A Graph Partitioning Algorithm for Parallel Agent-Based Road Traffic Simulation

Yadong Xu
TUMCREATE Ltd
1 CREATE Way
Singapore 138602, Singapore
yadong.xu@tum-create.edu.sg

Wentong Cai
Nanyang Technological University
50 Nanyang Avenue
Singapore 639798, Singapore
aswtcai@ntu.edu.sg

David Eckhoff
TUMCREATE Ltd
1 CREATE Way
Singapore 138602, Singapore
david.eckhoff@tum-create.edu.sg

Suraj Nair
TUMCREATE Ltd
1 CREATE Way
Singapore 138602, Singapore
suraj.nair@tum-create.edu.sg

Alois Knoll
Technische Universität München
Boltzmannstraße 3
85748 Garching b. München, Germany
knoll@in.tum.de

ABSTRACT

A common approach of parallelising an agent-based road traffic simulation is to partition the road network into subregions and assign computations for each subregion to a logical process (LP). Inter-process communication for synchronisation between the LPs is one of the major factors that affect the performance of parallel agent-based road traffic simulation in a distributed memory environment. Synchronisation overhead, i.e., the number of messages and the communication data volume exchanged between LPs, is heavily dependent on the employed road network partitioning algorithm. In this paper, we propose Neighbour-Restricting Graph-Growing (NRGG), a partitioning algorithm which tries to reduce the required communication between LPs by minimising the number of neighbouring partitions. Based on a road traffic simulation of the city of Singapore, we show that our method not only outperforms graph partitioning methods such as METIS and Buffoon, for the synchronisation protocol used, but also is more resilient than stripe spatial partitioning when partitions are cut more finely.

Keywords

Neighbour-Restricting Graph-Growing; parallel simulation; agent-based traffic simulation; graph partitioning

1. INTRODUCTION

Agent-based road traffic simulation has become an important tool in the evaluation of today's and future transportation systems. It is useful in solving many severe problems that modern large cities face such as increasing traffic congestion and high CO_2 emissions. To this end, simulating entire cities [14, 22] with thousands to millions of agents (i.e., vehicles) can give valuable insights, however, at the same time it poses a major challenge in terms of computational resources.

Parallel computing techniques can be used to speed-up these simulations. In a parallel agent-based road traffic simulation, computational workload is divided and executed by a group of Logical Processes (LPs), each of which is assigned to a physical processing unit. To maintain the correctness of the parallel simulation, inter-process communication is required due to data dependencies between LPs [4]. This is referred to as *synchronisation* and, in distributed memory environments, is typically achieved by message passing between the LPs. Due to the synchronisation between the LPs, it is crucial to consider load-balancing, so that the waiting time of each LP is reduced. The workload of traffic simulation is often dynamic, which necessitates dynamic load-balancing during simulation run-time. Inter-process communication and load imbalance are the two major factors that affect the performance of parallel simulation, and both of them are influenced by how the simulation is partitioned.

A common way to parallelise traffic simulation is to decompose the road network into multiple spatial subregions (i.e., partitions) and assign each partition to an LP. The most straightforward method of achieving this is through the use of geographical information, e.g., by cutting the network into stripes, grids, or areas of equal sizes [1, 12, 13, 23]. The downside of these methods is that they do not consider synchronisation overhead. Another approach is to convert the road network into a graph (where edges represent road segments and nodes represent connections between these segments) and then use graph partitioning algorithms [15, 20]. Graph partitioning algorithms use certain heuristics aiming to reduce edge-cut and thereby the dependencies among partitions. However, depending on the synchronisation protocol, minimising edge cut alone may not minimise the total synchronisation overhead [7].

SIGSIM-PADS '17, May 24–26, 2017, Singapore.

DOI: http://dx.doi.org/10.1145/3064911.3064914

In this paper, we propose Neighbour-Restricting Graph-Growing (NRGG), a graph partitioning algorithm that reduces synchronisation overhead by limiting the number of neighbouring LPs, that is, LPs exchanging messages. Two partitions are called *neighbouring partitions* if there exists at least one road link that connects the two partitions, thus creating data dependencies between the two LPs that execute them. Reducing the number of neighbours for each partition can therefore reduce the required synchronisation messages, especially when an asynchronous protocol is used for synchronisation. We achieve this by developing a two-step algorithm that first partitions the road network into stripe-like shaped regions, and then refines the partitions using a modified Kernighan-Lin (KL) local search algorithm. Our contributions can be summarised as follows:

- We present Neighbour-Restricting Graph-Growing (NRGG)[1], a novel two-step graph partitioning algorithm for parallel road traffic simulation.
- We show the applicability and performance of our proposed algorithm by applying it to *double-blinded* [24], a parallel agent-based microscopic road traffic simulator for distributed memory environments.
- We compare our method to the existing algorithms and show that it is not only more scalable than stripe spatial partitioning, but also leads to fewer synchronisation messages compared to the graph partitioning methods METIS [9] and Buffoon [18].

The reminder of this paper is organised as follows: Section 2 presents the related work. Section 3 provides some preliminary information for the work in this paper, including a brief description of the agent-based traffic simulation that the partitioning algorithm operates on, and the formulation of the partitioning problem we attempt to solve. Section 4 proposes the partitioning algorithm. Then experiments are described in Section 5. Lastly, we draw conclusions and discuss future work in Section 6.

2. RELATED WORK

To split the computational workload between LPs, road networks can be partitioned in various ways. For example, the road network can be cut into stripes or grid cells using coordinate information [12,13]. The resulting subregions do not necessarily need to have the same geometric area. They are weighted, e.g., in terms of traffic density to achieve a low workload imbalance between the LPs. Another method is recursive coordinate bisection (RCB) [1, 23], where the road network is cut into two equally sized sub-regions by a plane orthogonal to one of the coordinate axes. This procedure is then recursively applied until the desired number of sub-regions are obtained. The downside of cutting the road network into stripes, grids or using RCB method is that they mainly focus on workload balancing and do not consider the minimisation of synchronisation overhead. In an environment where message passing is time-consuming, ignoring synchronisation overhead can be significantly detrimental to the performance of parallel simulation.

Graph partitioning algorithms try to not only equalise the workload of partitions, but also minimise edge cut among partitions. Cutting an edge in a road network means that connected road segments geographically lie in two partitions, therefore requiring communication between the two responsible LPs. The graph partitioning problem is NP-hard, therefore heuristics are required to obtain approximate results. Various graph partitioning algorithms can be found in the literature [7, 9, 10, 18]. An example is the well-known multilevel partitioning approach [7, 9, 10]. In multilevel partitioning, the graph is first recursively coarsened to a smaller graph. Then a graph partitioning algorithm such as spectral bisection [16] or evolutionary algorithm [3, 17, 18] partitions the smaller graph. Finally, the smaller graph is uncoarsened back to the original graph. At each level of uncoarsening, local search refinement is commonly used to improve the quality of partitions. The Kernighan and Lin algorithm [10] is the classic algorithm for local refinement, where two vertices at the boundary of two neighbouring partitions are exchanged to reduce edge cut.

METIS is a widely used set of programs for partitioning graphs. It adopts the multi-level approach and has been used to parallelise traffic simulation [15, 20]. It was shown that graph partitioning outperforms spatial partitioning algorithms in terms of edge cut, which potentially reduces synchronisation overhead between LPs. Buffoon [18] is also a multi-level graph partitioning algorithm which uses natural cuts in road networks as a preprocessing technique to obtain a coarser graph. It is able to generate less edge cut compared to METIS for road networks. However, the algorithm is slower than METIS, which may make it unsuitable for dynamic partitioning at simulation run-time. To receive a better understanding of the performance of NRGG, we compare our proposed method to both METIS and Buffoon in Section 5.

The discussed graph partitioning algorithms in this section have in common that they aim at minimising edge cut. However, depending on the synchronisation protocol, reducing edge cut does not always lead to minimising synchronisation overhead, as also pointed out by Hendrickson and Kolda [7]. To address this problem, we extend the state of the art by also considering the number of neighbouring partitions to reduce synchronisation overhead.

3. PRELIMINARIES

In this section, we introduce the parallel agent-based traffic simulation platform that we use. The synchronisation protocol is described, which motivates our partitioning algorithm. We also formulate the partitioning problem.

3.1 Agent-based Road Traffic Simulation

3.1.1 Simulation space

A road network, containing links and nodes, is a *spatial network* that forms the simulation space. Links represent roads in the real world and can have one or more lanes, and nodes contain the connectivity information of links. Nodes possess geographical coordinate information, i.e., longitudes and latitudes. An agent always has to be situated on a link, making links effectively a container of agents.

3.1.2 Agents

An *agent* in the simulation represents a driver-vehicle unit. The behaviour of agents is usually modelled using car-following models [5,21] and lane-changing models [6,11]. Car-following

[1]Source code of implementation in C++ is available at https://github.com/xu-yadong/nrgg.

models calculate the velocity and acceleration of a vehicle according to the characteristics of the driver and the vehicle, and the surrounding traffic conditions. Commonly, car-following models adjust the velocity of an agent to maintain or reach a desired safety gap to the vehicle in front. Lane-changing models calculate whether an agent should change lanes, e.g., based on the current speed and on vehicles on other lanes. These models require agents to have a *sensing range*, which is the area in the road network around the agent within which other agents may have an effect on the agent's behaviour. The agent needs to examine the traffic conditions within its sensing range to make acceleration and lane-changing decisions. This is challenging in parallel traffic simulation when the sensing range is reaching into other partitions as it then potentially requires synchronisation between the responsible LPs.

3.1.3 Agent state variables

Each agent has a *state* at a particular virtual simulation, represented by state variables. Agent states potentially change upon executing *time-stamped events* scheduled by agent models. The simulation advances by executing these events in the ascending time-stamp order. State variables can be classified into two types depending on their visibility: *Agent-based state variables* belong to the agent as a whole and are visible to other agents, e.g., velocity and geographical location; while *component-based state variables* belong only to the models inside the agent and are not visible to other agents, e.g., state-of-charge of vehicle battery. We assume that agent-based state variables are updated periodically with a fixed interval, which is called an *update interval*, a.k.a., *time-step*. The events that change agent-based state variables may have an effect on other LPs, thus they affect the synchronisation between LPs. Other events that change component-based state variables are internal to an LP.

3.2 Parallelisation and Synchronisation of Logical Processes

3.2.1 Parallelisation

To parallelise the simulation, the road network is decomposed into multiple spatial subregions, i.e., partitions. The network is divided by cutting links. Links that are cut and therefore lie in two partitions are named *boundary links*. A boundary link is evenly divided between two partitions.

An LP is responsible for executing the events for the agents (e.g., moving the agent along a link) in one subregion. An agent is *local* to an LP if it is inside the subregion that belongs to the LP.

There are data read and write dependencies between neighbouring LPs. During the simulation, when an agent moves beyond the boundary of partition i and enters the area of partition j ($i \neq j$), the agent *migrates* from LP_i to LP_j. The migration of agents incurs a data *write dependency* between the two LPs. Migrated agents are destroyed in the original LP and recreated with all their state variables in the new LP. If there is an agent A in LP_i inside the sensing range of another agent B in LP_j, agent B should be aware of the agent-based state variables of agent A. To achieve this, a *proxy agent* is created in LP_j that mirrors agent A. It possesses exactly the same agent-based state variables as agent A. Hence, the agent-based state variables of agent A should be sent by LP_i to LP_j to keep the state of the proxy agent updated. In this case, there is a data *read dependency* between LP_i and LP_j. The agent-based state variables of agent A are *shared states*.

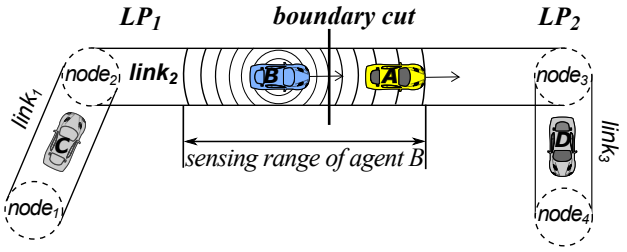


Figure 1: Road network partitioning with boundary cut on $link_2$.

An illustration of the above concepts is shown in Figure 1. $Link_2$ is a boundary link between LP_1 and LP_2. Agent A in LP_2 is inside the sensing range of agent B in LP_1, hence, the agent-based state variables of agent A are shared states. There is a proxy of agent A in LP_1. If agent B migrates into LP_2, the whole agent B, including all state variables and model parameters should be sent to LP_2 by LP_1. Agent B will be destroyed in LP_1 and recreated in LP_2. There are read and write dependencies between LP_1 and LP_2.

3.2.2 Synchronisation

Data read and write dependencies necessitate the communication between LPs. An LP should not progress the simulation over the point when a read or write dependency happens until the dependency is fulfilled by exchanging messages with the relevant LPs. In our use-case, *synchronisation* is the sending and receiving of migrating agents and shared states. We adopt a conservative approach where no violations of dependencies may ever occur during the simulation. The synchronisation between LPs is carried out using a Mutual Appointment (MA) protocol [26], which we will now briefly describe.

The progression of the simulation in LP_i using the MA protocol is shown in Algorithm 1. The simulation iteratively executes a synchronisation event and an update event for every update interval. Associated with each synchronisation event, there is a set of LPs that currently have appointments with LP_i, denoted as Syn_i^t. Syn_i^t may include all, none, or only a subset of the neighbouring LPs of LP_i. For each LP_j in the set Syn_i^t, LP_i sends one compound message containing migrating agents, shared states and lookahead. *Lookahead* of LP_i towards LP_j ($i \neq j$) at simulation time t is a time interval in the simulated future within which LP_i will not have data dependencies with LP_j. LPs estimate their lookahead by predicting when there will be agents to migrate and states to be shared. When Syn_i^t is empty, no message-passing occurs for LP_i at time t. After messages are received, the next appointment is made according to the lookahead, by adding the LP to the future $Syn_i^{t+\Delta t}$ set, where Δt is the mutual lookahead.

Using the MA synchronisation protocol, the number of synchronisation messages is affected by the number of neighbouring partitions and lookahead. Communication data volume is determined by the number of migrating agents and shared states. Sending synchronisation messages requires a

Algorithm 1: Simulation progression in LP_i using the MA protocol.

Definitions:
$l^t_{i,j}$ lookahead from LP_i to LP_j at simulation time t
Syn^t_i LPs having appointments with LP_i at time t
T_{end} simulation ending time
δ update interval

initialise $t \leftarrow 0$;
initialise Syn^0_i as all neighbouring LPs of LP_i;
while $t < T_{end}$ **do**
    `// synchronisation event`
    foreach $LP_j \in Syn^t_i$ **do**
        send migrating agents, shared states, and current lookahead (i.e., $l^t_{i,j}$) to LP_j;
        prepare to receive a message from LP_j;
    end
    wait for all message sending and receiving to finish;
    update the local agent set and proxy agent set;
    foreach $LP_j \in Syn^t_i$ **do**
        add LP_j to $Syn^{t+\Delta t}_i$, where $\Delta t = \min(l^t_{i,j},\ l^t_{j,i})$;
    end
    `// event for updating agent-based states`
    update the states of local agents for this update interval;
    $t \leftarrow t + \delta$; `// time-stamp of the update event`
end

start-up time and a transmission time. More synchronisation messages will require more start-up time, and a larger data volume means longer transmission time. Therefore, synchronisation overhead should consider both the number of messages and communication data volume.

3.2.3 Dynamic load-balancing

In addition, workload imbalance also affects the performance of parallel simulation in a distributed memory environment. Workload refers to the processing of the agent events. A workload imbalance can occur if some LPs are responsible for more agents (namely, the events of the agents) than others, which in turn leads to idle waiting times at the synchronisation stage of the simulation. The way the road network is partitioned has a considerable influence on both synchronisation overhead and workload distribution. Thus, the deployed partitioning algorithm plays an essential role in the performance of parallel agent-based road traffic simulation.

Often, the workload of LPs changes in traffic simulation as new agents are created and agents migrate between LPs. This necessitates dynamic load-balancing during the simulation, which can be achieved either by completely reapplying the partitioning algorithm to partition the simulation, or by incrementally changing the shapes of partitions (i.e., diffusive methods) [19].

The problem of dynamic load-balancing is a challenge on its own and it involves additional considerations such as workload exchange and initiation rules for load-balancing operations [2]. In this paper, we focus on the partitioning problem, meaning should dynamic load-balancing be required, we reapply the partitioning algorithm to partition the road network.

3.3 Partitioning Problem

3.3.1 Preprocessing

To use a graph partitioning algorithm, the road network needs to be converted to a weighted graph $G=(E, V)$ first, where E and V are the sets of edges and vertices, respectively. One node in the road network is mapped to one vertex in the graph. The links between two nodes in the road network are mapped to one edge between two vertices. After the graph is decomposed, the partitions are then mapped back to the road network.

Each link in the road network has some workload information and data dependency information, e.g., based on the expected traffic density and flow, or the length and the number of lanes. Based on this, we add weight to vertices and edges of the graph. Edges have weights which encode the communication data volume due to data dependencies, if the the edges are cut. Traffic density and traffic flow on their corresponding links in the road network can be used for calculating the weights. Vertices have weights which encode the workload on the connecting links of their corresponding nodes. Traffic density on the links in the road network can be used for calculating the weights. For a more detailed explanation of the pre-processing steps, we refer readers to [25].

When repartitioning is required during the simulation due to dynamic workload, weights of vertices and edges are recalculated using run-time traffic density and flow information on the road network.

3.3.2 Partitioning as an optimisation problem

A graph partitioning algorithm cuts G into I disjoint partitions, $\mathcal{G} = \{G_0, G_1, ..., G_{I-1}\}$. Edges of the weighted graph are cut. Let the set of vertices in partition G_i ($0 \leq i < I$) be V_i, then $V=\cup_{i=0}^{I-1} V_i$ and $V_i \cap V_j = \varnothing$ ($0 \leq j < I$, $i \neq j$). In order to increase the performance of the parallel simulation, we consider the following three optimisation objectives during the partitioning process:

The first objective is to minimise workload imbalance. Let W_i be the total weight of the vertices in V_i. Then, the first objective can be formulated as:

$$Obj_1 = \arg\min_{\mathcal{G}} \Big(\max_{0 \leq i < I} (W_i) - \overline{W} \Big) \tag{1}$$

where $\overline{W}$ is the average weight of all partitions.

The second objective is to minimise the total number of neighbouring partitions. To the best of our knowledge, this requirement has not yet been considered in other graph partitioning algorithms for traffic simulation. We minimise the number of neighbouring partitions to reduce the synchronisation overhead in road traffic simulation. Let $N=(n_{i,j})$, $0 \leq i < I$, $0 \leq j < I$, be a matrix of connectivity of partitions. If G_i and G_j are neighbours, $n_{i,j}=1$. Otherwise, $n_{i,j}=0$. The, the second objective can be formulated as:

$$Obj_2 = \arg\min_{\mathcal{G}} \Big(\sum_{i=0}^{I-2} \sum_{j=i+1}^{I-1} n_{i,j} \Big) \tag{2}$$

The third objective is to minimise edge cut which is the total weight of edges between all partitions. Let the weight of all edges between partition G_i and G_j be $W_{i,j}$ ($i \neq j$). Then, the third objective is formulated as:

$$Obj_3 = \arg\min_{\mathcal{G}} \Big(\sum_{i=0}^{I-2} \sum_{j=i+1}^{I-1} W_{i,j} \Big) \tag{3}$$

Since the optimisation problem is multi-objective and NP-hard, finding the optimal solution may be impractical. A common approach is to use heuristics to obtain a reasonably good result.

4. NEIGHBOUR-RESTRICTING PARTITIONING

We now present the main contribution of this paper, which is the graph partitioning algorithm NRGG, short for, Neighbour-Restricting Graph-Growing. It contains an initial partitioning phase and a refinement phase. The initial partitioning phase cuts the road network, trying to achieve Obj_1 and Obj_2. The refinement phase tries to achieve Obj_3, as well as improve Obj_1 if Obj_1 achieved in the initial partitioning phase is unsatisfactory, e.g., an imbalance threshold is exceeded.

4.1 Graph-grow Partitioning

4.1.1 Graph-growing algorithm

The graph generated from the road network is first partitioned by graph-growing. Starting from an initial vertex, subgraphs are grown one by one along the edges of the graph. It can be easily proven that for a 2D space, cutting the space into stripes generates the smallest number of neighbouring partitions. In order to limit the number of neighbouring partitions, we apply this idea for partitioning a graph, i.e., generate stripe-like partitions.

The graph-growing algorithm is shown in Algorithm 2. The first step is to select an initial vertex for graph-growing. The initial vertex should be an extreme point of the graph, so it is the furthest vertex along the graph-grow direction. It can be determined using the geographical coordinates of the vertex.

Then, the second step is to grow graphs starting from the initial vertex. A *priority queue* is used to control the order of vertices to be visited. The initial vertex is pushed into the priority queue (line 13) and marked as *enqueued* (line 14). Then, vertices are iteratively popped from the front of the queue (line 16). Based on the cumulative weight and the current vertex weight, it is determined whether a new partition should be initiated (lines 17 - 24). If the number of already generated partitions does not exceed the targeted total number ($i < I$), a new partition is initiated under two conditions (line 18):

1. the accumulated weight of vertices is equal to or greater than the average weight; or
2. the accumulated weight plus the weight of the current vertex is greater than the average weight, and a randomly generated number in the range $[0, 1]$ is less than 0.5 (the exact average weight cannot be achieved, thus the current vertex is assigned randomly to either the current or a new partition, i.e., a new partition is initiated with a probability of 0.5 in this case).

Then, the current or the new partition id i is assigned to the vertex (line 25). All adjacent vertices of the current

Algorithm 2: Partitioning a weighted graph into I partitions with graph-growing.

```
1  Definitions:
2  i      id of the partition
3  I      total number of partitions
4  W_i    total weight of vertices in the partition G_i
5  W̄      average total weight of vertices per partition
6  W̃      cumulative weight of the current partition i
7  w_v    weight of vertex v in the graph
8  Q      priority queue in which the tuples are ordered
9         according to the partition the vertices connect to
10        and the coordinates of the vertices
11 initialise i ← 0, W̃ ← 0, Q ← ∅;
12 determine the initial vertex v_init for graph-growing;
13 push tuple (i, v_init) into Q;
14 mark v_init as enqueued ;
15 while Q is not empty do
16     pop a tuple from Q, and let the vertex be v;
17     generate a random double number rdm in the
        range [0,1];
18     if
        (W̃ ≥ W̄ ∨ (W̃ + w_v > W̄ ∧ rdm < 0.5)) ∧ i < I − 1
       then
19         W_i ← W̃; // new partition is initiated
20         W̃ ← w_v;
21         i ← i + 1;
22     else
23         W̃ ← W̃ + w_v;
24     end
25     assign vertex v to the ith partition;
26     foreach vertex u, where e(u, v) ∈ E do
27         if u has not been enqueued before then
28             push tuple (i, u) into Q; // sorted
29             mark u as enqueued;
30         end
31     end
32 end
```

vertex that have not been enqueued before are pushed to the queue and marked as enqueued (lines 26 - 31). A vertex is enqueued together with partition id i, forming a *tuple* (lines 13 and 28). Partition id i in the tuple marks that the vertex being enqueued has an adjacent vertex that has already been assigned to partition i. This information is used to reduce neighbouring partitions (details will be explained later in this section).

Graph-growing terminates when every vertex is assigned a partition id. Graph-growing can reach all vertices, because the weighted graph is a connected graph. The complexity of this graph-growing algorithm is $O(|E|+|V|\cdot\log_2|Q|)$, where $|E|$ and $|V|$ are the number of edges and vertices in the graph respectively, and $|Q|$ is the average queue length. Each edge or vertex is visited exactly once, so traversing the graph takes $O(|E|+|V|)$ time. The time for sorting the priority queue can be achieved in $O(\log_2|Q|)$ whenever a tuple is pushed, thus the total time for sorting is $O(|V|\cdot\log_2|Q|)$. Due to the low time complexity, heuristics to decrease the size of the graph for reducing the partitioning time, such as multi-level coarsening [9], are not required.

4.1.2 Sorting rules of vertices during graph-growing

The essence of graph-growing lies in sorting the vertices in the priority queue. Each item in the priority queue is in fact a tuple (i, v) consisting of the partition id a vertex is assigned to and the vertex itself. The vertices are sorted primarily according to the partition ids marked in their tuples and secondarily according to their coordinates. Rules for comparing vertices are as follows: for two tuples (i, v_a) and (j, v_b), (i, v_a) ranks in front of (j, v_b) in the queue, if

1. $i < j$; or
2. $i = j$ and $|v_a.crd - v_{init}.crd| < |v_b.crd - v_{init}.crd|$, where $v_x.crd$ is the coordinate of vertex v_x along the graph-grow direction and v_{init} is the initial vertex; or
3. $i = j$ and $|v_a.crd - v_{init}.crd| = |v_b.crd - v_{init}.crd|$ and $v_a.\lambda < v_b.\lambda$, where $v_x.\lambda$ is another attribute of vertex v_x that can break the tie between the two vertices.

The first rule has a direct effect on the number of neighbouring partitions: According to Algorithm 2, when the (i+1)th partition is initiated, there exists a set of vertices in the queue marked with tuple partition ids equal to (or less than) i. These vertices are adjacent to the vertices at the boundary of the previous partition (or partitions). They will be ranked higher than any vertices pushed later to the queue, namely, vertices whose adjacent vertices are in the previous partitions are popped first and grouped into as few partitions as possible.

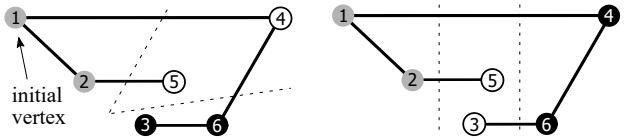
The second rule sorts vertices by their coordinates if they have the same associated partition id in tuples. Vertices that are closer to the initial vertex along the graph-grow direction are ranked higher. The graph-grow direction, as the name indicates, is a general line along which partitions expand. The use of one-dimensional distances along this line leads to stripe-like shaped partitions. For example, if *coord* is the coordinate in the horizontal direction, resulting partitions are positioned generally next to each other along the horizontal direction.

The third rule states another customisable rule in order to deterministically sort two vertices if they have equal partition ids and coordinates. For instance, it can be the degree or the id of the vertices.

4.1.3 Comparison with stripe spatial partitioning

The presented graph-growing algorithm is different from simply cutting the graph directly into stripes. A comparison is shown in Figure 2, where a small graph is partitioned into three partitions. Each partition contains two vertices.

Figure 2(a) shows the result when applying the proposed graph-growing algorithm. Vertex 1 is the initial vertex. We assume the cutting direction to be horizontal from left to right, i.e., vertices further left rank higher than those on the right side. Hence, the vertex visiting order of our algorithm is 1, 2, 5, 4, 6, and 3. In Figure 2(b), the graph is simply cut into three partitions vertically. This straightforward approach leads to the situation where the leftmost and rightmost partitions also become neighbouring partitions due to the long edge $e(1, 4)$. This is undesirable as it would introduce additional data dependencies between the partitions. For a larger and more complex road networks, the edges in the graph may considerably differ in lengths (e.g., expressways versus smaller roads). Simply cutting the



(a) Partitioning with NRGG, starting at initial vertex 1

(b) Vertical stripe spatial partitioning

Figure 2: Comparison of the proposed partitioning algorithm with stripe spatial partitioning, where a simple graph is divided into three partitions (assuming vertices have equal weights).

graph into stripes can therefore lead to a higher number of neighbouring partitions.

4.2 Refinement with Local Search

After the initial partitioning, partitions are refined in order to reduce edge cut (i.e., Obj_3), as well as to alleviate workload imbalance in case there is severe imbalance.

We propose a local search refinement heuristic similar to Karypis and Kumar's boundary refinement heuristic [8], which is a variation of the original heuristic of Kernighan and Lin [10]. A distinct feature of our proposed local search refinement heuristic is to restrict neighbouring partitions. The procedure is described below.

4.2.1 Gains of moving vertices

First, the *internal costs* and *external costs* of the vertices at the boundaries of partitions are calculated using the weights of the connected edges. For a vertex v in partition G_i, its *internal cost* is the total weight of all edges that connect vertex v and its adjacent vertices in the same partition. Let S_v be the set of adjacent vertices of v in the same partition G_i. Then, the interval cost can be calculated as:

$$I_v = \sum_{u \in S_i} w_{e(v,u)} \tag{4}$$

where $w_{e(v,u)}$ is the weight of edge $e(v,u)$.

The *external cost* of vertex v towards partition G_j (assuming G_i and G_j are neighbouring partitions) is the total weight of the edges connecting v and its adjacent vertices in partition G_j. Let $S_v^{(j)}$ be set of adjacent vertices of v in partition G_i. Then, the external cost is calculated as:

$$E_v^{(j)} = \sum_{k \in S_v^{(j)}} w_{e(v,k)} \tag{5}$$

A vertex can have only one internal cost, but multiple external costs if it has adjacent vertices in more than one neighbouring partitions. The external cost indicates the data volume of communication between neighbouring LPs. Therefore, the external costs of vertices should be reduced in the refinement phase.

The *gain* of moving a vertex from one partition to another is calculated according to the costs. It is the reduction of total external costs if a vertex is *moved*. To move a vertex means to change the partition id assigned to it. The gain of moving a vertex v from G_i to G_j is the difference between its external cost towards G_j and its internal cost, that is,

$$gain_v^{i \to j} = E_v^{(j)} - I_v \tag{6}$$

When $E_v^{(j)} > I_v$, $gain_v^{i \to j} > 0$, moving vertex v from G_i to G_j can reduce the total external cost (i.e., the total weight of cut edges). However, if $E_v^{(j)} < I_v$ the gain will be negative. Gains are computed for each potential partition a vertex could be moved to.

4.2.2 Moving rules

After the gains of boundary vertices are computed, the vertices are examined for moving from their current partitions to neighbouring partitions. The order of examination is from the higher gains to the lower gains. A vertex v will be moved from its origin partition G_i to a neighbouring partition G_j in two cases: i) when the external cost can be reduced; and ii) when there is severe weight imbalance that needs to be alleviated. The first case needs to satisfy all the following criteria:

1. $gain_v^{i \to j} > 0$
2. $W_i > W_{min}$ after v is moved, where W_{min} is the minimum weight an LP must contain, that is, the weight of the origin partition is above a low threshold after moving the vertex, for example, $W_{min} = 0.9 \cdot \overline{W}$
3. $W_j < W_{max}$ after v is moved, where W_{max} is the maximum weight an LP can contain, that is, the weight of the target partition is below an upper threshold after moving the vertex, for example, $W_{max} = 1.1 \cdot \overline{W}$
4. $\forall k, l$ such that if $n_{i,k}$=0 and $n_{j,l}$=0 (i, j, k, l being four partition ids, and $n_{i,j}$ the indicator whether i and j are neighbours), then $n_{i,k}$ and $n_{j,l}$ have to remain 0 after v is moved, that is, moving vertex v does not create new neighbouring partitions

The second case needs to satisfy all the following criteria:

1. $W_i > W_{max}$ before the move, that is, the weight of the origin partition has exceeded an upper threshold
2. $W_j < W_i$ after v is moved, that is, the move is only from a partition with more weights to another partition with less weights
3. $w_v > 0$
4. $\forall k, l$ such that if $n_{i,k}$=0 and $n_{j,l}$=0, $n_{i,k}$ and $n_{j,l}$ remain to be 0 if v is moved (in line with the fourth criterion in the first case)

The fourth criterion in both cases is a neighbour-restricting constraint, which is illustrated with an example in Figure 3. G_2 is a neighbouring partition of both G_1 and G_3. Vertices 1 and 2 in G_2 have external costs with both other partitions. However, it is forbidden to move either of them to G_1 or G_3 according to the neighbour-restricting constraint, because G_1 and G_3 are not neighbouring partitions. Moving vertex 1 or 2, however, would make G_1 or G_3 neighbours.

Whenever a vertex is moved, its internal cost and external costs are updated, as well as the costs of its adjacent vertices. A *pass* of refinement is finished when all boundary vertices are examined for moving. At the end of each pass, the boundary vertex set is updated. Then, a new pass begins. This process repeats until there is no vertex movement in a pass or the number of passes has reached a pre-defined maximum value. The complexity of the refinement phase is $O(B \cdot \log_2 B \cdot P)$ where B is the number of boundary vertices and P is the maximum number of passes. The factor $\log_2 B$ is for sorting the vertices according to their gains.

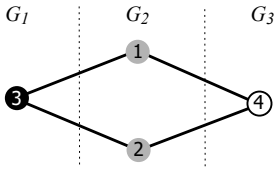


Figure 3: A simple graph in three partitions. Due to the neighbour-restricting constraint, vertices 1 and 2 are not allowed to be moved to G_1 or G_3.

5. EXPERIMENTS

The performance of the proposed partitioning algorithm was investigated in a parallel agent-based traffic simulation, *double-blinded* [24]. The simulation is implemented using C++. The communication between LPs is realised using OpenMPI. Comparison with stripe spatial partitioning, and graph partitioning methods, i.e, METIS [9] and Buffoon [18] is performed. We compare our algorithm, denoted as *NRGG*, to stripe spatial partitioning (denoted as *Stripe*), to METIS partitioning (denoted as *METIS*) and to Buffoon partitioning (denoted as *Buffoon*). Parameter settings of METIS and Buffoon are the default values provided in their software packages. Since the simulation involves stochastic elements, the simulation was run multiple times for each method.

5.1 Simulation Configuration

5.1.1 Data and models

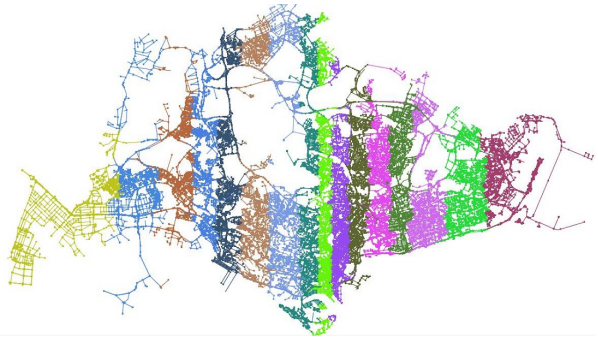
The data used in the experiments is based on real-world data, including the road network and origin-destination matrix of agents. The experiment scenario is set up as follows: The road network is the network of Singapore city that consists of approximately 43,000 nodes and 84,000 links in our representation. The origin-destination matrix of agents is derived from the data of the 2008 Household Interview and Travel Survey (HITS). Agents move according to the intelligent driver car-following model [21] and a simple rule-based lane-changing model. The traffic of 19 hours from 5am to midnight of one day is simulated. The maximum number of agents during the peak traffic hours of the day is approximately 73,000.

5.1.2 Partitioning and dynamic load-balancing settings

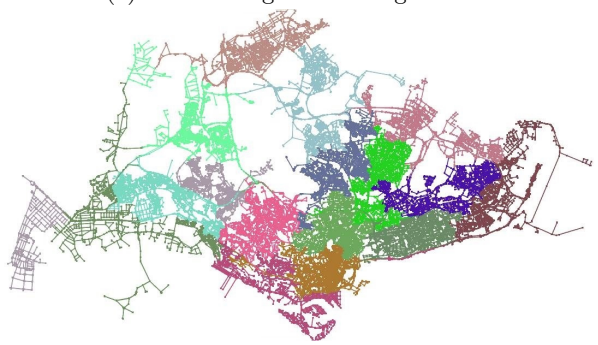
We use run-time dynamic partitioning to achieve dynamic load-balancing. Workload of LPs is examined periodically every 10 minutes of simulation time. When the imbalance exceeds 500 agents (value obtained from our previous work in [25]) the road network is repartitioned by a partitioning algorithm, including recalculation of vertex and edge weights. After that, agents are re-distributed to their new partitions.

To partially solve the problem that the choice of initial vertex for graph-growing affects the partitioning result, we run the proposed partitioning algorithm twice concurrently by two LPs using two different initial vertices. One is the vertex furthest to the east, and the other is the vertex furthest to the west. The partitioning result with less edge cut is chosen as the final result.

In the refinement phase, the workload thresholds for moving vertices, W_{min} and W_{max}, are $0.9 \cdot \overline{W}$ and $1.02 \cdot \overline{W}$, respectively. A maximum of 8 refinement passes are allowed, based on the observation that additional passes did not lead to significant improvement on edge cut.



(a) Partitioning result using NRGG



(b) Partitioning result using METIS

Figure 4: Partitioning of the Singapore road network into 16 partitions.

5.1.3 Hardware

The experiments were run on a cluster consisting of three compute nodes each of which has the following hardware configurations: two $Octacore\ Intel(R)\ Xeon(R)$ CPUs with 2.60 GHz clock frequency (i.e., 16 physical processors), and 192 GB RAM. The compute nodes are connected via 56 Gbps InfiniBand. Each LP in a parallel simulation is bonded to one physical CPU core.

5.2 Results

A visual representation of the difference between our algorithm and METIS is given in Figure 4. The road network of Singapore is divided into 16 partitions. Different colours represent different partitions. Stripe and Buffoon are not shown since they generate similar shapes as NRGG and METIS, respectively.

As expected, the partitions generated from NRGG (Figure 4(a)) are in roughly striped shapes with saw-toothed boundaries, because the vertices are sorted by their longitudes, whereas METIS (and Buffoon) create more irregular shapes (Figure 4(b)).

5.2.1 Workload Imbalance

The first indicator of the quality of partitioning algorithms is the workload imbalance of the simulation. We measure the imbalance with the number of agents assigned to the LPs, since every agent has roughly the same computational workload. The imbalance was captured every 4 seconds of simulation time. The average for the entire simulation is

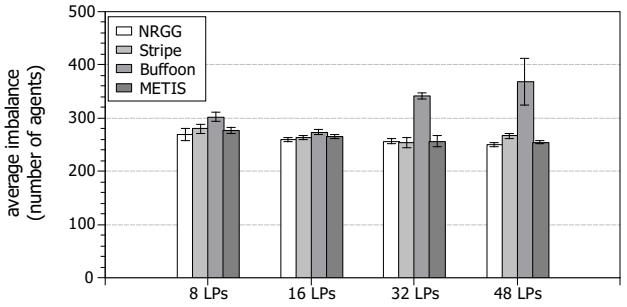


Figure 5: Average load imbalance per update interval in terms of number of agents. Smaller is better.

shown in Figure 5. We observe that all partitioning methods have achieved similar balancing results, except Buffoon that has a higher imbalance with 32 and 48 LPs. The reason is that the workload change of LPs is more severe for partitions generated by Buffoon. The load-balancing operation was also triggered more often using Buffoon as a result of a more dynamic workload, as shown in Table 1.

5.2.2 Synchronisation Messages

The second indicator is the number of messages sent throughout the simulation. The average number of neighbouring partitions per partition and the total number of synchronisation messages are shown in Figure 6.

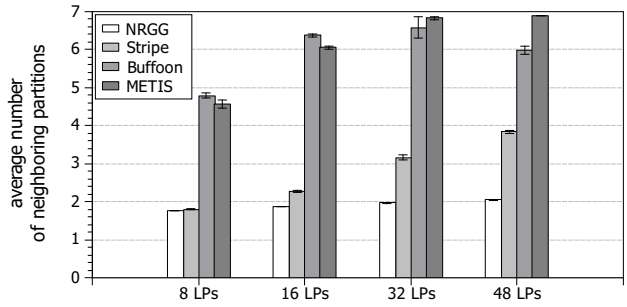
We can observe from Figure 6(a) that the average number of neighbouring partitions increases as the number of LPs increases for all partitioning methods. NRGG has generated the smallest average number of neighbouring partitions. Stripe has a similar result for 8 LPs, but not 16 and 32 LPs. The reason is that when a spatial graph is cut into a large number of partitions using Stripe, long edges tend to span further than two partitions and lead to more neighbouring partitions (see Figure 2(b)). Thus, we conclude that NRGG is more resilient than Stripe when partitions are cut more finely. Buffoon and METIS generate a much larger number of neighbouring partitions, because they do not consider the minimisation of the number of neighbouring partitions.

Consequently, NRGG is able to keep the number of messages the lowest in all the cases above, as shown in Figure 6(b). Using Buffoon and METIS, we observe a considerable higher number of synchronisation messages, resulting from the larger number of neighbouring partitions. Looking at the lookaheads of LPs, we observe that Buffoon and METIS have around 40 percent larger lookahead than NRGG and Stripe. This is an advantage because larger lookahead leads to fewer synchronisation messages. The larger lookahead is a result of fewer boundary links between partitions, which generate less data dependencies. However, this larger lookahead did not compensate the disadvantage of the larger number of neighbouring partitions.

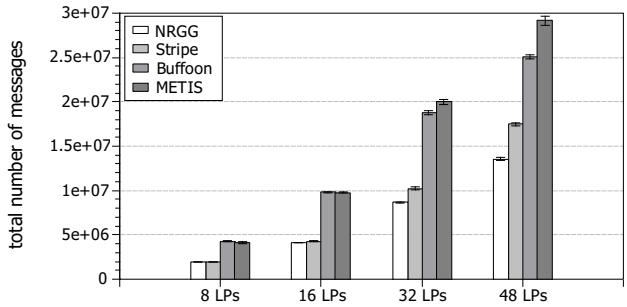
5.2.3 Communication Data Volume

The third indicator for the quality of partitioning algorithms is the communication data volume. It is quantified using the number of migrated agents and the number of shared states. The results are shown in Figure 7. Smaller values mean lower transmission time for synchronisation messages.

It can be observed from Figure 7(a) that NRGG has a



(a) Average number of neighbouring partitions



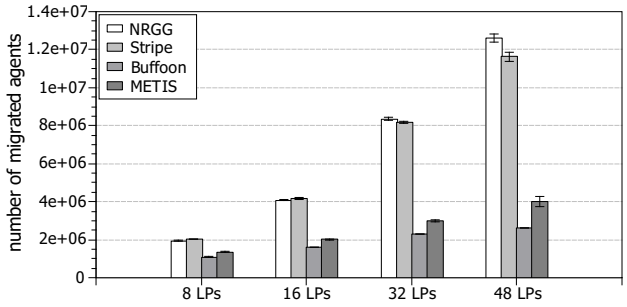
(b) Total number of exchanged messages between LPs

Figure 6: Statistics on synchronisation overhead between LPs. Smaller values are better.

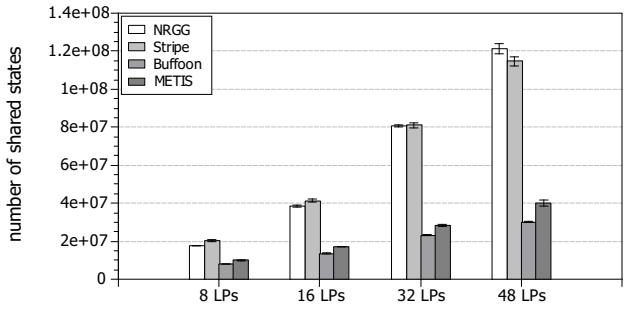
considerably larger number of migrated agents and shared states than METIS and Buffoon. This is because edge cut is not minimised in the initial partitioning phase, and the local search refinement can only reduce edge cut to a certain extent. Buffoon leads to the lowest number of migrated agents and shared states in all cases, which shows that the natural cut heuristic improves the quality of edge cut in partitioning road networks. Stripe performs similar to NRGG. In the cases of 32 and 48 LPs, NRGG is slightly worse than Stripe, which shows that to restrict neighbouring partitions may sacrifice some edge cut quality.

We also investigated the possibility of using multi-level refinement to increase the search space of local search. It can reduce the number of migrated agents and shared states by approximately 10 percent when using 8 LPs. However, for a higher numbers of LPs, no obvious improvement was observed. The reason may be that for a large number of partitions, the search space cannot be increased by the multi-level approach, and the neighbour-restricting constraint limits the search space. Hence, multi-level refinement can be considered for a small number of partitions to reduce communication data volume.

In conclusion, it can be observed that NRGG trades a higher number of migrated agents and shared states to achieve a considerably lower number of neighbouring partitions and synchronisation messages. Whether this improves the overall performance of the parallel simulation compared to other algorithms depends on which indicator plays a more important role in the synchronisation overhead. The execution time of the simulation using the various partitioning methods will be presented later in this section.



(a) Total number of migrated agents



(b) Total number of shared states

Figure 7: Indicators of communication data volume, smaller is better.

5.2.4 *Partitioning Overhead*

The total overhead of partitioning using the partitioning methods are shown in Table 1. Results of multiple runs were recorded and the averages were taken.

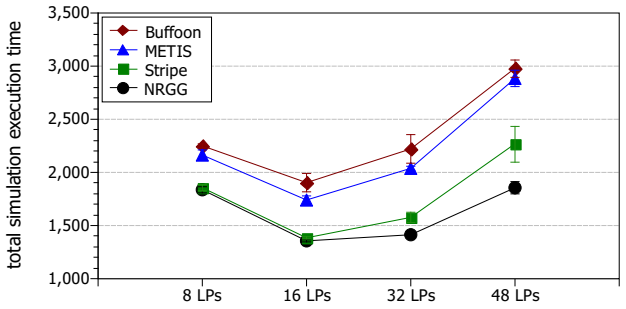
Table 1 shows that the overhead of NRGG partitioning is almost negligible compared to the total execution time of the simulation, which is shown in Figure 8a. The overhead of NRGG is less than 0.5 percent for all LP configurations. This results from its low complexity. Low overhead also applies to Stripe and METIS. Stripe takes the least time since only one traverse of the vertices is needed for partitioning. The overhead of Buffoon is higher than other three methods, however, Buffoon partitioning generated the lowest edge cut. The large overhead is a result of the more complex coarsening and partitioning algorithms in Buffoon.

Table 1: Dynamic load-balancing count and total overheads of the partitioning algorithms in dynamic load-balancing.

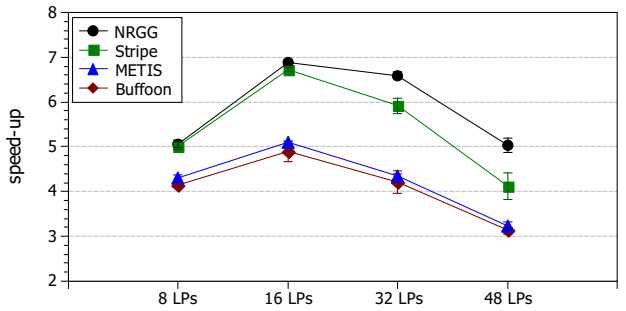
	8 LPs	**16 LPs**	**32 LPs**	**48 LPs**
NRGG count	48±2	41±1	30±2	25±2
Time (s)	4.42±0.16	4.94±0.18	5.00±0.27	4.02±0.02
Stripe count	47±1	40±2	31±2	28±2
Time (s)	1.20±0.07	1.33±0.04	1.23±0.07	1.20±0.07
Buffoon count	53±3	42±2	43±5	62±3
Time (s)	91.9±2.38	94.3±3.48	185.1±11.0	421.4±25.0
METIS count	47±2	41±1	31±3	27±1
Time (s)	4.57±0.62	6.25±0.23	4.21±0.54	3.9±0.2

5.2.5 *Overall Execution Time of the Simulation*

The overall execution time of the parallel simulation is shown in Figure 8a. The execution time of the sequential simulation is around 9,000 seconds. The speed-up of parallel simulation relative the sequential simulation is shown in Figure 8b.



(a) Execution time of the simulation using various partitioning methods (lower is better)



(b) Speed-up of the simulation using various partitioning methods relative to the sequential simulation

Figure 8: Overall execution time and speed-up of the simulation using various partitioning methods.

Figures 8a and 8b show that the simulation using NRGG always provides the lowest execution time and thus the best speed-up. With 8 and 16 LPs, Stripe has similar performance as NRGG. This is because Stripe is able to generate comparable neighbouring partitions as NRGG with such a small number of LPs. However, with 32 and 48 LPs, the performance of Stripe degrades faster due to the increasing number of neighbouring partitions generated by Stripe (see Figure 6(b)). Thus, NRGG has much better resilience than Stripe based on our experiment.

In our simulated road network, Buffoon had the longest execution time. Taking 16 LPs as an example, it spends approximately 40 percent more time than NRGG and 9 percent more than METIS. According to the figures of load imbalance and synchronisation overhead indicators, Buffoon should have similar performance as METIS. However, longer execution time is incurred because of its much higher overhead for dynamic partitioning (see Table 1).

5.2.6 *Discussion*

The performance of a parallel simulation depends on many factors: load balance, number of synchronisation messages, and data volume of communication. Compared to the existing methods, we have shown that for conventional microscopic agent-based traffic simulation, NRGG improves the

overall performance of the simulation by considering the objective of reducing the number of neighbouring partitions. This demonstrates that general graph partition algorithms may not work best in some parallel applications.

NRGG emphasizes more on minimising the number of neighbouring partitions than minimising edge cut. However, there is a trade-off between the two objectives. In a setting where communication data volume plays a more critical role in the synchronisation overhead than the number of messages, METIS and Buffoon may provide better performance. One example for such a setting is that the increased communication data volume using NRGG may activate the rendezvous protocol of MPI, thereby tremendously increasing the synchronisation overhead. In addition, depending on the synchronisation protocol, the number of synchronisation messages may not be heavily affected by the number of neighbouring partitions. In these cases, NRGG may not achieve the best performance. For our microscopic agent-based traffic simulation, however, NRGG has been shown to outperform the existing approaches.

6. CONCLUSION AND FUTURE WORK

We have proposed Neighbour-restricting Graph-Growing (NRGG), a graph partitioning algorithm for reducing the total execution time of parallel agent-based road traffic simulation. In addition to minimising the imbalance of partitions and minimising edge cut, NRGG particularity focuses on reducing the number of neighbouring partitions. It makes use of a graph-growing algorithm, followed by a modified KL local search refinement algorithm. The essence of the proposed algorithm is that it tries to constrain the number of neighbouring partitions of each partition in both the graph-growing phase and the refinement phase, guided by both connectivity information and the geographical information of the road network. Our approach is able to considerably reduce the number of synchronisation messages. Experiments using our parallel agent-based traffic simulation have shown that NRGG outperforms stripe spatial partitioning and both the popular graph partitioning methods, METIS and Buffoon.

Future work can be conducted from the following aspects: i) for different settings of traffic simulation and for other types of parallel applications, the trade-off between the three objectives can be tuned for optimised performance, depending on their respective significance; and ii) for dynamic load-balancing, neighbour-restricting diffusion can be investigated, where partitions incrementally change their shapes instead of repartitioning.

7. ACKNOWLEDGEMENTS

This work was financially supported by the Singapore National Research Foundation under its Campus for Research Excellence And Technological Enterprise (CREATE) programme.

8. REFERENCES

[1] M. J. Berger and S. H. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Transactions on Computers*, C-36(5):570–580, May 1987.

[2] A. Boukerche and S. K. Das. Dynamic Load Balancing Strategies for Conservative Parallel Simulations. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pages 20–28, Austria, June 10 - 13, 1997.

[3] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph Partitioning with Natural Cuts. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011)*, pages 1135–1146, Anchorage, AK, USA, May 2011.

[4] R. M. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley Interscience, December 1999.

[5] P. G. Gipps. A behavioural car-following model for computer simulation. *Transportation Research Part B: Methodological*, 15(2):105–111, April 1981.

[6] P. G. Gipps. A model for the structure of lane-changing decisions. *Transportation Research Part B: Methodological*, 20(5):403–414, October 1986.

[7] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519–1534, Nov. 2000.

[8] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, Jan. 1998.

[9] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, January 1998.

[10] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(1):291–307, February 1970.

[11] A. Kesting, M. Treiber, and D. Helbing. General Lane-Changing model MOBIL for Car-Following Models. *Transportation Research Record: Journal of the Transportation Research Board*, No. 1999(1):86–94, Jan. 2007.

[12] R. Klefstad and Y. Zhang. A distributed, scalable, and synchronized framework for large-scale microscopic traffic simulation. In *Proceedings of the 2005 IEEE Intelligent Transportation Systems Conference (ITSC 2005)*, pages 813–818, Vienna, Austria, September 2005. IEEE.

[13] D.-H. Lee. A Framework for Parallel traffic simulation using multiple instancing of a simulation program. *Journal of Intelligent Transportation Systems*, 7(3):279–294, July 2002.

[14] H. Mizuta, Y. Yamagata, and H. Seya. Large-scale traffic simualtion for Low-Carbon City. In *Proceedings of the 2012 Winter Simulation Conference (WSC'2012)*, pages 1–12, Berlin, Germany, December 2012.

[15] K. Nagel and M. Rickert. Parallel implementation of the TRANSIMS micro-simulation. *Parallel Computing*, 27(12):1611–1639, November 2001.

[16] A. Pothen, H. D. Simon, and K.-P. P. Liu. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM Journal on Matrix Analysis and Applications*, 11(2):430–452, 1990.

[17] T. Potuzak. Distributed/Parallel Genetic Algorithm for Road Traffic Network Division Using a Hybrid Island Model/Step Parallelization Approach. In *Proceedings of the 2016 IEEE/ACM 20th International Symposium on Distributed Simulation*

and Real Time Applications (DS-RT), pages 170–177, London, UK, September 21-23 2016.

[18] P. Sanders and C. Schulz. Distributed Evolutionary Graph Partitioning. In *Meeting on Algorithm Engineering & Experiments (ALENEX '12)*, pages 16–29, Kyoto, Japan, jan 2012.

[19] K. Schloegel, G. Karypis, and V. Kumar. Wavefront diffusion and LMSR: Algorithms for dynamic repartitioning of adaptive meshes. *IEEE Transactions on Parallel and Distributed Systems*, 12(5):451–466, 2001.

[20] T. Suzumura and H. Kanezashi. Highly Scalable X10-Based Agent Simulation Platform and Its Application to Large-Scale Traffic Simulation. In *Proceedings of the 2012 IEEE/ACM 16th International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2012)*, pages 243–250, Dublin, Ireland, October 2012.

[21] M. Treiber, A. Hennecke, and D. Helbing. Congested traffic states in empirical observations and microscopic simulations. *Physical Review E*, 62(2):1805–1824, February 2000.

[22] S. Uppoor and M. Fiore. Large-scale Urban Vehicular Mobility for Networking Research. In *Proceedings of the 3rd IEEE Vehicular Networking Conference (VNC 2011)*, pages 62–69, Amsterdam, Netherlands, November 2011. IEEE.

[23] D. Wei, F. Chen, and X. Sun. An improved road network partition algorithm for parallel microscopic traffic simulation. In *Proceedings of the 2010 International Conference on Mechanic Automation and Control Engineering (MACE)*, pages 2777 – 2782, Wuhan, China, June 2010.

[24] Y. Xu, H. Aydt, and M. Lees. SEMSim: A Distributed Architecture for Multi-scale Traffic Simulation. In *Proceedings of the 26th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS'12)*, pages 178–180, Zhangjiajie, China, July 15-19 2012. IEEE.

[25] Y. Xu, W. Cai, H. Aydt, and M. Lees. Efficient graph-based dynamic load-balancing for parallel large-scale agent-based traffic simulation. In *Proceedings of the 2014 Winter Simulation Conference (WSC'14)*, pages 3483–3494, Savannah, GA, USA, December 2014. IEEE.

[26] Y. Xu, W. Cai, H. Aydt, M. Lees, and D. Zehe. An Asynchronous Synchronization Strategy for Parallel Large-scale Agent-based Traffic Simulations. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS'15)*, pages 259–269. ACM, June 2015.