rapid long-distance trains (e.g. IC, EC)
- 1,257 long-distance trains (e.g. IR, IRE, D)
- 21,897 local trains (e.g. RE, RB)
- 9,195 suburban trains (S-Bahn)

This sums up to approx. 3.0 Mio vehicle-km traveled on an average working day. For this the total running time is 44,000 hours ($\approx$ 68km/h mean speed).

There are two main difficulties in building up such a big railway network. The first difficulty concerns the integration of the timetables into the railway network and the second is to generate automatically the connectors which connect the right stations with the right zones.

Integrating the timetable requires a matching between the timetable station names of the information system and the station names of the network. Without this allocation it is not possible to transfer the timetables into the network. The allocation is simple when the stations have the same name in both systems. But only about 50% of the station names match perfectly. For the rest it was necessary to develop other methods to get a higher matching rate.

Two methods are applied to make an alignment between the timetable stations and the network stations. The first method is to replace special parts in a name. This method has the

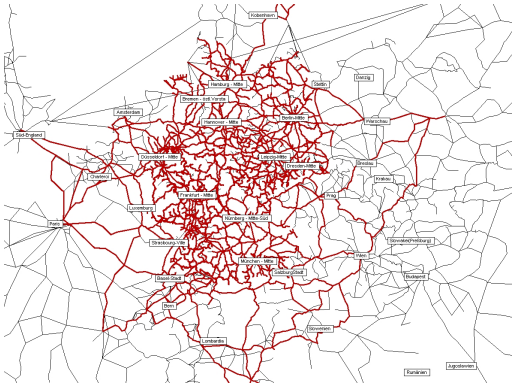

Figure 2: Service Frequency in German Railway Network

disadvantage that only defined changes can be regarded, other changes will be ignored. For this reason a dynamical alignment method, called “Needleman-Wunsch-Algorithm” was adapted. This algorithm has its origin in the alignment of DNA-Sequences. It allows to compare two strings on similarity under the use of a scoring matrix, which has evaluated values for each combination of two characters. In the case of making an alignment between timetable stations and network stations, it can be said that this method can be used additionally to the first alignment which compares the two strings on total similarity.

The second difficulty is the generation of connectors in the railway network. The reason why it is so difficult to generate connectors is that the direct-distances between the centroid of a zone and the associated stations have wild fluctuations. The reason is that the centroid of a zone is the geometrical balance-point of the zone polygon and not the balance-point of the infrastructure. That for example the zone named “Stuttgart-Mitte” will be connected to the right stations, it is also possible to connect zones to the stations whose names have a great affinity to the zone names, combined with a limitation in the direct distance. The method for the measure of the affinity is the same like in the alignment of the train stations, but with a lower scoring limit.

4 Parallelization Strategies for the Implementation

Larger problems and thereby larger networks require certain techniques for enhancing the computational speed. One possibility is the use of parallelization techniques. A first solution to this problem is outlined in this section.

The investigated branch & bound algorithm is optimized for sequential computations and performs very good in that case (see [1]). However, this optimization and its development for the primarily sequential usage entails some drawbacks that make a parallel processing of the entire algorithm very difficult. Thus, only parts of it can be taken into account for parallelisation. We have identified the connection search as a particularly promising part

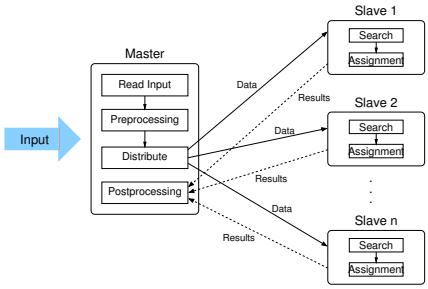

Figure 3: Outline of an algorithm path

of the algorithm both in terms of speed-up potentials and in terms of parallelizability. The resulting parallel algorithm with dynamic load balancing is described in the following.

The basic approach is to distribute the work of computing all connection trees among the different participating processors p. Thereby, each processor only computes a subset of these connection trees.

The basic workflow of our implementation is outlined in Figure 3. The first general step comprises the build-up process of the traffic network. Therefore, all necessary data are read from appropriate input files in order to build the network (read data). Parallelization of this step is not very useful since the build-up process is a strict sequential workflow. The preprocessing step as outlined in the previous sections is also done in the master process completely analogous to the sequential algorithm (see [1]).

After this preprocessing, we start to distribute the created data among the processes. Different connection trees can be computed independently. Connection segments are not computed separately since they can easily be constructed with the help of route segments and the timetable information. This saves computation time and memory, as well.

All necessary information is broadcasted to each of the p processors and is used to build a simplified traffic network on each processor (distribute). The simplified network contains the whole traffic network. However, it is reduced to the minimum data required. In fact, we only need the data computed in the preprocessing step and timetable information, as well. An explicit representation of the network graph is not required. To support efficient communication, we use appropriate and compact data structures for broadcasting (e.g. arrays of basic data types).

After a simplified traffic network is build on each processor, our implementation starts to distribute the network nodes in their function as source nodes for connection trees. However, it is not useful to statically assign each process a specific subset of the nodes. The computation time for building a connection tree is likely to vary in dependence of source nodes. Therefore, we apply a dynamic strategy. In a first approach, we use a master-slave model for load balancing, i.e. a process requests a new source node as soon as it has finished to compute the connection tree of its current source node. In the long run, each processor p_i is assigned a certain subset s_i of all network nodes n.

Each processor computes the connection tree for its current source node. Afterwards,

the actual assignment step takes place, i.e. the total number of trips is distributed among the connections as described in Section 2.2. We use an empirical time variation curve to split the demand for each time interval (see [2]).

The results of each assignment step (carried out once for each connection tree) are stored in the simplified network representation of each process. If no more network nodes are available for processing, i.e. all connection trees and assignments have been computed, each of the p processes transmits its total assignment results to the master. There, all data are collected in a postprocessing step in which the results are prepared for further processing.

5 Results

We tested our implementation on different architectures and with a varying number of processors. For test purposes, we used a eight node Linux cluster with dual Intel Pentium III processors with 800 Mhz. The second Linux cluster used is equipped with 64 nodes, each with dual Intel Xeon 3 Ghz processors. The results of our measurements are depicted in Figure 4.

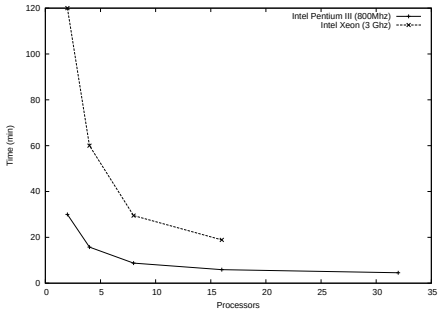

Figure 4: Runtime measurements

The graph suggests an almost constant runtime and thereby no improvement with an increasing number of processes. However, it has to be taken into account that parsing the network and the demand file plays an important role in the runtime behavior. Both files are required for building the traffic network and for the assignment step. This phase lasts about 150 seconds and cannot be improved (linear time in parsing). Taking a closer look at the runtimes for 16 and 32 processors on the 64 node cluster (354 seconds and 273 seconds respectively), still shows partially the desired behavior, i.e. doubling the number of used processors results in halved runtime.

A problem of our implementation gets obvious for smaller numbers of processors. As outlined in the description of our implementation in the previous section, each processor collects all of its computation results and sends the data back to the master only at the end

of the overall computation. Having smaller numbers of processors results in larger amounts of data that is stored at each processors memory. This causes longer transmission times for each message transmitted to the master. If we only have few nodes, the message size can be large – depending on the traffic network.

Another problem is the applied load balancing strategy which turns out to become a bottleneck with an increasing number of processors. This is due to the general problems of the used master-slave model.

6 Discussion

The current implementation of the presented algorithm shows good characteristics. However, there is still enough room for optimization. A different load balancing strategy is required when using this algorithm with a higher number of processors.

Another limiting factor is the size of the messages that have to be transmitted between the processors. As outlined in the previous section, the fewer the number of processors, the larger is the size of these messages. Therefore, we examine strategies to reduce the message sizes. Particularly, two strategies need to be considered: the reduction of the messages that transmit the simplified network to the slave processors and the strategy for returning the computed data to the master processor.

In addition, different strategies for partitioning the network are under consideration which may reduce the amount of data transferred considerably. Furthermore, other algorithms may show better characteristics for parallelization, e.g. adpated shortest path algorithms.

Nevertheless, our implementation reduced the computation time from more than one hour (sequential algorithm) to a few minutes which is a noticeable improvement. However, it has to be noted that the pure sequential algorithm includes a lot of optimization which increase the algorithm's overall performace. These optimizations cannot be used for parallelization sometimes do hinder an effective parallelization.

7 Acknowledgement

This work is funded by the Landesstiftung Baden-Württemberg within the framework of a program called "Modellierung und Simulation auf Hochleistungsrechnern". We would also like to thank PTV AG for their cooperation.

References

[1] *Friedrich, M.; Hofsäß, I.; Wekeck, S.:* Timetable-based Transit Assignment Using Branch & Bound Techniques. Transportation Research Records 1752 (2001).

[2] *Schnabel, W.; Lohse, D.:* Grundlagen der Straßenverkehrstechnik und der Verkehrsplanung. Verlag Bauwesen, 1997.