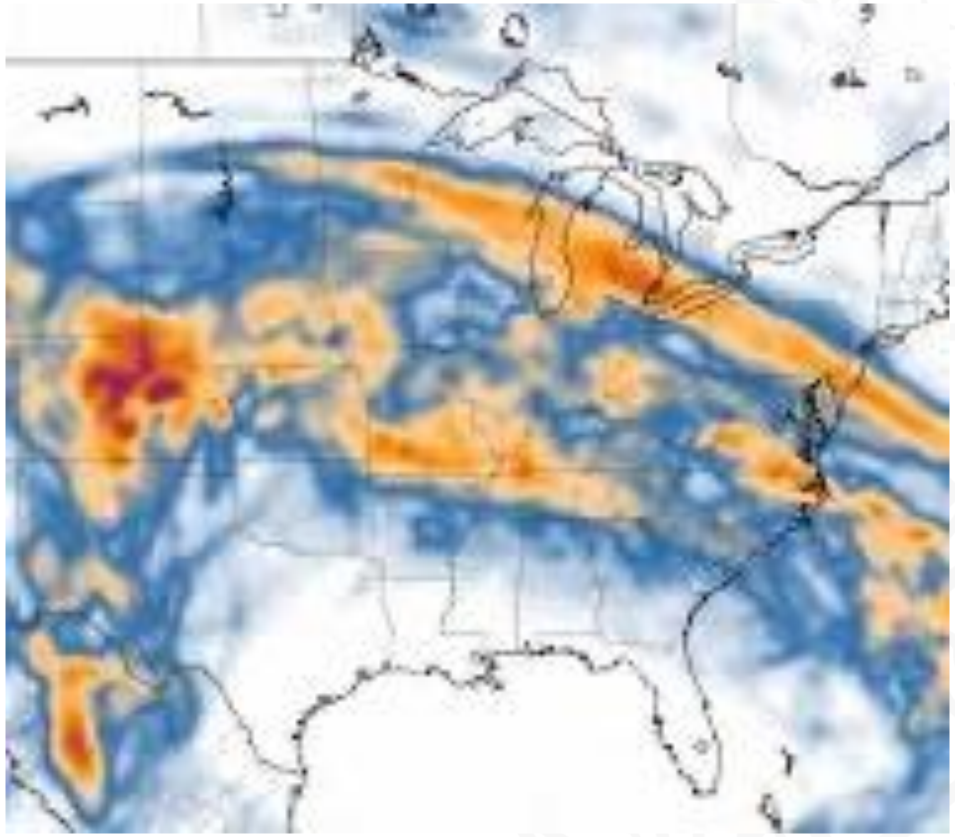
WEATHER DIFFUSION SIMULATION USING MPI

CSE 633

Sri Sai Rithvik Thota

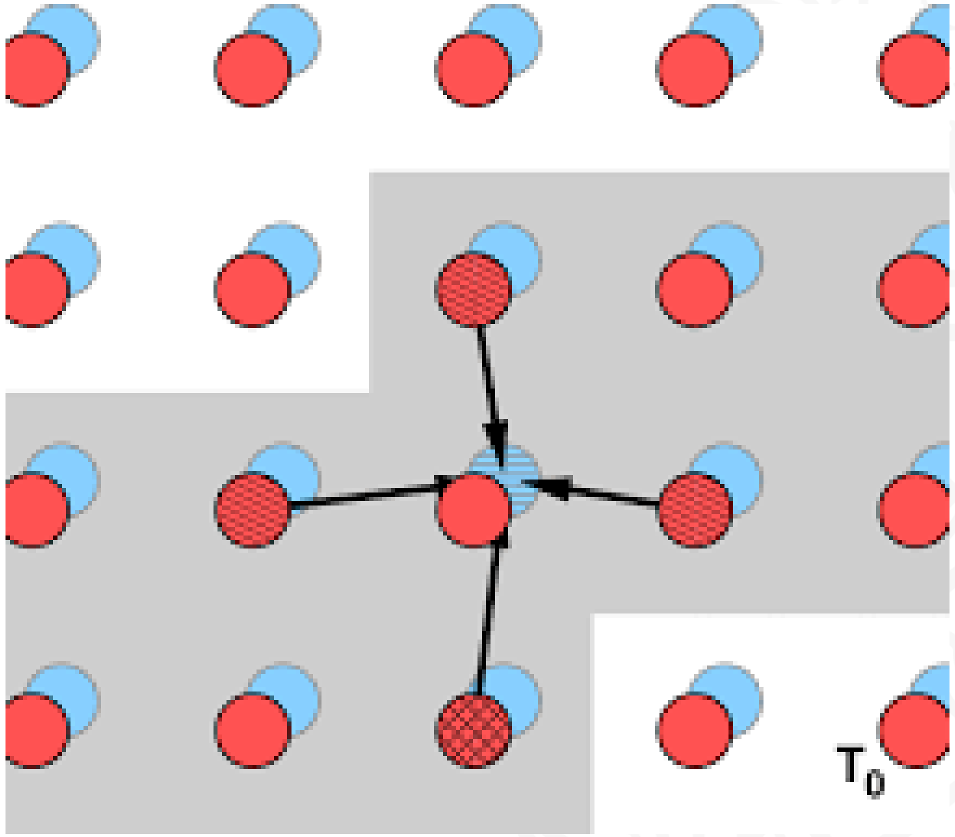
Motivation – Why Weather Diffusion?

- Diffusion models heat or pollutant spreading in the atmosphere.
- 2D grid + local stencil = natural fit for domain decomposition.
- Thousands of Jacobi iterations give a clear compute–communicate pattern.
- Great test case to study real scaling limits on actual hardware.
- Focus on both correctness and performance behavior.



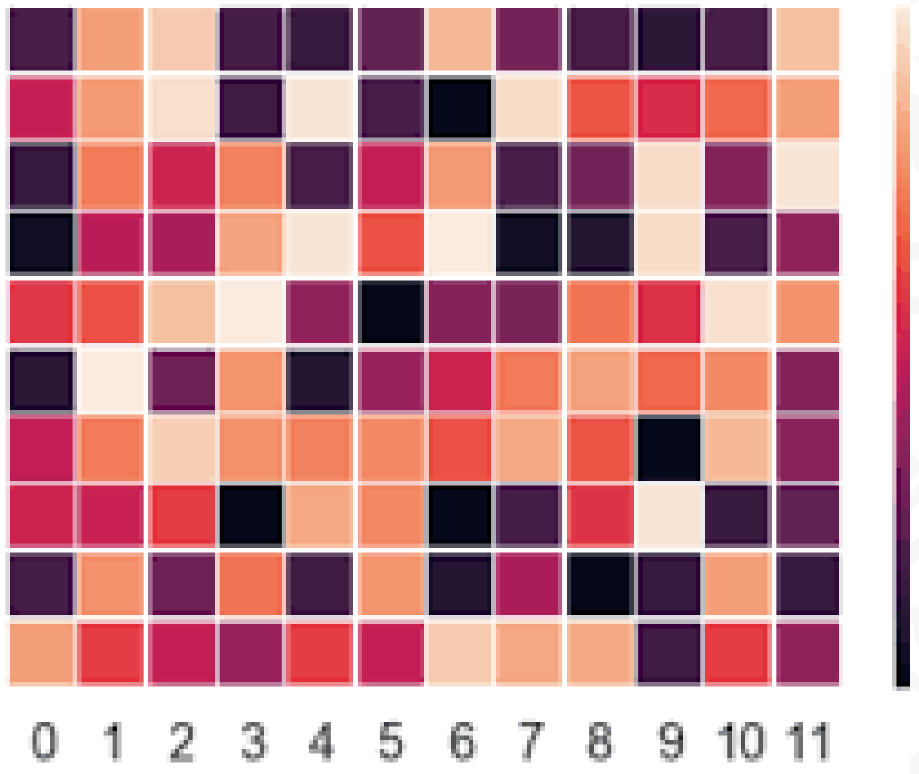
2D Diffusion Model and Jacobi Stencil

- Rectangular 2D grid up to 4096 × 4096 cells.
- Each cell stores a scalar (e.g., temperature or concentration).
- Jacobi 5-point stencil: average of north, south, east, and west neighbors.
- Dirichlet boundary conditions on selected edges, fixed in time.
- Iterate until residual < tolerance or max iterations reached.



Sequential Baseline Solver

- Pure Python + NumPy implementation of Jacobi iteration.
- Two grids (old/new) to avoid in-place update hazards.
- Vectorized updates over interior cells each iteration.
- Residual computed periodically to monitor convergence.
- Runtime, iterations and final residual exported as JSON.

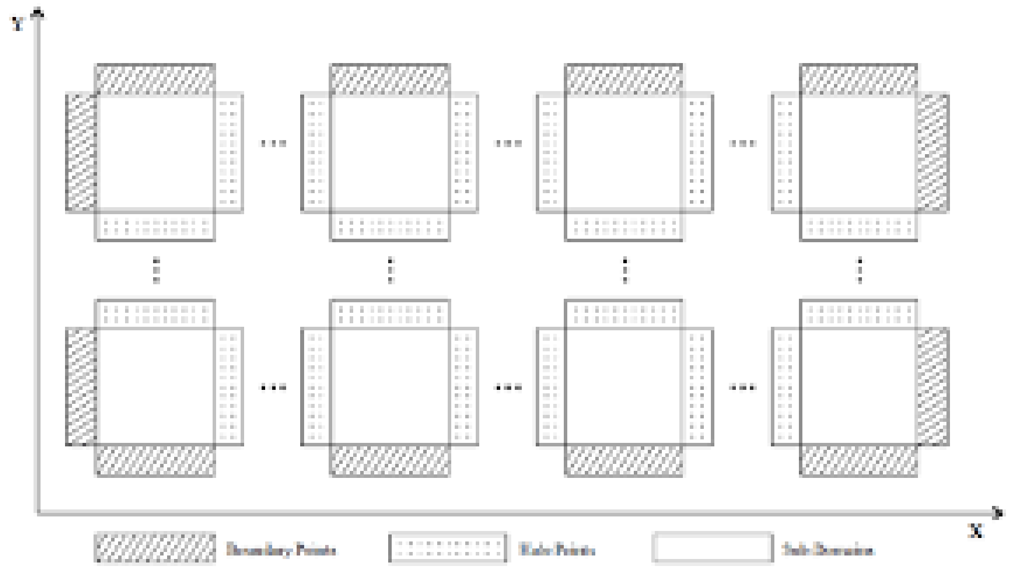


Why Parallelize? Limits of the Sequential Approach

- Large grids → millions of cell updates per iteration.
- Thousands of iterations needed for realistic tolerances.
- Computation is memory-bound and dominated by array traffic.
- Single core cannot exploit multi-core or cluster hardware.
- Motivation: distribute grid across processes using MPI.

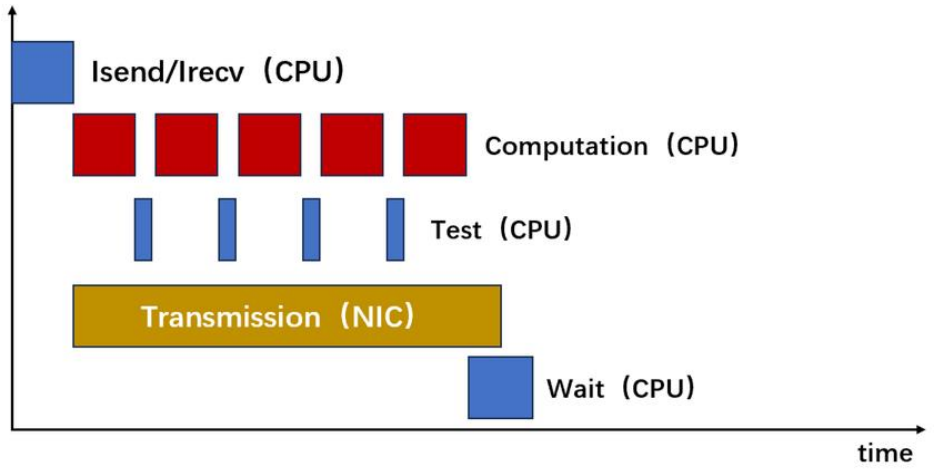
MPI Design – 2D Cartesian Decomposition

- Use MPI Cartesian communicator to form a 2D process grid.
- Global grid split into px × py rectangular subdomains.
- Each rank stores its local block plus halo (ghost) cells.
- Neighbors identified automatically (north, south, east, west).
- No master–worker: all ranks compute and communicate.



Parallel Jacobi Algorithm per Rank

- Compute interior cells that do not depend on halo data.
- Start non-blocking halo exchanges (Isend/Irecv) with neighbors.
- Overlap interior computation with in-flight communication.
- Wait for halos, then update boundary cells using new halo data.
- Use MPI_Allreduce to compute global residual and check convergence.

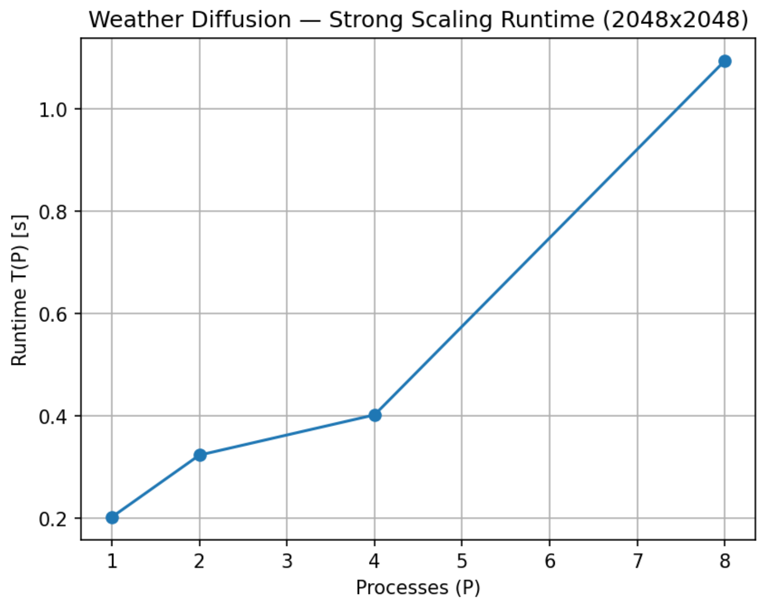


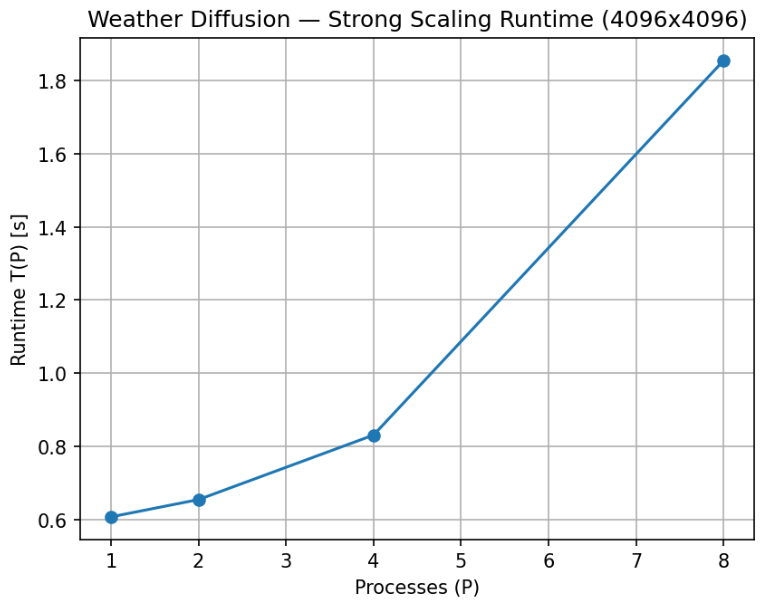
Experimental Setup

- Hardware: Apple Silicon system with 8 CPU cores (shared memory).
- Software: Python 3, NumPy, mpi4py, OpenMPI.
- Process counts: P = 1, 2, 4, 8.
- Strong scaling: fixed grids 2048^2 and 4096^2.
- Weak scaling: $\approx 2048^2$ cells per rank (problem grows with P).
- Each configuration executed 8 times; report mean timings.

Strong Scaling – Runtime vs Processes

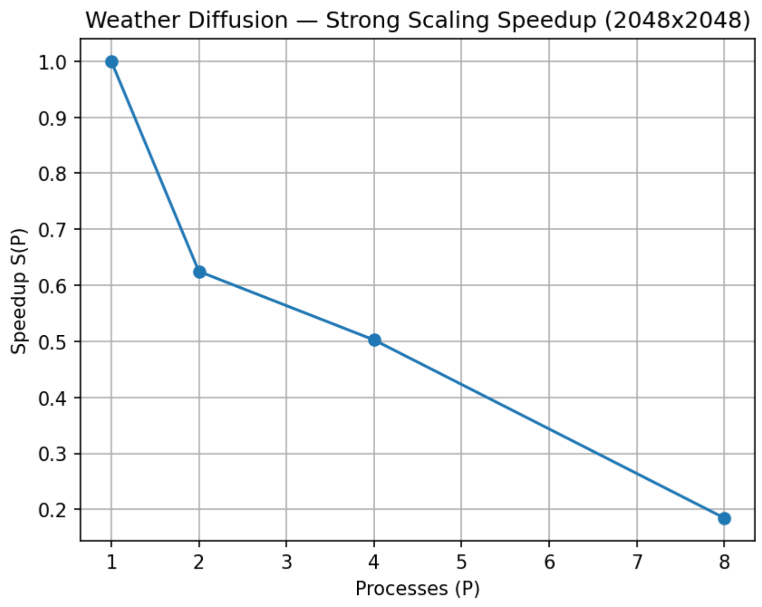
- Left: 2048 × 2048 grid | Right: 4096 × 4096 grid

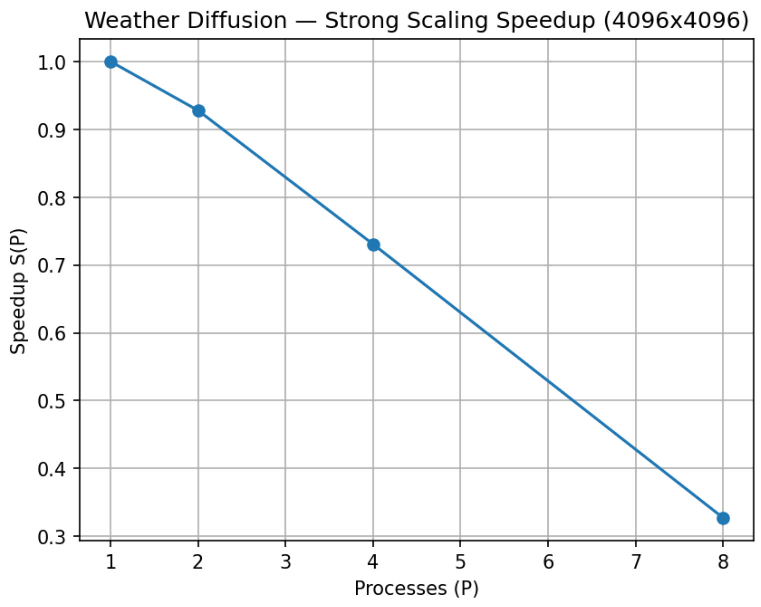




Strong Scaling – Speedup S(P)

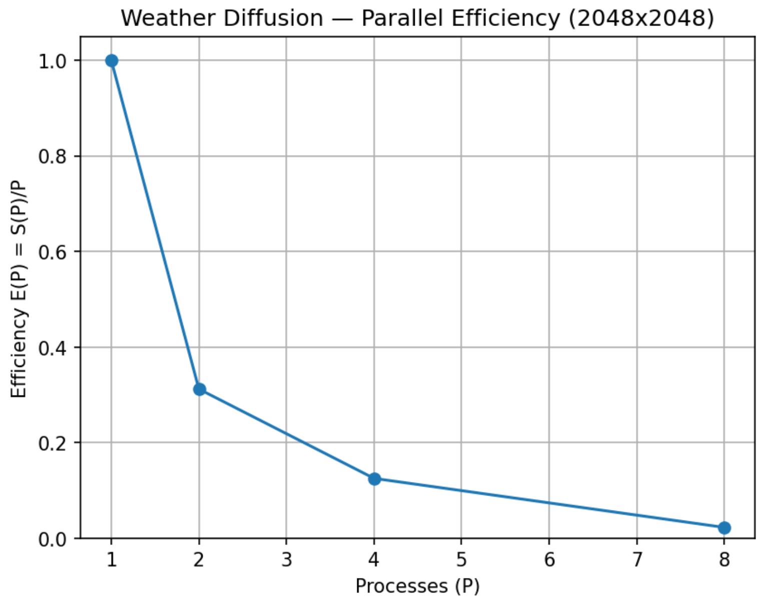
- Speedup relative to sequential baseline for both grid sizes.

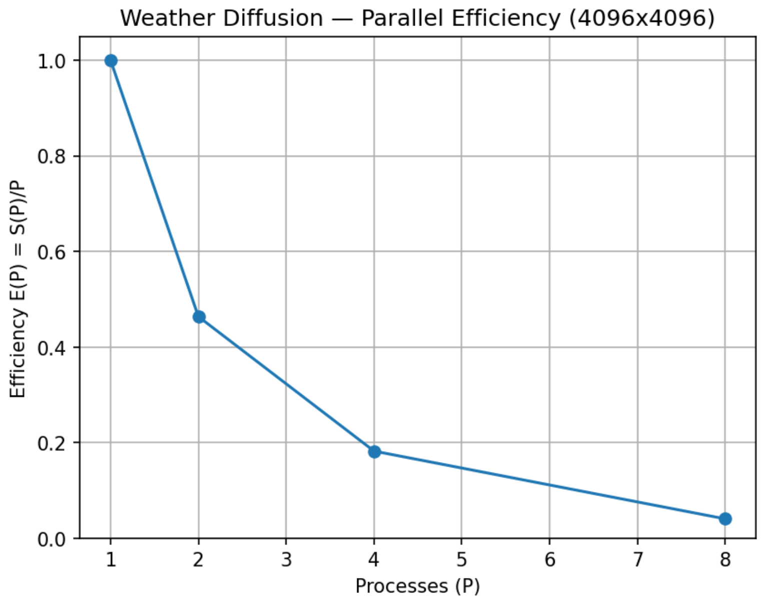




Parallel Efficiency $E(P) = S(P)/P$

- Efficiency drops sharply as we increase P, especially for the smaller grid.

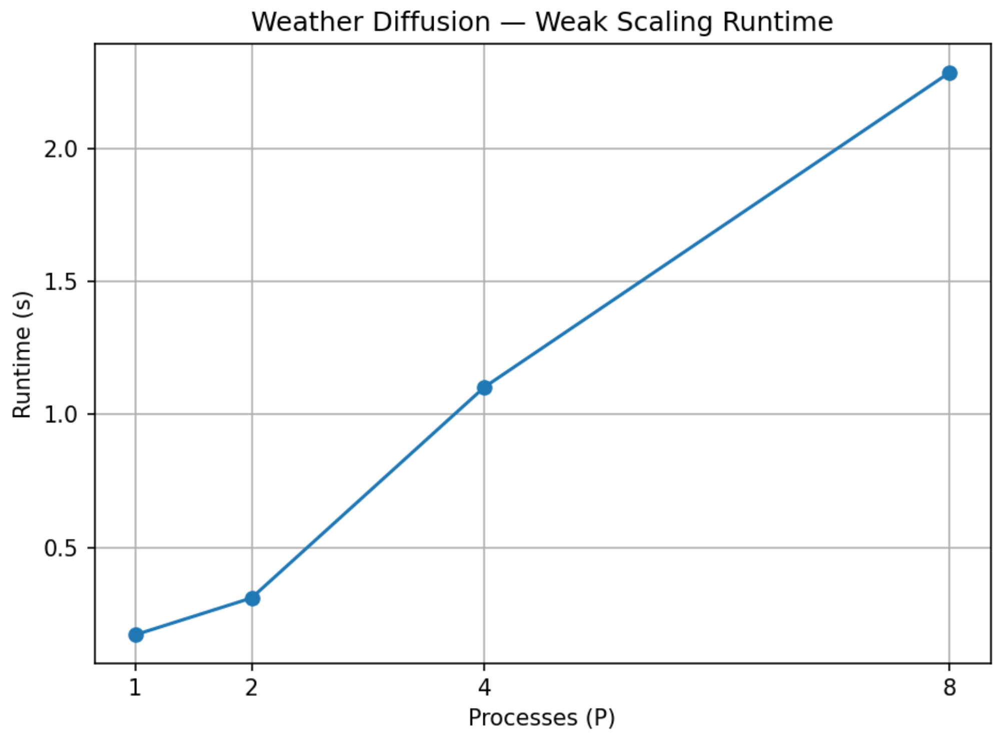




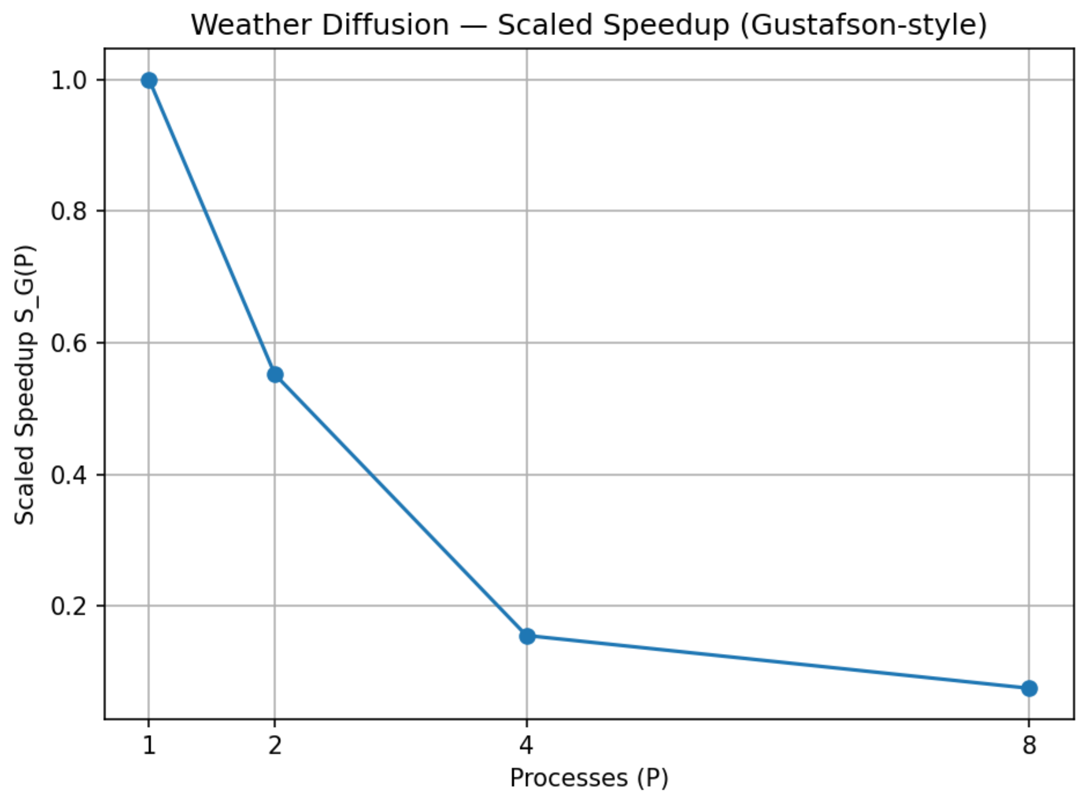
University at Buffalo The State University of New York

Weak Scaling and Gustafson's Perspective

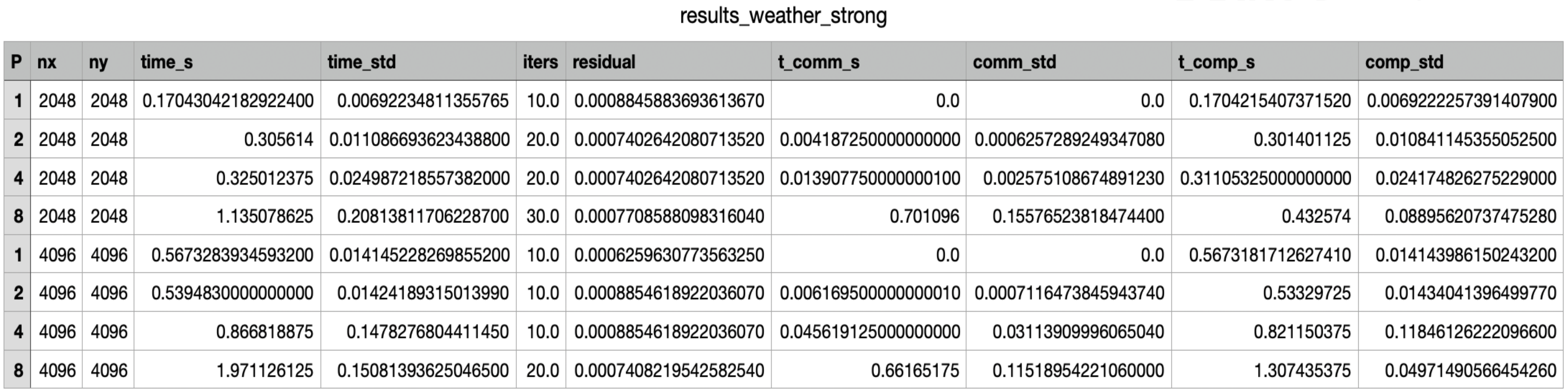
- Keep local grid size ≈2048 × 2048 cells per rank.
- Increase total problem size as we increase P.
- Ideal weak scaling: runtime ~ constant, scaled speedup ≈ P.
- Observed: runtime increases; scaled speedup stays sub-linear.
- Reason: shared memory bandwidth and MPI overhead on single node.



Weak Scaled Speedup



Sample Strong-Scaling Results



results_weather_strong

P	nx	ny	time_s	time_std	iters	residual	t_comm_s	comm_std	t_comp_s	comp_std
1	2048	2048	0.17043042182922400	0.00692234811355765	10.0	0.0008845883693613670	0.0	0.0	0.1704215407371520	0.0069222257391407900
2	2048	2048	0.305614	0.011086693623438800	20.0	0.0007402642080713520	0.004187250000000000	0.0006257289249347080	0.301401125	0.010841145355052500
4	2048	2048	0.325012375	0.024987218557382000	20.0	0.0007402642080713520	0.013907750000000100	0.002575108674891230	0.311053250000000000	0.024174826275229000
8	2048	2048	1.135078625	0.20813811706228700	30.0	0.0007708588098316040	0.701096	0.15576523818474400	0.432574	0.08895620737475280
1	4096	4096	0.5673283934593200	0.014145228269855200	10.0	0.0006259630773563250	0.0	0.0	0.5673181712627410	0.014143986150243200
2	4096	4096	0.5394830000000000	0.01424189315013990	10.0	0.0008854618922036070	0.006169500000000010	0.0007116473845943740	0.53329725	0.01434041396499770
4	4096	4096	0.866818875	0.1478276804411450	10.0	0.0008854618922036070	0.045619125000000000	0.03113909996065040	0.821150375	0.11846126222096600
8	4096	4096	1.971126125	0.15081393625046500	20.0	0.0007408219542582540	0.66165175	0.11518954221060000	1.307435375	0.04971490566454260

Conclusions and Key Takeaways

- MPI Jacobi solver is numerically correct and stable on large grids.
- Strong scaling effective only up to a modest number of processes.
- Communication and shared-memory bandwidth dominate at high P.
- Weak scaling is limited by hardware rather than algorithmic parallelism.
- Results align well with Amdahl’s and Gustafson’s theoretical predictions.

Future Work

- Explore hybrid MPI + OpenMP to reduce communication volume.
- Implement a GPU version of the stencil for higher throughput.
- Port the solver to C/C++ or Cython for lower interpreter overhead.
- Run on a multi-node HPC cluster with high-speed interconnect.