



Salvador Costilla Caballero

Simulating Traffic Jams in City Networks

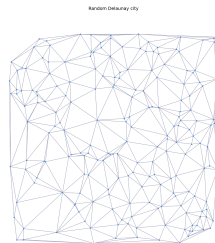
A parallel Nagel-Schreckenberg cellular automaton with MPI

Table of contents

- 1 Solution approach
- 2 Sequential solution
- 3 Parallelized solution
- 4 Performance analysis

The Problem: Cars, City Networks, and Traffic Jams

- Simulate cars in a city network and the **phantom traffic jams** that emerge.
- Traffic jams are *emergent*: simple local rules lead to global congestion without a single cause.
- **Goal:** Reproduce traffic physics and scale to large networks on the **GWDG cluster** (MPI).



Random Delaunay

road network

Why is this a great HPC problem?

Millions of cells updated per timestep, local interaction rules, and natural spatial decomposition for parallelization.

Solution approach: the Nagel–Schreckenberg model

Each road is a strip of **7.5 m cells**; each car occupies one cell and carries an integer velocity. Every timestep ($= 1$ s), each car applies four rules:

- 1 **Accelerate:** $v \leftarrow \min(v + 1, v_{\max})$
- 2 **Brake:** $v \leftarrow \min(v, \text{gap ahead})$
- 3 **Randomize:** with probability p : $v \leftarrow v - 1$
- 4 **Move:** advance v cells

The key insight

Rule 3 (random braking) is what creates **phantom jams** — traffic jams that emerge with no accident or bottleneck, exactly as on a real motorway.

Why this model, and the tools it needs

Why Nagel–Schreckenberg

- Minimal, yet reproduces real traffic physics
- A *cellular automaton*: local, discrete, and therefore parallel-friendly

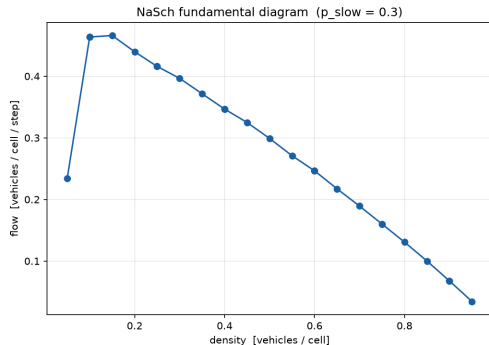
Libraries — and why

- numpy — fast cell arrays
- NetworkX — road graph, shortest paths
- mpi4py — MPI bindings for Python
- OSMnx — real maps (OpenStreetMap)

Four environments, one CLI

The same simulator runs on four kinds of network, chosen from the command line:

- ring — periodic road (classic NaSch)
- random — Delaunay city
- image — network from an image skeleton
- map — real city from OpenStreetMap



Validation: the fundamental diagram

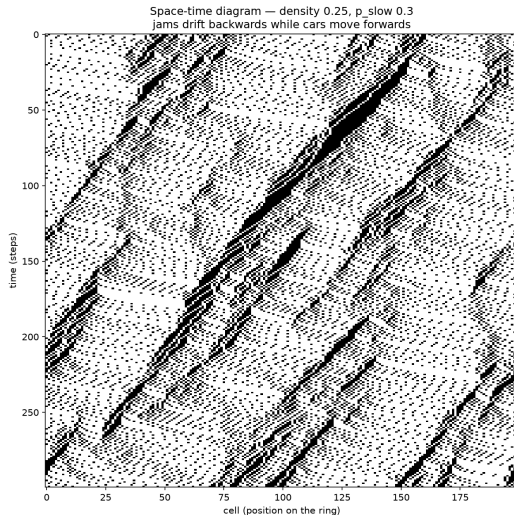
Sequential solution

The serial step, two phases:

- 1 **Velocities** — each car reads a snapshot of the occupancy and applies rules 1–3.
- 2 **Movement** — cars move (rule 4); intersection conflicts resolved by a live occupancy check.

Validated against theory:

- fundamental diagram peaks at $\rho \approx 0.13$
- jams drift **backward** (space-time diagram)



Why sequential is not enough

- A realistic city has **millions of cells**; one core updates them one after another.
- Measured: random city with 3.4M cells, 200 steps = 428 s on a single MPI rank (old campaign, median of 2 runs).
- Serial runtime grows about linearly with city size (1.9–3.5M cell updates/s): large cities or parameter sweeps become slow.
- But the physics is **local**: a car only looks a few cells ahead — distant parts are independent within a step.

Consequence

The workload is a natural fit for **domain decomposition**: split the city across processes, communicate only at the seams.

What is eligible for parallelization?

Eligible

- Velocity update — independent per car
- Spatial regions — local interactions only
- Parameter sweeps — independent runs

Needs coordination

- Intersection conflicts
- Cars crossing region borders

Chosen: MPI domain decomposition

- Partition edges by x into slabs, one per rank
- Each rank owns its streets' cells
- **Halo exchange** of border cells each step
- Cars migrate as messages

Why MPI, not OpenMP? Python's GIL blocks thread speedup; MPI launches *processes*, scaling within and across nodes with one code path.

The distributed timestep

Three communication rounds per step, only between neighboring ranks:

- 1 **Entry halos** — owners send the occupancy of their first $v_{\max} + 1$ cells upstream, so cars can compute their gap across a border.
- 2 **Migration** — a car crossing into a remote edge is sent (id, velocity, remaining route) to that edge's owner.
- 3 **Verdicts** — the owner resolves entry conflicts deterministically and replies accepted / rejected.

Correctness invariant

The global vehicle count is asserted constant (every step in correctness runs).
Fundamental diagram of 2, 4, 8 ranks agrees with the serial backend on a ring and a random city (40 seeds, $|z| \leq 2.93$).

How performance was measured (old campaign)

Setup that actually ran

- Random city (3.4M cells), Leipzig 20 km (1.04M), ring (6.4M); 200 steps, 1–32 ranks
- Partition medium: shared nodes, no core pinning, no fixed CPU frequency
- Seeds 1 and 2: **median of 2 runs** (below the 3-run rule)

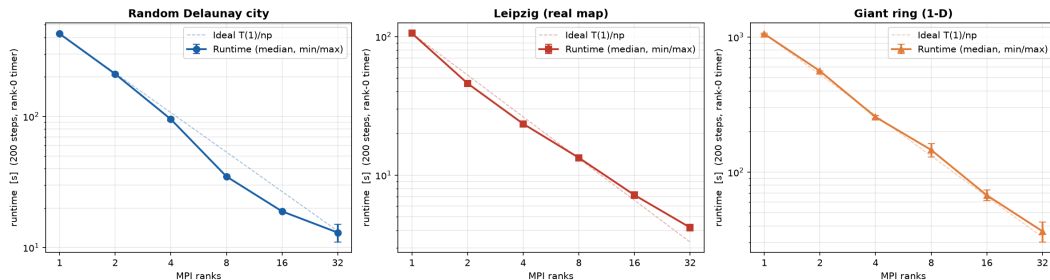
Corrected campaign submitted, not run before the deadline: all numbers below are from the old campaign.

Protocol (now in the job scripts)

- Timed region = stepping loop only
- Barrier before start and stop; time = max over ranks
- Exclusive node, pinned ranks, fixed frequency
- Median of 3 runs, provenance header in every log

Old measured results — runtime

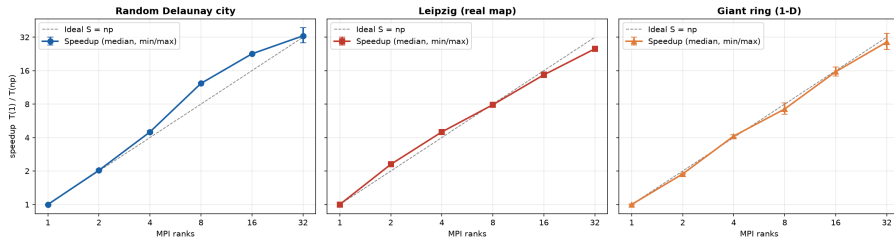
MPI strong scaling by family (dashed = perfect $T(1)/np$)
old campaign before benchmark-fix: median of 2 runs (seeds 1 and 2), rank-0 timer



Median of 2 runs (seeds 1 and 2), min/max bars; dashed: ideal $T(1)/p$. $T(1)$: 427.7 s (random), 105.5 s (Leipzig), 1055.6 s (ring).

Old measured results — speedup

MPI strong-scaling speedup by family (baseline: MPI backend at 1 rank; dashed = perfect scaling)
old campaign before benchmark-fix: median of 2 runs (seeds 1 and 2), rank-0 timer

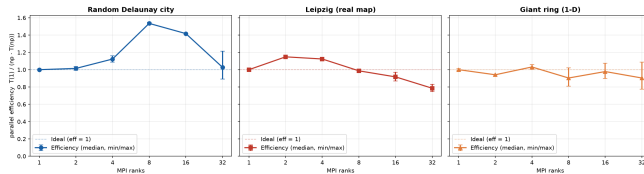


- At 32 ranks: $32.9\times$ (random), $25.1\times$ (Leipzig), $28.9\times$ (ring); ring at 16 ranks: $15.7\times$
- Random city superlinear from 4 ranks ($12.3\times$ at 8 ranks)

Median of 2 runs (seeds 1 and 2); baseline = MPI backend on 1 rank.

Old measured results — efficiency

MPI strong-scaling efficiency by family (baseline: MPI backend at 1 rank; dashed = perfect scaling)
old campaign before benchmark-fix: median of 2 runs (seeds 1 and 2), rank-0 timer



	random		Leipzig		ring	
p	S	E	S	E	S	E
2	2.03	1.01	2.30	1.15	1.88	0.94
4	4.49	1.12	4.49	1.12	4.12	1.03
8	12.28	1.53	7.89	0.99	7.23	0.90
16	22.67	1.42	14.67	0.92	15.65	0.98
32	32.89	1.03	25.14	0.79	28.92	0.90

Median of 2 runs (seeds 1 and 2).

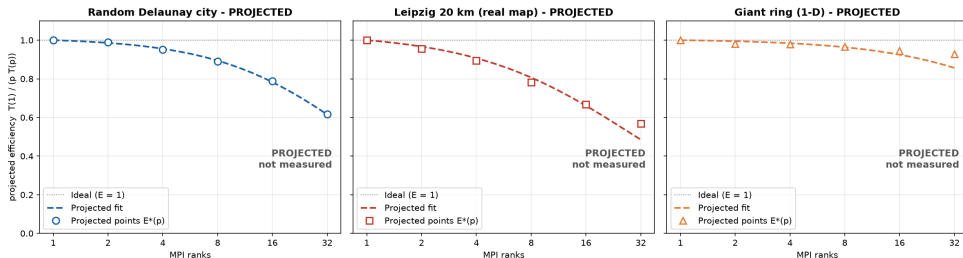
- Random: E up to 1.53, Karp-Flatt < 0 ; its cell imbalance caps $E(32)$ at 0.66–0.70
- Leipzig: $E(32) = 0.79$, Karp-Flatt $0.002 \rightarrow 0.009$
- Ring: E 0.90–1.03, large spread
- $E > 1$: treated as measurement artefact

What benchmark-fix corrected

- Timer: barrier before start and stop, max over ranks
- Reduce skipping: metrics `allreduce` every K steps (`-check-every`), `-quiet`
- Provenance in every log (commit, host, CPU, bindings); parser rejects incomplete campaigns
- Jobs: exclusive node, pinned ranks, fixed frequency, 3 repetitions
- Why $E > 1$ was an artefact: excess sits in the *compute* time; same step code gave a 1-rank baseline of 251 s (Aug) vs. 425–431 s (Sep). Old code already synchronised every step, so the timer fix explains almost none of it — the shared, unpinned nodes do.
- Mixed campaign: seed-0 logs came from older code and a different problem; earlier numbers (28.9×, 25.9×, 14.3×) are withdrawn.
- Queue: no exclusive `scc-cpu` node became free before the deadline.

Projected results after the fix (model, not measured)

PROJECTION (model, not a measurement): efficiency after the fix from the old seed-1/2 data



Model: keep measured non-compute time, replace compute time by $C(1)I(p)/p$ (static imbalance I), fit Amdahl + $c \log_2 p$.

Projected $E(32)$: random 0.62, Leipzig 0.49–0.57, ring 0.86–0.93 (serial fraction $f = 1.05\%$, 3.43% , 0.54%).

PROJECTED from the old data (2 runs per point, 20 km Leipzig) — not a measurement.

Conclusion

Accomplished

- NaSch on 4 topologies
- MPI domain decomposition
- Physics preserved
- Scaling measured; corrected run pending

Learned

- Halo exchange patterns
- Amdahl / Karp-Flatt analysis
- Superlinear = suspect first
- Provenance and exclusive nodes

Deliverables

- congestion CLI
- MPI engine
- SLURM scripts
- GWDG GitLab repo

Last Frame

Thank you!

Questions?