

Project Report (Practical Course on HPC)

Simulating Traffic Jams in City Networks

Salvador Costilla Caballero

MatrNr: 21775759

Supervisor: Aasish Kumar Sharma

Georg-August-Universität Göttingen
Institute of Computer Science

September 29, 2026

Abstract

Traffic jams can emerge without an external cause, from drivers who brake and accelerate imperfectly. The Nagel-Schreckenberg (NaSch) cellular automaton reproduces this with a few local rules, but simulating a whole city with millions of road cells is expensive, especially in an interpreted language. This project implements the NaSch model on directed road graphs (a ring, random cities, networks traced from images and real OpenStreetMap (OSM) maps) as a Python package with a serial backend and an Message Passing Interface (MPI) backend. The MPI backend replicates the road graph on every rank, distributes the road cells in slabs along the x-axis, and exchanges entry halos, migrating vehicles and arbitration verdicts with neighbouring ranks; a global vehicle count checks its correctness. The only strong-scaling campaign with data (1 to 32 ranks, shared and unpinned nodes, two runs per point) gave speedups at 32 ranks of 32.9 (random city), 25.1 (Leipzig) and 28.9 (ring). The superlinear values contradict the load-imbalance bound and are treated as a measurement artefact. A corrected campaign was prepared but did not run before the deadline, so a model based on the measured data projects the efficiencies at 32 ranks instead: about 0.62 (random city), 0.49–0.57 (Leipzig) and 0.86–0.93 (ring). The serial runtime grows approximately linearly with the number of cells.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☐ During brainstorming
- ☐ When creating the outline
- ☒ To write individual passages, altogether to the extent of 50% of the entire text
- ☐ For the development of software source texts
- ☒ For optimizing or restructuring software source texts
- ☐ For proofreading or optimizing
- ☒ Further, namely: - Claude Code, GitHub Copilot

Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables	iv
List of Figures	iv
List of Listings	iv
List of Abbreviations	v
1 Introduction	1
2 Methodology	1
2.1 Problem statement and requirements	1
2.2 The Nagel-Schreckenberg model on a road graph	2
2.3 Sequential design	2
2.4 Options for parallelisation	3
2.5 Parallel design	3
2.6 Expected performance	4
2.7 Validation and benchmarking protocol	5
3 Implementation	6
4 Performance analysis	7
4.1 Configurations	7
4.2 Static partition analysis	7
4.3 Serial growth	8
4.4 Strong scaling: measured results	8
4.5 Validity of the measured results	9
4.6 Projected results after the fix (model, not measured)	10
4.7 Weak scaling	11
4.8 Performance analysis tools	11
4.9 Discussion of the results	11
5 Challenges and discussion	12
6 Conclusion	12
References	14
A Work sharing	A1
B Code samples	A1
C Code repository	A3
D Additional figures and tables	A3

List of Tables

1	Problem definitions of the measured strong-scaling campaign	7
2	Static properties of the x-slab partition	8
3	Measured strong-scaling results of the old campaign	10
4	Projected strong scaling after the fix (model, not measured)	A6

List of Figures

1	X-slab partition of a random city on four ranks	3
2	Communication rounds of one time step	4
3	Space-time diagram and fundamental diagrams of both backends	5
4	Measured strong scaling of the old campaign	9
5	Projected efficiency after the fix (model, not measured)	11
6	Measured runtime of the old campaign	A4
7	Measured speedup of the old campaign	A4
8	Measured Karp-Flatt metric of the old campaign	A4
9	Projected runtime after the fix (model, not measured)	A5
10	Serial runtime growth with problem size	A5
11	Normalised serial cell-update rate	A5

List of Listings

1	Timed region with barriers and maximum over ranks (<code>run()</code> , abridged).	6
2	Edge ownership by x-slabs of the source node (<code>runners/mpi.py</code>).	A1
3	Round 1, entry halos (<code>ParallelNaSch.step</code> , abridged).	A1
4	Owner-side arbitration of entry requests (abridged).	A2
5	Phase 2 of <code>ParallelNaSch.step</code> : local moves and entry requests.	A2
6	Applying the verdicts after round 3.	A2
7	Submission and launch of one job of the corrected strong-scaling campaign (abridged; submitted, but not run before the deadline).	A3

List of Abbreviations

CLI Command-Line Interface

GWDG Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

HPC High-Performance Computing

MPI Message Passing Interface

NaSch Nagel-Schreckenberg

OSM OpenStreetMap

RNG Random Number Generator

1 Introduction

Problem and motivation. Much of the congestion in cities is not caused by accidents or bottlenecks but emerges from traffic itself: small fluctuations in driving grow into stop-and-go waves that travel backwards against the traffic. The NaSch cellular automaton [NS92] is the standard minimal model of this effect. It divides roads into cells and updates every vehicle with four local rules; one of them, random braking, is enough to produce spontaneous jams. A single ring road is cheap to simulate, but a city has millions of cells and hundreds of thousands of vehicles. Large simulators such as TRANSIMS [NR01] and SUMO [Lop+18] use compiled code, and TRANSIMS distributes the road graph over many processes. This project asks how far such a decomposition can speed up a Python implementation of the NaSch model on general road graphs on the High-Performance Computing (HPC) cluster of the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG).

Objectives. The project has two goals: (1) a Python package with a Command-Line Interface (CLI) that runs the NaSch model on four kinds of road networks, with a serial and an MPI backend (requirements in section 2.1); and (2) a measurement of the strong scaling of the MPI backend from 1 to 32 ranks and of the growth of the serial runtime with problem size.

Outcomes. The package is complete. The serial model reproduces backward-moving jams and the fundamental diagram, and the MPI backend conserves vehicles and agrees statistically with the serial backend (section 2.7). The performance goal was reached only in part: the only scaling campaign with data ran on shared, unpinned nodes and shows superlinear speedups, which are treated as a measurement artefact (section 4.5). A corrected campaign was submitted but did not run before the deadline, so section 4.6 gives a model-based projection, kept separate from the measurements.

Organisation. Section 2 describes the model, the designs and the experiments, section 3 the implementation, section 4 the performance results, section 5 the challenges, and section 6 concludes.

2 Methodology

2.1 Problem statement and requirements

The road network is a directed graph: nodes are intersections with planar coordinates, edges are roads with a length and a speed limit. The functional requirements are:

- F1 Road networks from four sources: ring, random, image and real map (OSM via OSMnx [Boe17]), all deterministic for a given seed.
- F2 NaSch dynamics on every road, including crossing intersections and resolving conflicts when several vehicles compete for the same entry cell.

- F3 Two vehicle modes: wandering vehicles that take a random exit at every intersection, and routed vehicles that follow a shortest path and leave on arrival.
- F4 A serial backend and an MPI backend, switchable from the CLI.
- F5 Outputs: animations, congestion heatmaps, space-time diagrams, the fundamental diagram, and timing lines for the scaling study.

The parallel backend must also conserve vehicles exactly, reproduce the statistics of the serial model, and scale with the number of ranks for a fixed problem size.

2.2 The Nagel-Schreckenberg model on a road graph

Every edge e of length ℓ_e is divided into $n_e = \max(1, \text{round}(\ell_e/7.5 \text{ m}))$ cells, each empty or holding one vehicle. One time step is one second, so one cell per step equals 27 km/h, and each edge gets its own maximum velocity from its speed limit s_e in km/h:

$$v_{\max}(e) = \min(5, \max(1, \text{round}(s_e/27))). \quad (1)$$

In every step, each vehicle with velocity v on edge e applies the four NaSch rules [NS92]:

$$1. \text{ acceleration: } v \leftarrow \min(v + 1, v_{\max}(e)) \quad (2)$$

$$2. \text{ braking: } v \leftarrow \min(v, g) \quad (3)$$

$$3. \text{ randomisation: } v \leftarrow \max(v - 1, 0) \quad \text{with probability } p_{\text{slow}} \quad (4)$$

$$4. \text{ movement: } x \leftarrow x + v \quad (5)$$

The gap g is the number of empty cells ahead, capped at v . If the scan reaches the end of the edge, it continues into the next edge: the next route element for routed vehicles, a random out-edge for wandering ones. The default p_{slow} is 0.3; rule 3 is the source of spontaneous jams. The observables are the density $\rho = N/L$ (N vehicles on L cells) and the flow $q = \rho \bar{v}$ with the time-averaged mean velocity $\bar{v}$; the relation $q(\rho)$ is the fundamental diagram.

On a graph, two vehicles from different roads can target the same entry cell. The update therefore has two phases: phase 1 computes all velocities from the current occupancy; phase 2 moves the vehicles in a random order, reshuffled every step, and checks a move into the next edge against the live occupancy of that edge. A vehicle that finds the entry cells taken stops at the last cell of its edge. Within an edge, the gaps of phase 1 already exclude collisions.

2.3 Sequential design

The package `congestion` has three independent layers. `networks/` contains `RoadGraph`, a typed wrapper around a NetworkX [HSS08] `DiGraph`, and the generators `from_ring`, `from_random` (Delaunay triangulation of random points), `from_image` (posterise, blur, skeletonise, trace) and `from_map` (OSMnx). `models/nasch.py` contains `NaSchSimulator`, which stores one NumPy array per edge (vehicle id, or -1 for empty) and implements the two-phase update. `runners/` contains the serial loop and the distributed engine `ParallelNaSch`. The CLI builds a network and passes it to the runner chosen by `-backend serial|mpi`; the subcommands `sweep` and `spacetime` produce the fundamental and space-time diagrams. The loop iterates over vehicles, so at fixed density the cost of a step is proportional to the number of cells.

2.4 Options for parallelisation

Shared-memory threading is blocked by the CPython global interpreter lock unless the kernel is compiled. *Independent simulations* (e.g. one density per process) are trivially parallel but do not speed up one large network. *Distributed-memory domain decomposition* with MPI [Mes21], where every rank owns part of the roads and exchanges only boundary data, is the approach of large traffic simulators [NR01] and the one chosen here, using mpi4py [DF21] with one rank per core.

2.5 Parallel design

Replicated topology, distributed state. Every rank runs the same deterministic generator and holds the full road graph, which is static and small. Only the cell arrays of its own edges are allocated. Vehicles are placed by the same seeded code on every rank, and each rank keeps those on its own edges.

Ownership. The x-coordinates of the source nodes of all edges are sorted, and $p - 1$ cuts are taken at the quantiles r/p . An edge belongs to the slab of its source node. Every rank computes this identically, so no partition data is sent and every rank knows the owner of every edge. The slabs have equal numbers of edges, but not necessarily equal numbers of cells (fig. 1).

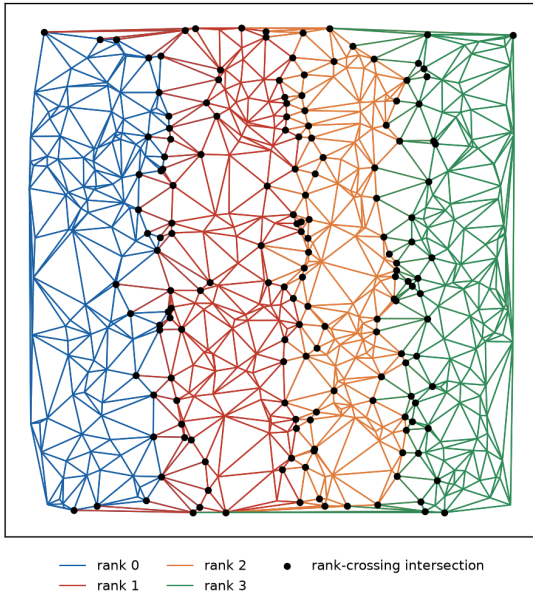


Figure 1: X-slab partition of a 500-node random city on four ranks. Black dots mark intersections where a vehicle can move onto an edge of another rank.

Communication pattern. Each rank knows from the graph which ranks its vehicles can move to (downstream) and which ranks can send vehicles to it (upstream). Each step has three rounds of point-to-point messages (fig. 2):

1. *Entry halos.* For every edge with a remote predecessor, the owner sends its first $v_{\max} + 1 = 6$ cells upstream, so gaps can be computed across the rank boundary.
2. *Migration requests.* Every vehicle that crosses into another edge becomes a request (id, desired entry cell, remaining distance, route) sent to the owner of that edge; local edges are handled the same way.

3. *Verdicts.* The owner shuffles the requests for each edge with a generator seeded by $(\text{seed}, \text{step}, \text{edge})$ and accepts them while entry cells are free, as in the serial live check. It creates accepted vehicles and replies; the source rank removes migrated vehicles and stops rejected ones at the end of their edge.

Every rank sends one message to every partner in every round, even if empty, so all receives can block without deadlock.

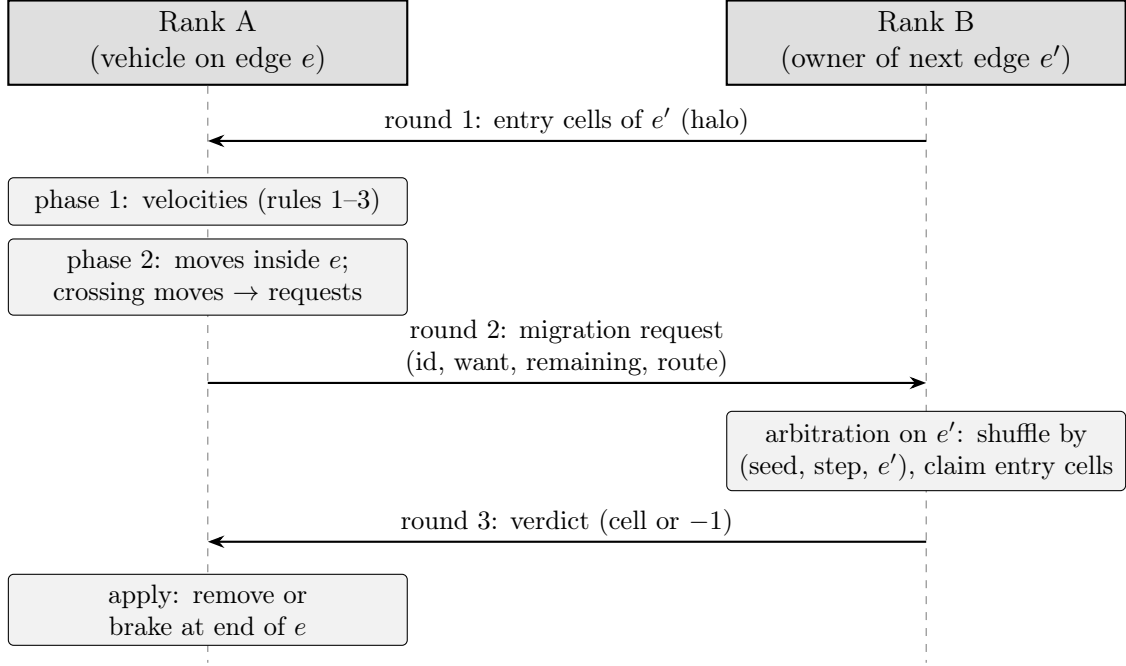


Figure 2: The three communication rounds of one time step between an upstream rank A and the owner B of the next edge. Every pair of neighbouring ranks exchanges messages in both roles.

Randomness. Random braking and exit choices use one Random Number Generator (RNG) per rank, seeded with the run seed and the rank, so trajectories depend on the number of ranks and differ from the serial backend.

Conservation invariant. A single `allreduce` sums the vehicle counts, velocities and arrivals of all ranks, and the run asserts that vehicles in the network plus arrived vehicles equal the initial number. A lost or duplicated migration breaks this immediately. `-check-every K` sets the interval (default: every step); the measured campaign checked every step, the corrected jobs only at the last step.

2.6 Expected performance

Let W be the work of one step (proportional to the number of vehicles) and W_r the share of rank r . The slowest rank determines the step time, so

$$T(p) \approx \max_r W_r + T_{\text{comm}}(p), \quad E(p) = \frac{T(1)}{pT(p)} \leq \frac{W/p}{\max_r W_r} = \frac{1}{I(p)}, \quad (6)$$

where $I(p)$ is the load imbalance. Vehicles are placed evenly over all cells, so $I(p)$ equals the cell imbalance of the partition, which can be computed without running the simulation (section 4.2). Communication costs a latency per partner and a volume proportional to the crossing edges and migrating vehicles. Nothing in the loop is executed redundantly, so the serial fraction in the sense of Amdahl’s law [Amd67] is small; the expected limits are load imbalance and message overhead once the work per rank becomes small.

2.7 Validation and benchmarking protocol

Serial model. On a ring of 150 m segments ($v_{\max} = 5$, $p_{\text{slow}} = 0.3$), jams form spontaneously and drift backwards at density 0.25, and the flow peaks at a density between 0.10 and 0.15 and then decreases almost linearly (fig. 3). Unit tests check that the ring conserves vehicles, that no two vehicles ever share a cell, and that all routed vehicles arrive.

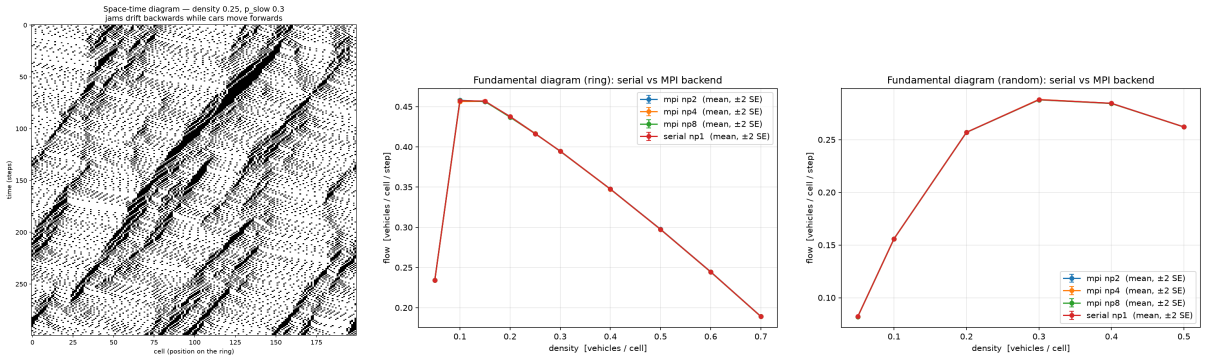


Figure 3: Left: space-time diagram at density 0.25 on the ring (black: occupied cell; time runs downwards). Middle and right: fundamental diagrams of the serial backend and of the MPI backend with 2, 4 and 8 ranks on the ring (2000 cells) and on a random city (150 nodes, 11 936 cells); mean over 40 seeds, ± 2 standard errors.

Parallel backend. Since the random streams differ, the MPI backend is validated statistically: its fundamental diagram must agree with the serial one within the run-to-run spread. `validate_backends.py` compares the serial backend with 2, 4 and 8 ranks, 40 seeds per density, on two networks:

- a ring of 2000 cells (ten densities, 500 warm-up and 1000 measured steps): every node has one incoming road, so halos and migration are tested but no request is ever rejected;
- a random city with 150 nodes (six densities, 300 warm-up and 500 measured steps): up to ten roads enter an intersection, and one four-rank run at density 0.3 rejected 11 264 requests, so the arbitration is tested.

The largest normalised difference of the mean flows is $|z| = 2.15$ on the ring and 2.93 on the random city (at density 0.05, where the absolute difference is below 10^{-4}). The sums $\sum z^2$ are 17.1, 5.8 and 6.9 on the ring (10 degrees of freedom) and 10.3, 5.5 and 7.4 on the random city (6 degrees of freedom) for 2, 4 and 8 ranks, consistent with agreement. Ten seeds were not enough (a first ring comparison gave $|z| \approx 4$). These runs are untimed and do not cover routed vehicles.

Benchmarking protocol. With $S(p) = T(1)/T(p)$, $E(p) = S(p)/p$ and the Karp-Flatt metric [KF90] $e(p) = (1/S(p) - 1/p)/(1 - 1/p)$, measurements must follow these rules:

- the baseline $T(1)$ is the MPI backend on one rank, never the serial backend;
- the timer covers only the stepping loop, with a barrier right before start and stop, and the reported time is the maximum over ranks;
- the problem is fixed per family and only the rank count varies (1 to 32); halo cells are read-only copies, so no cell is updated twice;
- every point is the median of at least three runs, with minimum and maximum;
- jobs run on exclusive nodes with a fixed number of tasks per node, pinned ranks and a fixed CPU frequency.

The log parser `scaling_logs.py` enforces these rules before plotting. A superlinear speedup or a negative $e(p)$ is treated as a measurement error, not a cache effect, until these rules have been checked. The measured campaign predates the protocol (section 4.1).

3 Implementation

The package uses Python ≥ 3.12 with NumPy, NetworkX, SciPy, scikit-image, OSMnx, Matplotlib and mpi4py. The MPI backend (`runners/mpi.py`, 490 lines) shares only the `Vehicle` class and the empty-cell constant with the serial model. Its key parts are listed in appendix B: ownership (listing 2), the halo round (listing 3), arbitration (listing 4), phase 2 and applying the verdicts.

All three rounds use the same pattern: one `isend` per partner, one blocking `recv` per partner in the other direction, then `wait`. The lower-case mpi4py calls pickle dictionaries keyed by edge, and tags 1–3 separate the rounds. In the arbitration, local and remote requests enter the same list, so a rank does not favour its own vehicles.

Listing 1 shows the timed region after the benchmark fix (the measured campaign used an earlier version, section 4.5). Each rank also sums its computation and communication time inside `step()`; both are reported as the maximum over ranks. Waiting in a blocking receive counts as communication, so load imbalance shows up there.

```

1 comm.Barrier()
2 t0 = MPI.Wtime()
3 for s in range(cfg.steps):
4     ...                                # sim.step(), conservation assert
5 comm.Barrier()
6 t_local = MPI.Wtime() - t0
7 elapsed = comm.allreduce(t_local, MPI.MAX) # reported wall time
8 t_min = comm.allreduce(t_local, MPI.MIN)

```

Listing 1: Timed region with barriers and maximum over ranks (`run()`, abridged).

All functional requirements are implemented: F1 in `networks/generators.py`, F2 in `models/nasch.py`, F3 in both backends, F4 in `runners/` and F5 in `viz/`. The `pytest` suite (four tests) covers the serial model; the MPI backend is tested by the conservation assertion in the job scripts and by `validate_backends.py`.

4 Performance analysis

4.1 Configurations

Measured campaign. All scaling numbers come from the first strong-scaling campaign, run in September before the benchmark fix (section 5). It violates the protocol in several ways: the jobs used `sbatch -ntasks=p` on the shared `medium` partition with plain `mpirun`, so nodes were not exclusive, ranks were not pinned and the frequency was not fixed; the metrics `allreduce` ran every step and rank 0 printed progress inside the timed loop; and only two runs (problem seeds 1 and 2) were made per point, so every point is the median, i.e. the mean, of two. Each run did 200 steps for $p \in \{1, 2, 4, 8, 16, 32\}$ with wandering vehicles at density 0.10 (table 1). For the random city the seed also changes the graph. On the Leipzig map vehicles leave at nodes without an outgoing road; 879 to 943 of 103 732 left within 200 steps, so the total work differs by less than 0.1 % between rank counts (for the other families it is identical). Logs with seed 0 in the same directory are excluded (section 5).

Table 1: Problem definitions of the measured strong-scaling campaign (`results/parallel_scaling/`). Counts from the log headers; for the random city for seed 1 / seed 2.

Family	Network flags	Nodes	Cells	Vehicles	$v_{\max}$
random	<code>-network random -n 5000 -size 50000</code> <code>-density 0.10</code>	5 000	3 374 460 3 380 276	337 446 338 028	2
leipzig	<code>-network map -lat 51.3397 -lon 12.3731</code> <code>-radius 20000 -density 0.10</code>	19 602	1 037 317	103 732	1–5
ring	<code>-network ring -segments 320000 -seg-len 150</code> <code>-density 0.10</code>	320 000	6 400 000	640 000	5

Corrected campaign (no data). The corrected campaign (`jobs/submit_scaling_all.sh`, listing 7) follows the protocol: one exclusive `scc-cpu` node per job with `-ntasks-per-node=p -cpu-freq=High`, `srun -cpu-bind=cores`, three repetitions, `-check-every 200` and `-quiet`, on the same random city, a 10 km Leipzig map and the ring at density 0.25. It was submitted but did not start before the deadline because no whole node became free. The old logs have no provenance header, so their CPU model and frequency are unknown. Both campaigns use GCC 14.2.0, Open MPI 5.0.6 and uv 0.10.7.

4.2 Static partition analysis

Table 2 lists the imbalance and communication structure of the partition, computed from the ownership code without running the simulation (`report_src/scripts/partition_stats.py`). It gives the bound of eq. (6).

The ring lies on a circle, so every cut crosses it twice and the slabs are perfectly balanced. For the random city the slabs hold equal numbers of edges but not of cells: at 32 ranks the two outer slabs carry 1.42 and 1.52 times the mean, the inner ones 0.83 to 1.13. The outer slabs contain the long edges along the convex hull of the triangulation, and a slab is only about two road lengths wide (854 m), so these edges are not averaged

Table 2: Static properties of the x-slab partition. Imbalance: maximum over mean per rank. Crossings: pairs of consecutive edges $(u, v) \rightarrow (v, w)$ with different owners. Partners: maximum number of ranks a rank exchanges messages with. The ring has imbalance 1.000, $2(p - 1)$ crossings and at most 2 partners for all p .

Ranks	random (seed-0 graph)				leipzig (20 km)		
	Edge imb.	Cell imb.	Crossings	Partners	Cell imb.	Crossings	Partners
2	1.000	1.002	2 068	1	1.027	663	1
4	1.000	1.042	6 151	3	1.129	1 324	2
8	1.001	1.099	14 613	5	1.246	2 910	3
16	1.002	1.241	31 610	7	1.298	5 863	5
32	1.005	1.516	63 423	10	1.436	11 733	9

out; long edges also connect distant slabs, giving up to 10 partners. The seed-1 and seed-2 graphs of the measured campaign reach 1.507 and 1.430, which limits the efficiency of the random city at 32 ranks to about 0.66–0.70 before any communication. Leipzig has a similar imbalance (1.436, bound 0.70) with far fewer crossings.

4.3 Serial growth

The serial backend was run once per size (August, commit `c180866`, density 0.10, 100 steps, shared unpinned nodes) while one dimension grew per family: random cities with 500 to 8000 nodes (1.09 to 68.6 million cells), Leipzig with radius 5 to 40 km (0.19 to 3.30 million cells) and the ring with 80 000 to 1 280 000 segments (1.6 to 25.6 million cells). Since each vehicle is visited once per step, linear growth is expected.

The runtime is approximately linear in the number of cells (figs. 10 and 11 in appendix D), at 2.7–3.5 million cell updates per second for the random city, 2.3–3.0 for Leipzig and 1.9–2.5 for the ring. The largest sizes lose 15–20 % of throughput (random city 3.33 at 1.09 vs. 2.77 at 68.6 million cells), which may be a cache effect, but single runs on shared nodes cannot separate it from node load, and the Leipzig series is not monotonic. The result is indicative only; a rerun with three repetitions on exclusive nodes did not produce data in time.

4.4 Strong scaling: measured results

Figure 4 shows the efficiency of the measured campaign and table 3 all values. Runtime, speedup and Karp-Flatt plots are in appendix D.

Observations.

- *Random city*: superlinear from 4 ranks on, up to $E = 1.53$ at 8 ranks, and $S(32) = 32.9$; $e(p)$ is negative for every p .
- *Leipzig*: $E > 1$ at 2 and 4 ranks; from 8 ranks on the efficiency falls to 0.79 at 32 ranks ($S = 25.1$), and the growing positive $e(p)$ points to an overhead that increases with p rather than a constant serial fraction.
- *Ring*: E between 0.90 and 1.03 ($S(32) = 28.9$), with a large spread between the two runs from 8 ranks on.

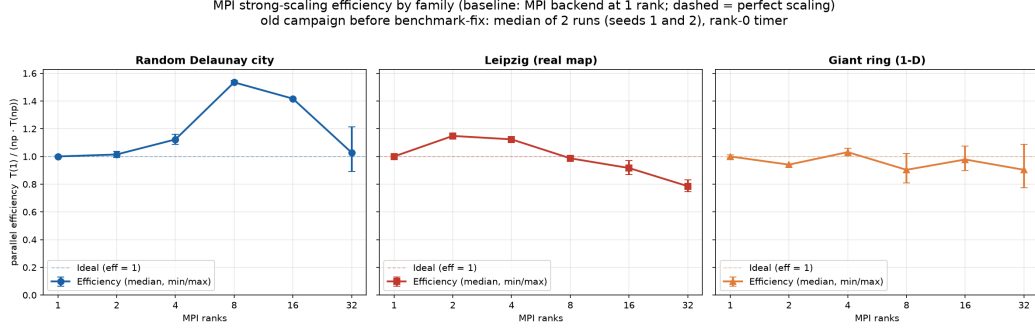


Figure 4: Measured parallel efficiency of the old campaign (median of 2 runs, bars: minimum and maximum; baseline: MPI backend on one rank). Values above 1 are suspected artefacts (section 4.5).

Time split. Both columns are maxima over ranks and do not add up to T . At one rank the communication column sends no messages; it contains the grouping of requests by owner (large on the ring, where about 23 % of the vehicles change segment per step) and the per-step `allreduce`. At 32 ranks, communication is about 64 % of T for the random city, 70 % for Leipzig and 42 % for the ring, which has at most two partners. Since waiting for a slower neighbour counts as communication, this share includes the load imbalance.

4.5 Validity of the measured results

Following the protocol, the superlinear values are treated as measurement errors. Four observations support this:

1. The random city's $E(32) = 1.03$ exceeds its static bound of 0.66–0.70.
2. The excess is in the computation time, which each rank measures locally and which does not depend on barriers: $C(8) = 29.3$ s is 1.7 times faster than $C(1)/8$, although the busiest rank held 1.09 times the mean number of cells.
3. The seed-0 logs from August use the same `step()` code. For an almost identical random city (3.41 vs. 3.37 million cells), the one-rank run took 250.6 s in August but 424.8 and 430.6 s in September, while the 32-rank runs took about the same (14.7 s vs. 11.0/15.0 s). With the August baseline, the speedup at 8 ranks would be about 7.2 ($E \approx 0.9$) instead of 12.3. The graphs differ slightly, so this is an indication, not a proof.
4. The ring, balanced and with the largest data per rank, shows no systematic superlinearity, which argues against a pure cache effect.

Attribution. The old timer does not explain the excess: it already had a barrier before the start, and the per-step `allreduce` synchronised all ranks, so rank 0 stopped almost together with the slowest rank. The per-step collective made parallel runs slower, not faster. The remaining causes are those that the Slurm part of the fix removes: nodes shared with other jobs (especially harmful for the one-rank baseline, which ran up to 18 minutes), no pinning and no fixed frequency. The factor of 1.7 between the two baselines shows that the baseline depended on the environment. A cache effect (from 3.4 to 0.1

Table 3: Measured strong-scaling results of the old campaign: median wall time T of 2 runs with minimum and maximum, speedup S , efficiency E , Karp-Flatt metric e , and the maximum over ranks of the computation time C and the communication time.

Family	p	T [s]	min–max [s]	S	E	e	C [s]	comm. [s]
random	1	427.71	424.79–430.62	1.00	1.00	–	402.23	2.35
random	2	210.82	206.03–215.62	2.03	1.01	–0.0142	195.83	46.51
random	4	95.23	92.13–98.34	4.49	1.12	–0.0364	86.28	41.81
random	8	34.83	34.55–35.12	12.28	1.53	–0.0498	29.34	13.08
random	16	18.87	18.78–18.95	22.67	1.42	–0.0196	15.12	8.82
random	32	13.00	11.01–15.00	32.89	1.03	–0.0009	9.78	8.29
leipzig	1	105.46	105.43–105.48	1.00	1.00	–	99.17	1.81
leipzig	2	45.92	45.54–46.30	2.30	1.15	–0.1291	41.69	3.96
leipzig	4	23.47	23.42–23.51	4.49	1.12	–0.0367	21.95	10.84
leipzig	8	13.36	13.16–13.56	7.89	0.99	+0.0019	11.97	7.44
leipzig	16	7.19	6.79–7.59	14.67	0.92	+0.0061	5.37	4.46
leipzig	32	4.20	3.97–4.42	25.14	0.79	+0.0088	2.84	2.92
ring	1	1055.62	1044.62–1066.63	1.00	1.00	–	967.76	48.88
ring	2	560.86	559.53–562.18	1.88	0.94	+0.0626	506.45	85.88
ring	4	255.92	249.54–262.29	4.12	1.03	–0.0101	228.28	42.88
ring	8	146.01	129.07–162.96	7.23	0.90	+0.0152	130.53	46.66
ring	16	67.44	61.33–73.56	15.65	0.98	+0.0015	58.20	23.27
ring	32	36.51	30.38–42.63	28.92	0.90	+0.0034	31.23	15.47

million cells per rank) cannot be excluded but was not tested, so it is not claimed. The measured efficiencies are not reliable evidence of superlinear scaling.

4.6 Projected results after the fix (model, not measured)

Since the corrected campaign could not run, a model applied to the measured data estimates what it would show (`projection_model.py`). The non-compute time $X(p) = T(p) - C(p)$ (communication, waiting and uninstrumented code) is kept, and only the implausible computation time is replaced by the one-rank value scaled with the static imbalance $I(p)$:

$$T^*(1) = T(1), \quad T^*(p) = X(p) + \frac{C(1) I(p)}{p}, \quad E^*(p) = \frac{T(1)}{p T^*(p)}. \quad (7)$$

$I(p)$ is taken for the graphs that were run: the mean of the seed-1 and seed-2 graphs for the random city, the 20 km Leipzig map (recomputed from current OSM data: 1 037 325 vs. 1 037 317 cells), and $I = 1$ for the ring. A smooth curve

$$T_{\text{fit}}(p) = T(1) \left(f + \frac{1-f}{p} \right) + c \log_2 p, \quad f, c \geq 0, \quad (8)$$

is fitted to $T^*(p)$ for $p \geq 2$ by non-negative least squares, so it cannot be superlinear. The assumptions are:

1. computation time is proportional to the cells of the busiest rank, with no cache or frequency advantage (if a real cache effect exists, the projection is pessimistic);

2. $X(p)$ is unchanged by the fix, and the saving from the single `allreduce` is not credited (conservative);
3. $T(1)$ is taken as measured; if the true exclusive-node baseline is shorter, all projected efficiencies drop proportionally;
4. every input is a median of only two runs, and the Leipzig projection refers to the 20 km problem, not the 10 km problem of the corrected campaign.

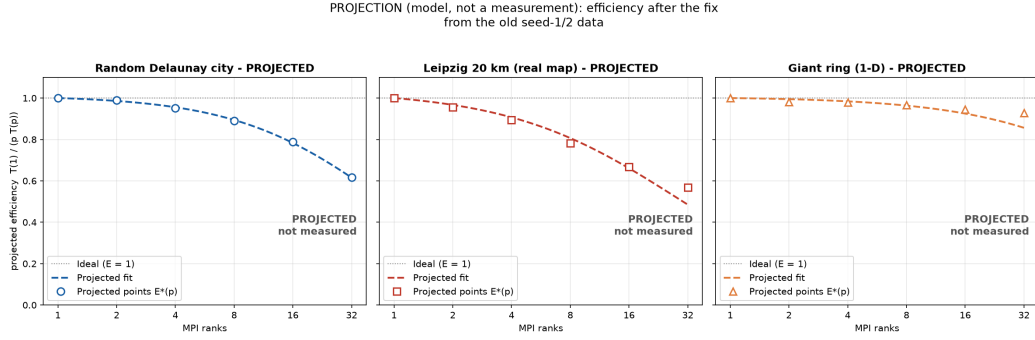


Figure 5: Projected parallel efficiency (model, not measured): corrected points $E^*(p)$ and fitted curve (dashed).

Projected values. Figure 5 shows the result; all values are in table 4 (appendix D). For the random city ($f = 0.0105$, $c = 0.80$ s) the projected efficiency is 0.89 at 8, 0.78 at 16 and 0.62 at 32 ranks ($S_{\text{fit}}(32) = 19.7$), close to the static bound. For Leipzig ($f = 0.0343$, $c = 0$) the point at 32 ranks gives $E^*(32) = 0.57$ and the fit, which underestimates it, 0.49 ($S_{\text{fit}}(32) = 15.5$). For the ring ($f = 0.0054$, $c = 0$), $E^*(32) = 0.93$ and $E_{\text{fit}}(32) = 0.86$ ($S_{\text{fit}}(32) = 27.4$). The Karp-Flatt metric of the fit is nearly constant by construction of the model, so it is not a finding.

4.7 Weak scaling

No weak-scaling campaign [Gus88] was performed. The ring with 20 000 p segments at fixed density (400 000 cells per rank, at most two partners) is the natural candidate.

4.8 Performance analysis tools

The measurements rely on instrumentation in the program: the `MPI.Wtime` timer and the per-rank computation/communication split, which located the superlinear excess in the computation time. A profile during development showed that NetworkX out-edge queries per vehicle dominated the runtime, so both backends now cache the out-edges per node.

4.9 Discussion of the results

The design (sections 2.6 and 4.2) predicts nearly linear scaling for the ring, an efficiency bounded by the cell imbalance for the random city, and the weakest scaling for Leipzig, the smallest problem (about 32 000 cells per rank at 32 ranks). The measured ring, and Leipzig from 8 ranks on, match this; the random city does not, for the reasons in

section 4.5. The projection follows the expected order (ring, random city, Leipzig) and lies below the static bounds for the random city and Leipzig, so the imbalance of the x-slab partition is the main limit, and a cell-weighted partition would be the first improvement. The projection cannot replace the corrected campaign.

5 Challenges and discussion

The mixed campaign. Earlier evaluations, including a previous version of the slides, treated the logs of seeds 0, 1 and 2 as three repetitions and reported speedups of 28.9 (random, 32 ranks), 25.9 (Leipzig, 32 ranks) and 14.3 (ring, 16 ranks). This was wrong: the seed-0 logs were written in August by commit `c180866` and are a different problem for Leipzig (4 765 nodes, 193 278 cells) and for the ring (density 0.25), while seeds 1 and 2 were run in September by commit `2ca5a5a`. Without a provenance header in the logs the mix went unnoticed; this report uses only seeds 1 and 2.

The benchmark fix. The fix (commit `027defb` and follow-ups) brings code and job scripts in line with the protocol of section 2.7. In the code: barriers before starting and stopping the timer, the maximum over ranks as wall time, `-check-every` and `-quiet`. In the job scripts: one exclusive node per run, fixed tasks per node, core binding, fixed CPU frequency, a fixed seed with repetitions, and a provenance header in every log (host, git commit, CPU, governor, bindings, command). `scaling_logs.py` rejects logs that break these rules, and would have rejected the mixed campaign. As argued in section 4.5, the environment part addresses the likely cause of the superlinear results.

Queue. Exclusive whole nodes were hard to obtain: neither the corrected campaign nor the corrected serial sweep started before the deadline, and short wall-time limits for backfilling did not help enough. In hindsight, the protocol should have been fixed, and the campaign started, earlier. Compute nodes also have no internet access, so `jobs/prefetch_maps.sh` warms the OSMnx cache on a login node for every radius used.

Other approaches. Cell-weighted cuts or a graph partitioner that minimises crossings (as in TRANSIMS [NR01]) would remove the imbalance of table 2. A counter-based generator keyed by (seed, step, vehicle id) would make trajectories independent of the decomposition. A compiled kernel (Numba, C++), buffer-based `Isend/Irecv` of NumPy arrays instead of pickled dictionaries, and neighbourhood collectives would improve absolute performance; a compiled kernel would also make threads inside a node worthwhile.

6 Conclusion

The project delivered a Python package that simulates traffic jams with the NaSch model on rings, random cities, image-derived networks and real OSM maps, with a serial backend and an MPI backend based on replicated topology, x-slab ownership, entry halos, migration requests and owner-side arbitration. The serial model reproduces spontaneous jams and the fundamental diagram, and the MPI backend conserves vehicles and agrees with it statistically, so the functional goals were reached.

The performance goals were reached only in part. The measured speedups at 32 ranks (32.9 random city, 25.1 Leipzig, 28.9 ring) come from shared, unpinned nodes; the random city exceeds its static efficiency bound, so these numbers are not reliable evidence of superlinear scaling. A projection gives efficiencies at 32 ranks of about 0.62, 0.49–0.57 and 0.86–0.93 (plausible, not measured). The serial runtime grows approximately linearly with the number of cells.

The main lessons are that a distributed traffic simulation needs a clear rule for boundary conflicts, that a cheap global invariant is an effective test, and that performance numbers only mean something once timing, baseline, environment and provenance are fixed before the first run. The next step is to run the corrected campaign (`jobs/submit_scaling_all.sh`) and replace the projection with measurements, followed by the serial sweep, a weak-scaling study, a validation with routed vehicles, a cell-weighted partitioner and a compiled kernel.

References

- [Amd67] Gene M. Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *Proceedings of the AFIPS Spring Joint Computer Conference*. 1967, pp. 483–485.
- [Boe17] Geoff Boeing. “OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks”. In: *Computers, Environment and Urban Systems* 65 (2017), pp. 126–139.
- [DF21] Lisandro Dalcin and Yao-Lung L. Fang. “mpi4py: Status Update After 12 Years of Development”. In: *Computing in Science & Engineering* 23.4 (2021), pp. 47–54.
- [Gus88] John L. Gustafson. “Reevaluating Amdahl’s law”. In: *Communications of the ACM* 31.5 (1988), pp. 532–533.
- [HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. “Exploring Network Structure, Dynamics, and Function using NetworkX”. In: *Proceedings of the 7th Python in Science Conference (SciPy2008)*. 2008, pp. 11–15.
- [KF90] Alan H. Karp and Horace P. Flatt. “Measuring parallel processor performance”. In: *Communications of the ACM* 33.5 (1990), pp. 539–543.
- [Lop+18] Pablo Alvarez Lopez et al. “Microscopic Traffic Simulation using SUMO”. In: *21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2018, pp. 2575–2582.
- [Mes21] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 4.0*. 2021. URL: <https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf>.
- [NR01] Kai Nagel and Marcus Rickert. “Parallel implementation of the TRANSIMS micro-simulation”. In: *Parallel Computing* 27.12 (2001), pp. 1611–1639.
- [NS92] Kai Nagel and Michael Schreckenberg. “A cellular automaton model for free-way traffic”. In: *Journal de Physique I* 2.12 (1992), pp. 2221–2229.

A Work sharing

This is an individual project. All parts were carried out by the author, Salvador Costilla Caballero a part of the code and the report were written with the help of AI assistants (Claude Code, GitHub Copilot). The work can be divided into the following parts:

- design of the package structure (networks, models, runners) and the command-line interface;
- network generators for ring, random Delaunay cities, image-derived networks and OSMnx maps;
- the serial Nagel-Schreckenberg simulator and its unit tests;
- the MPI backend `ParallelNaSch` (ownership, halo exchange, migration, arbitration, conservation check, timing);
- visualisation (animations, heatmaps, space-time and fundamental diagrams);
- Slurm job scripts, the benchmarking campaigns on the GWDG cluster, the log parser and the plotting scripts;
- this report and the presentation.

B Code samples

Listings 2 to 4 show the ownership computation, the first communication round and the owner-side arbitration referred to in section 3.

```

1 edges = sorted(G.edges())
2 xs = sorted(G.nodes[u]["pos"][0] for u, _ in edges)
3 cuts = [xs[int(len(xs) * r / self.size)] for r in range(1, self.size)]
4 self.owner = {}
5 for u, v in edges:
6     x = G.nodes[u]["pos"][0]
7     r = sum(1 for c in cuts if x >= c)
8     self.owner[(u, v)] = min(r, self.size - 1)

```

Listing 2: Edge ownership by x-slabs of the source node (`runners/mpl.py`).

```

1 out_halos = {r: {} for r in self.neighbors_up}
2 for e, ranks in self.upstream.items():
3     snap = self.cells[e][:self.halo_len].copy() # halo_len = v_max + 1
4     for r in ranks:
5         out_halos[r][e] = snap
6 reqs = [comm.isend(out_halos[r], dest=r, tag=1) for r in self.neighbors_up]
7 halos = {}
8 for r in self.neighbors_down:
9     halos.update(comm.recv(source=r, tag=1))
10 for rq in reqs:
11     rq.wait()

```

Listing 3: Round 1, entry halos (`ParallelNaSch.step`, abridged).

```

1 for e, cand in my_reqs.items():
2     arb = random.Random(f"{self.seed}/{self.step_counter}/{e}")
3     arb.shuffle(cand)
4     arr = self.cells[e]
5     for (vid, want, remaining, route, route_idx, spawn_step) in cand:
6         entry = -1
7         for i in range(min(want, len(arr) - 1) + 1):
8             if arr[i] != EMPTY:
9                 break
10            entry = i
11            # entry >= 0: claim arr[entry], create the vehicle if it migrates in
12            # entry < 0: reject; the source rank brakes it at the end of its edge

```

Listing 4: Owner-side arbitration of entry requests (abridged).

Listing 5 shows phase 2 of the distributed step, in which moves within an edge are executed immediately and moves across an intersection are turned into entry requests. Listing 6 shows how a rank applies the verdicts to its crossing vehicles after round 3.

```

1 self.rng.shuffle(plans)
2 requests = {} # target edge -> request list
3 crossing_local = {} # vid -> (veh, next_edge, remaining)
4 arrived_now = []
5 for veh, next_edge in plans:
6     if veh.v == 0:
7         veh.stats_steps_stopped += 1
8         continue
9     arr = self.cells[veh.edge]
10    remaining = len(arr) - 1 - veh.cell
11    if veh.v <= remaining: # stays on its edge
12        arr[veh.cell] = EMPTY
13        veh.cell += veh.v
14        arr[veh.cell] = veh.vid
15        continue
16    if next_edge is None: # end of route: arrival
17        arr[veh.cell] = EMPTY
18        arrived_now.append(veh.vid)
19        continue
20    want = veh.v - remaining - 1 # target cell on the next edge
21    requests.setdefault(next_edge, []).append(
22        (veh.vid, want, remaining,
23         veh.route, veh.route_idx, veh.spawn_step))
24    crossing_local[veh.vid] = (veh, next_edge, remaining)

```

Listing 5: Phase 2 of ParallelNaSch.step: local moves and entry requests.

```

1 for vid, (veh, next_edge, remaining) in crossing_local.items():
2     entry = my_verdicts[vid]
3     arr = self.cells[veh.edge]
4     if entry < 0: # rejected: brake at edge end
5         arr[veh.cell] = EMPTY
6         veh.cell = len(arr) - 1
7         veh.v = remaining
8         if veh.v == 0:
9             veh.stats_steps_stopped += 1
10            arr[veh.cell] = veh.vid
11    elif next_edge in self.cells: # crossed into my own edge
12        arr[veh.cell] = EMPTY

```

```

13     veh.edge = next_edge
14     veh.cell = entry
15     veh.v = remaining + 1 + entry
16     if veh.route is not None:
17         veh.route_idx += 1
18     else:                                     # migrated to another rank
19         arr[veh.cell] = EMPTY
20         del self.vehicles[vid]

```

Listing 6: Applying the verdicts after round 3.

Listing 7 shows how one strong-scaling job is submitted and launched (jobs/submit_scaling_all.sh and jobs/scaling_family.job, abridged).

```

1 # submit_scaling_all.sh: one job per (family, rank count)
2 sbatch --nodes=1 --ntasks=$NP --ntasks-per-node=$NP --exclusive \
3     --cpu-freq=$CPU_FREQ --time=$LIMIT \
4     --export=ALL,LABEL=$FAMILY,SEED=$SEED,REPS=$REPS,STEPS=$STEPS,... \
5     jobs/scaling_family.job
6
7 # scaling_family.job: REPS repetitions of the identical problem
8 LAUNCH="srun --mpi=pmix -n $NP --cpu-bind=verbose,cores --cpu-freq=$CPU_FREQ"
9 CMD="congestion run ${SIZEARG} --steps ${STEPS} --backend mpi \
10     --seed ${SEED} --check-every ${STEPS} --quiet"
11 for REP in $(seq 0 $((REPS - 1))); do
12     $LAUNCH $CMD >> results/strong_scaling/scaling_${LABEL}_np${NP}_s${SEED}_r${REP}.log
13 done

```

Listing 7: Submission and launch of one job of the corrected strong-scaling campaign (abridged; submitted, but not run before the deadline).

C Code repository

The source code, job scripts, plotting scripts and results are available at https://gitlab.gwdg.de/s.costillacaballer/traffic_jam. The script that computes the static partition statistics in table 2 and draws fig. 1 is report_src/scripts/partition_stats.py. All figures of the report and the slides are regenerated by make_figures.sh. The measured strong-scaling figures are produced by plot_scaling.py (time, and the rows of table 3 with -table-tex), plot_scaling_speedup.py, plot_scaling_eff.py and plot_scaling_kf.py, called with -input-dir results/parallel_scaling -seeds 1 2 -min-runs 2 -old-timer. The projection is computed by projection_model.py and plotted by plot_projection.py into results/projected/; its imbalance input comes from report_src/scripts/partition_imbalance_old.py. The serial growth figures are produced by plot_growth.py and plot_growth_eff.py from results/serial_growth/. Without the old-campaign options, the same scripts evaluate the corrected campaign in results/strong_scaling/, which does not contain data yet.

D Additional figures and tables

Figures 6 to 8 complement the measured results of section 4.4; fig. 9 and table 4 complement the projection of section 4.6; fig. 11 complements the serial growth of section 4.3.

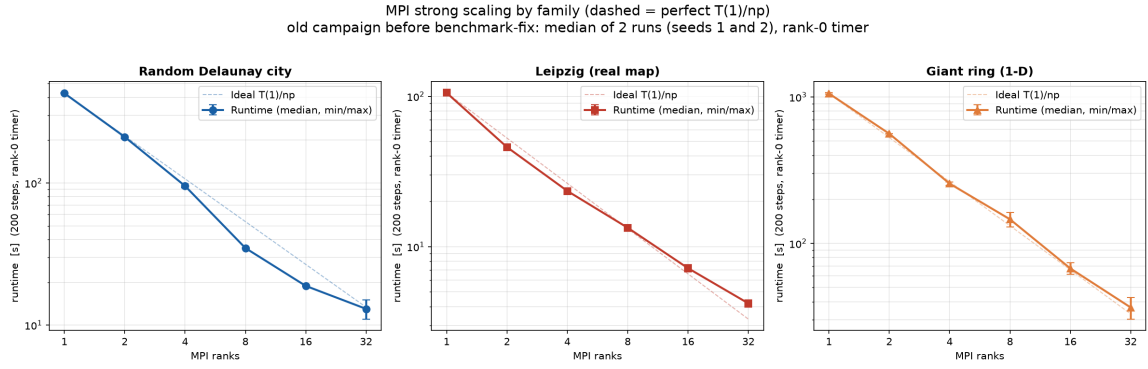


Figure 6: Measured runtime of the old campaign (shared, unpinned nodes), 200 steps, median of 2 runs (seeds 1 and 2) with minimum and maximum; dashed: ideal $T(1)/p$.

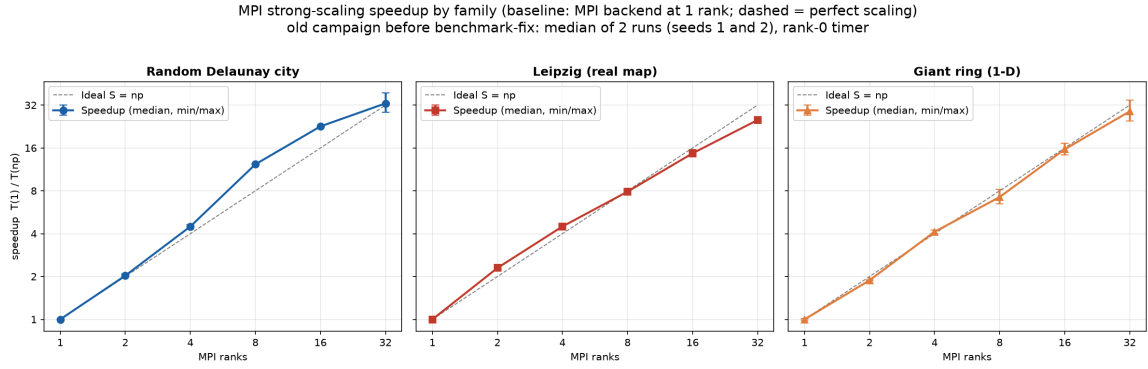


Figure 7: Measured speedup of the old campaign (shared, unpinned nodes), median of 2 runs (seeds 1 and 2); dashed: ideal speedup.

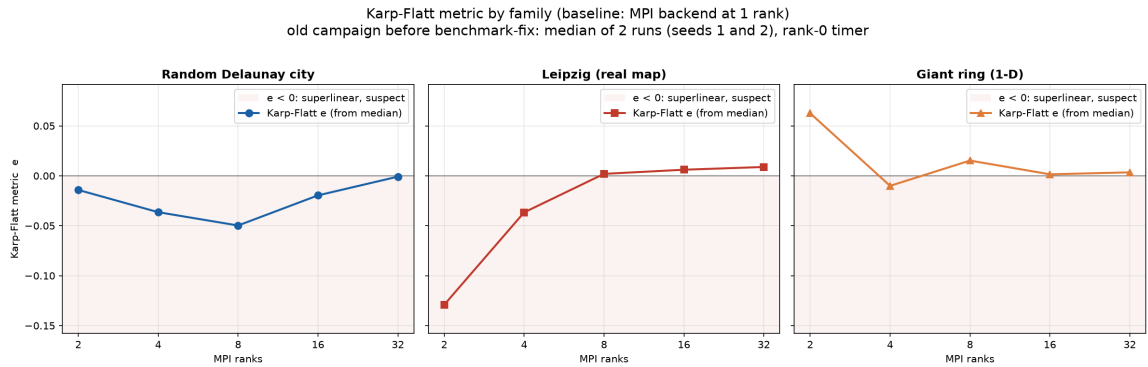


Figure 8: Measured Karp-Flatt metric $e(p)$ of the old campaign, median of 2 runs (seeds 1 and 2). Negative values are suspected measurement artefacts (section 4.5).

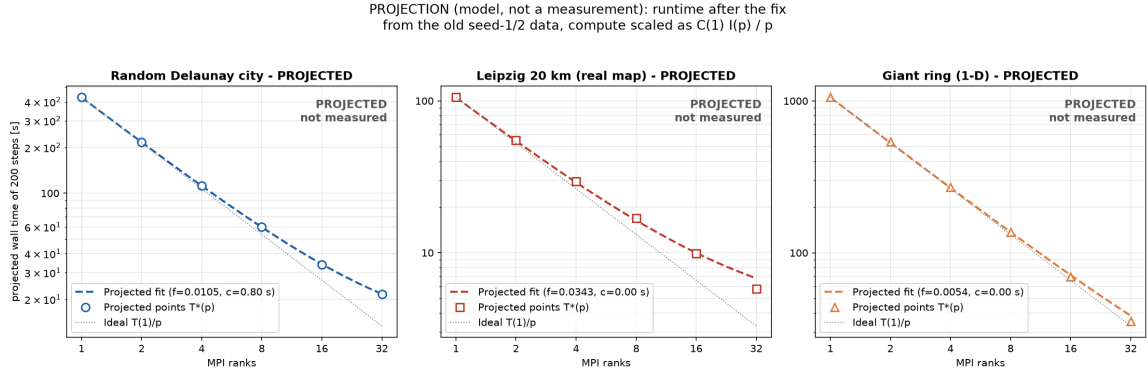


Figure 9: Projected runtime after the fix (model, not measured): corrected points $T^*(p)$ and fitted curve (dashed), section 4.6.

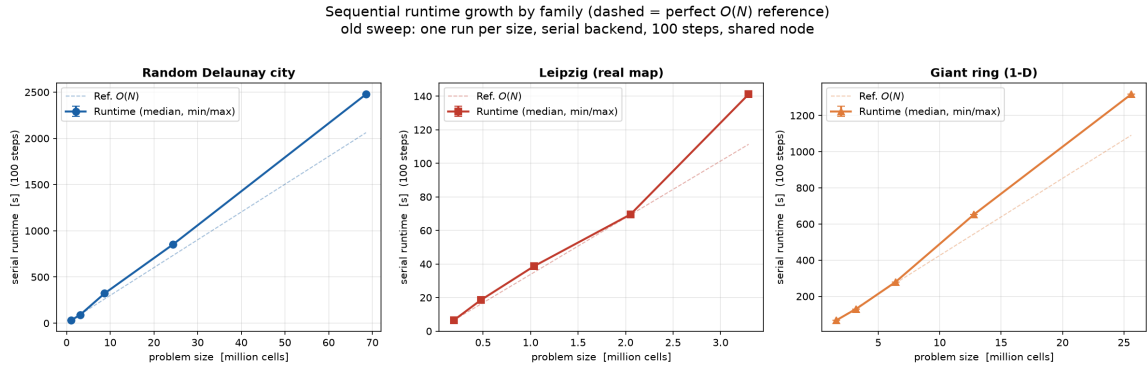


Figure 10: Serial runtime as a function of problem size (dashed: $O(N)$ reference), 100 steps at density 0.10. Single runs per size on shared, unpinned nodes (August, commit c180866).

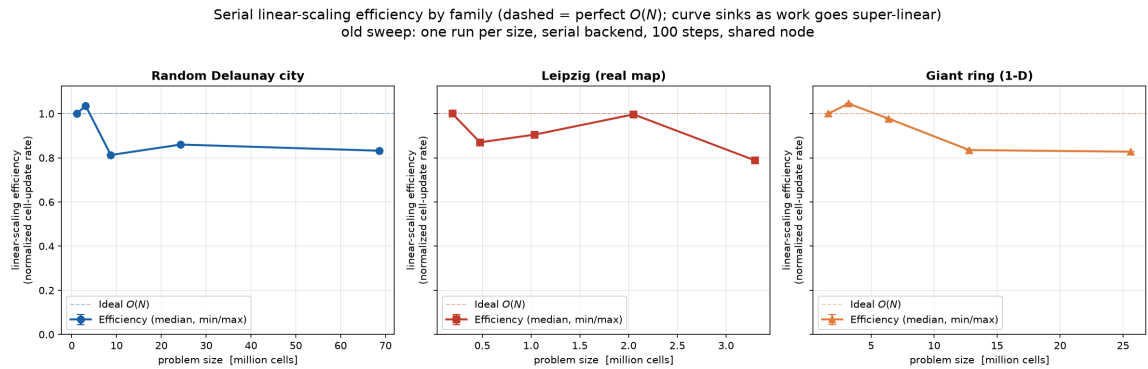


Figure 11: Serial cell-update rate normalised to the smallest size of each family (1 = perfectly linear growth); old single runs on shared, unpinned nodes.

Table 4: Projected strong scaling after the fix (model, not measured): static imbalance $I(p)$, projected points $T^*(p)$ and $E^*(p)$, and the fitted curve with its speedup, efficiency and Karp-Flatt metric.

Family	p	Projected values (model, not measured)						
		$I(p)$	$T^*(p)$ [s]	$E^*(p)$	T_{fit} [s]	S_{fit}	E_{fit}	e_{fit}
random	1	1.000	427.71	1.00	427.71	1.00	1.00	—
random	2	1.001	216.21	0.99	216.90	1.97	0.99	+0.0143
random	4	1.029	112.39	0.95	111.90	3.82	0.96	+0.0155
random	8	1.085	60.03	0.89	59.80	7.15	0.89	+0.0169
random	16	1.201	33.94	0.79	34.14	12.53	0.78	+0.0185
random	32	1.469	21.68	0.62	21.72	19.70	0.62	+0.0202
leipzig	1	1.000	105.46	1.00	105.46	1.00	1.00	—
leipzig	2	1.027	55.14	0.96	54.53	1.93	0.97	+0.0343
leipzig	4	1.129	29.50	0.89	29.07	3.63	0.91	+0.0343
leipzig	8	1.246	16.84	0.78	16.34	6.45	0.81	+0.0343
leipzig	16	1.298	9.87	0.67	9.98	10.57	0.66	+0.0343
leipzig	32	1.436	5.80	0.57	6.79	15.52	0.49	+0.0343
ring	1	1.000	1055.62	1.00	1055.62	1.00	1.00	—
ring	2	1.000	538.29	0.98	530.67	1.99	0.99	+0.0054
ring	4	1.000	269.57	0.98	268.19	3.94	0.98	+0.0054
ring	8	1.000	136.45	0.97	136.95	7.71	0.96	+0.0054
ring	16	1.000	69.74	0.95	71.33	14.80	0.92	+0.0054
ring	32	1.000	35.52	0.93	38.52	27.40	0.86	+0.0054