UNIVERSITÀ
DEGLI STUDI
DI TRIESTE

HPC-The Setencil Method

Scaling Analysis on Leonardo Supercomputer

Elena Ruiz de la Cuesta Castaño

October 2025

University of Trieste – HPC 2024/2025

Outline

1. Introduction
2. Methodology
3. Implementation
4. Results
5. Discussion & Conclusion

Introduction

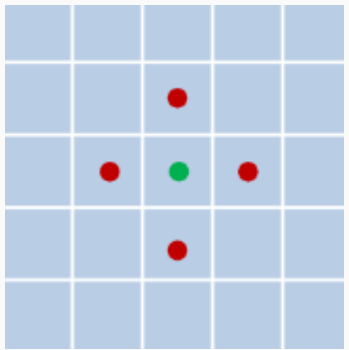
The Problem

Heat Equation in 2D

Two-dimensional heat diffusion problem.

$$\frac{\partial E}{\partial t} = \alpha \left(\frac{\partial^2 E}{\partial x^2} + \frac{\partial^2 E}{\partial y^2} \right)$$

- The temperature at each grid point $x_{ij}(t)$ depends on its own value at the previous time step, $x_{ij}(t-1)$, and on its four neighbors.
- This local interaction corresponds to the classical **5-point stencil** scheme.



Motivation and Objective

Motivation

For large grids, the computational cost of the stencil operation becomes significant $\rightarrow$ **Parallelization**.

Project Goal

Implement a **parallel 5-point stencil computation** using a hybrid approach with:

- **MPI**
- **OpenMP**

Evaluate the **scalability** on the Leonardo supercomputer:

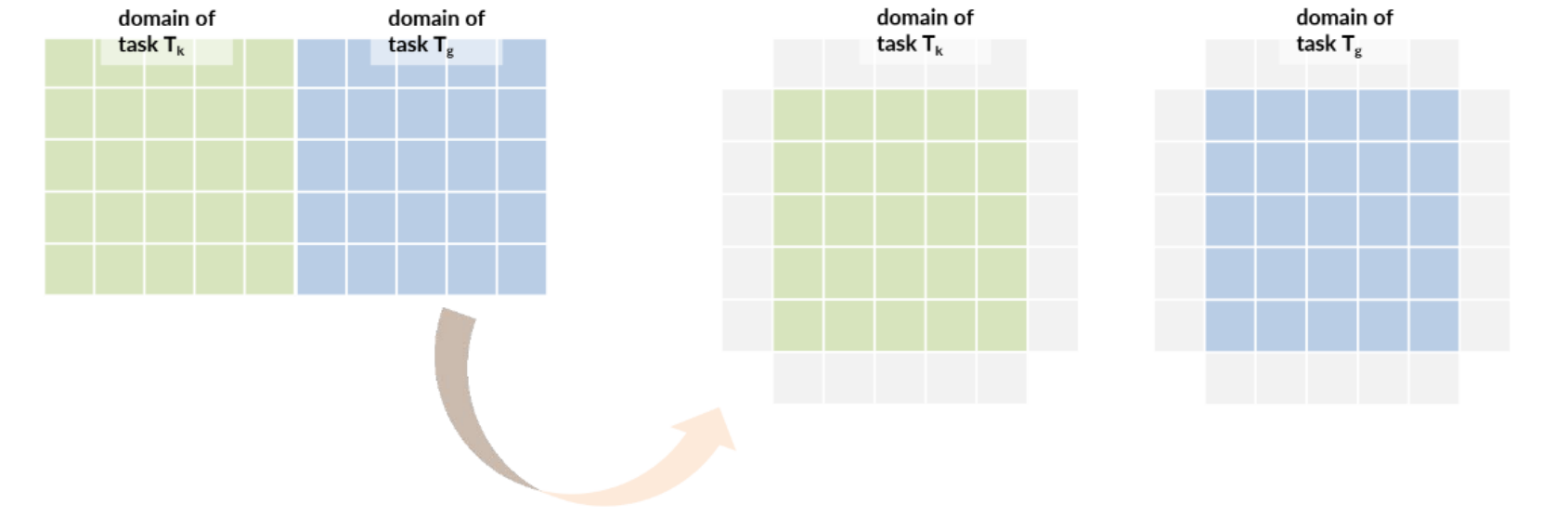
- Thread scaling
- Strong scaling
- Weak scaling

Methodology

Parallelization Strategy

Domain Decomposition

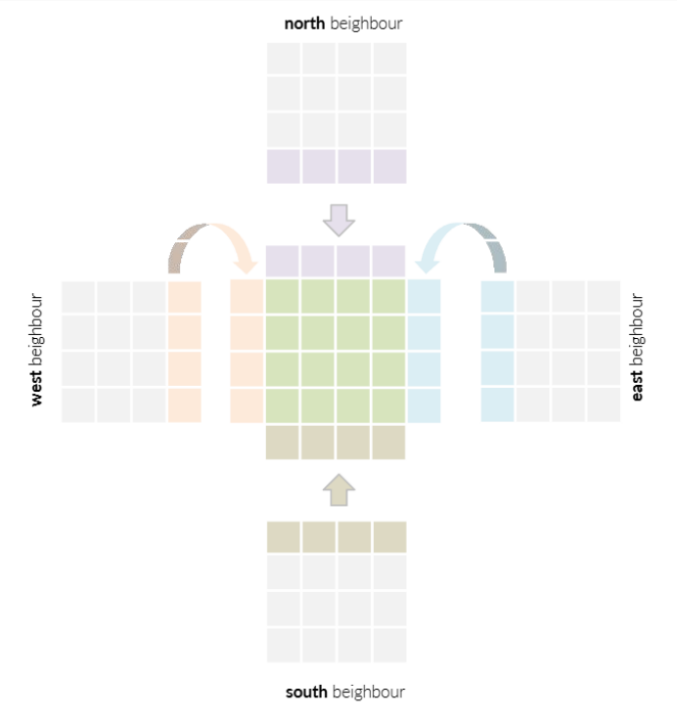
- The grid is divided into subdomains, each handled by one MPI task.
- Each task has **halo cells** (ghost layers) to store neighbor borders.



Parallelization Strategy

Memory Allocation

- **North/South:** receiving buffers point to halo rows, sending buffers point to the first/last rows of the local subdomain.
- **East/West:** separate allocated buffers; data must be copied before and after communication.



Implementation

Distributed-memory parallelism (MPI)

Grid Decomposition

Form-Factor Heuristic:

- Global grid split among MPI tasks.
- Choose $N_x \times N_y$ to make subdomains as square as possible $\rightarrow$ minimize communication cost.

Distributed-memory parallelism (MPI)

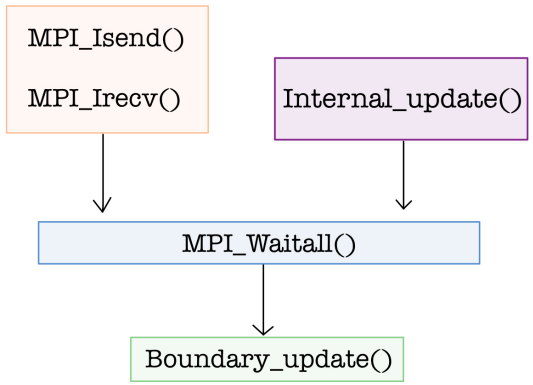
Grid Decomposition

Form-Factor Heuristic:

- Global grid split among MPI tasks.
- Choose $N_x \times N_y$ to make subdomains as square as possible $\rightarrow$ minimize communication cost.

Overlap Optimization

- Initiate non-blocking communication: `MPI_Isend / MPI_Irecv`
- Compute internal points (while halo data is in transit): `Internal_update()`
- Ensure all halo communications have completed: `MPI_Waitall()`
- Halo data is avaliable $\rightarrow$ update border points: `Boundary_update()`



Shared-memory parallelism (OpenMP)

- Within each MPI subdomain, OpenMP parallelizes the main update loops using **#pragma omp parallel for** $\rightarrow$ multiple threads work on different parts of the local grid at the same time.
- **schedule(static)** $\rightarrow$ same amount of work.
- **reduction(+:totenergy)** $\rightarrow$ safely accumulates total energy and avoids data races.

Compilation & Build-time Optimizations

- Compilation managed through a **Makefile** for consistency and easy rebuilds.

- **Compiler Flags:** `-O3 -march=native -fopenmp -Wall`
  - **-O3**: Enables aggressive compiler optimizations.
  - **-march=native**: Generates code optimized for the host CPU architecture, using all available instruction sets.
  - **-fopenmp**: Enables OpenMP support for parallel regions.
  - **-Wall**: Shows warnings to detect potential issues.

Runtime Measurements & Debug Flags

Measured Timings:

- `MPI_Wtime()` used to measure:
  - `comp_time` → computation phases (`Internal_update()`, `Boundary_update()`).
  - `comm_time` → communication phases (halo exchange: `MPI_Isend/Irecv` + `MPI_Waitall`).
  - `total_time` → full iteration loop wall-clock time.
- Reduced timings across ranks with `MPI_Reduce()` for global maxima → reflects the slowest process.

Runtime Measurements & Debug Flags

Measured Timings:

- `MPI_Wtime()` used to measure:
  - `comp_time` → computation phases (`Internal_update()`, `Boundary_update()`).
  - `comm_time` → communication phases (halo exchange: `MPI_Isend/Irecv` + `MPI_Waitall`).
  - `total_time` → full iteration loop wall-clock time.
- Reduced timings across ranks with `MPI_Reduce()` for global maxima → reflects the slowest process.

Debug / Runtime Flags:

- `-v <level>` → verbosity: prints local planes per rank (debug only).
- `-o 1` → outputs energy statistics per step (optional).

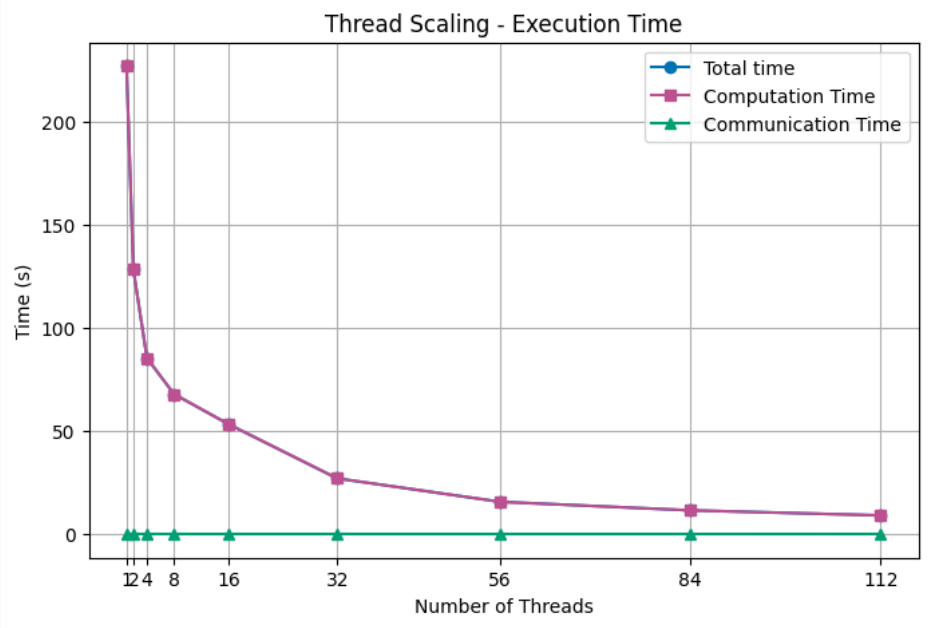
Results

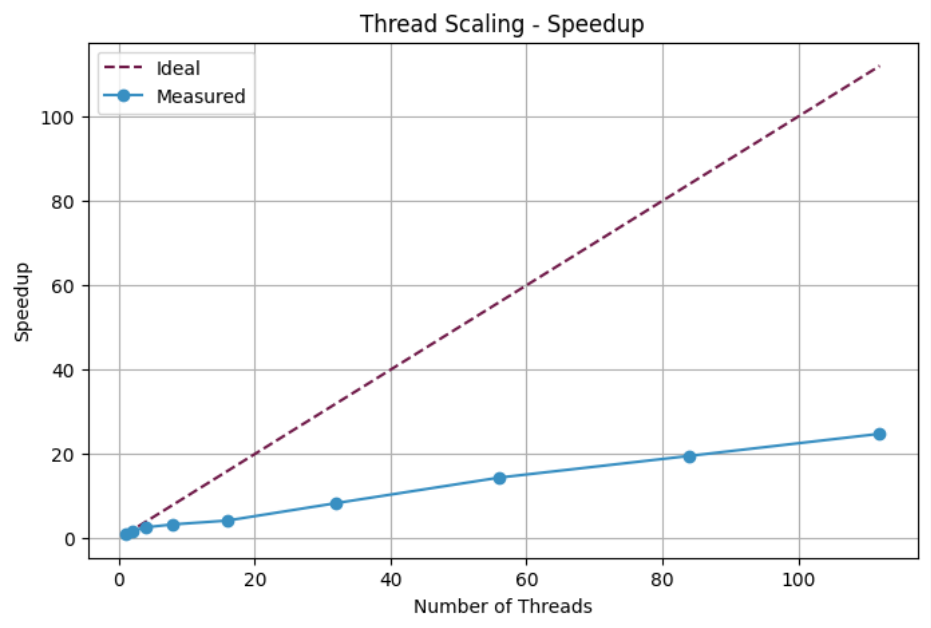
Scaling Tests Overview

1. **OpenMP Thread Scaling**
   - Single MPI task
   - Number of threads varied (1, 2, 4, 8, 16, 32, 56, 84, 112).
2. **Weak Scaling**
   - Fixed workload per node.
   - Increase both problem size and number of nodes (1, 2, 4, 8, 16) proportionally.
3. **Strong Scaling**
   - Fixed problem size.
   - Number of nodes increased (1, 2, 4, 8, 16).

OpenMP Thread Scaling

Single MPI task, varying number of threads.



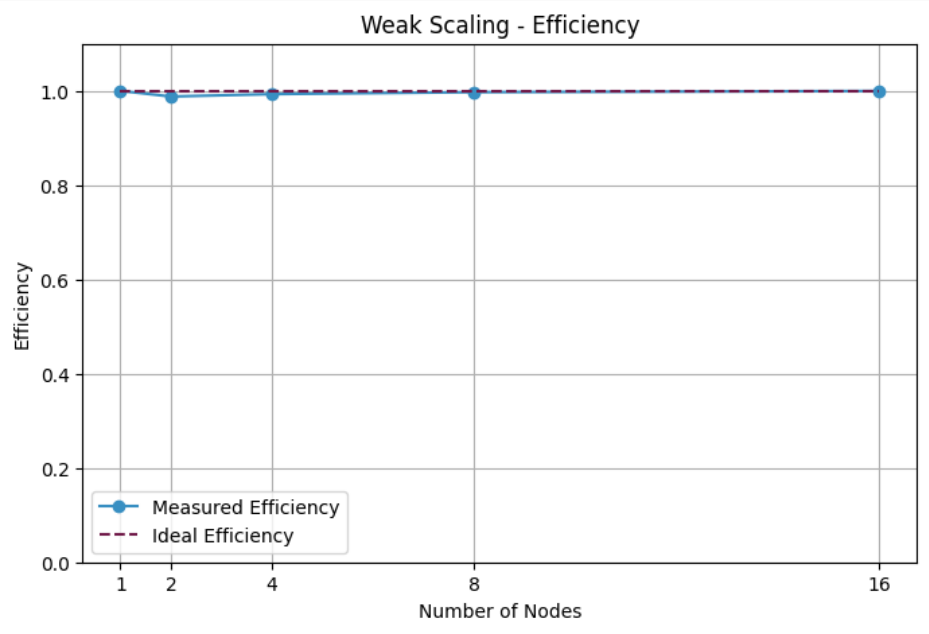


- Execution time decreases as the number of threads increases.
- Performance saturates due to memory bandwidth limits and thread overhead.

Next tests run with 14 threads and 8 MPI tasks per node.

MPI Weak Scaling

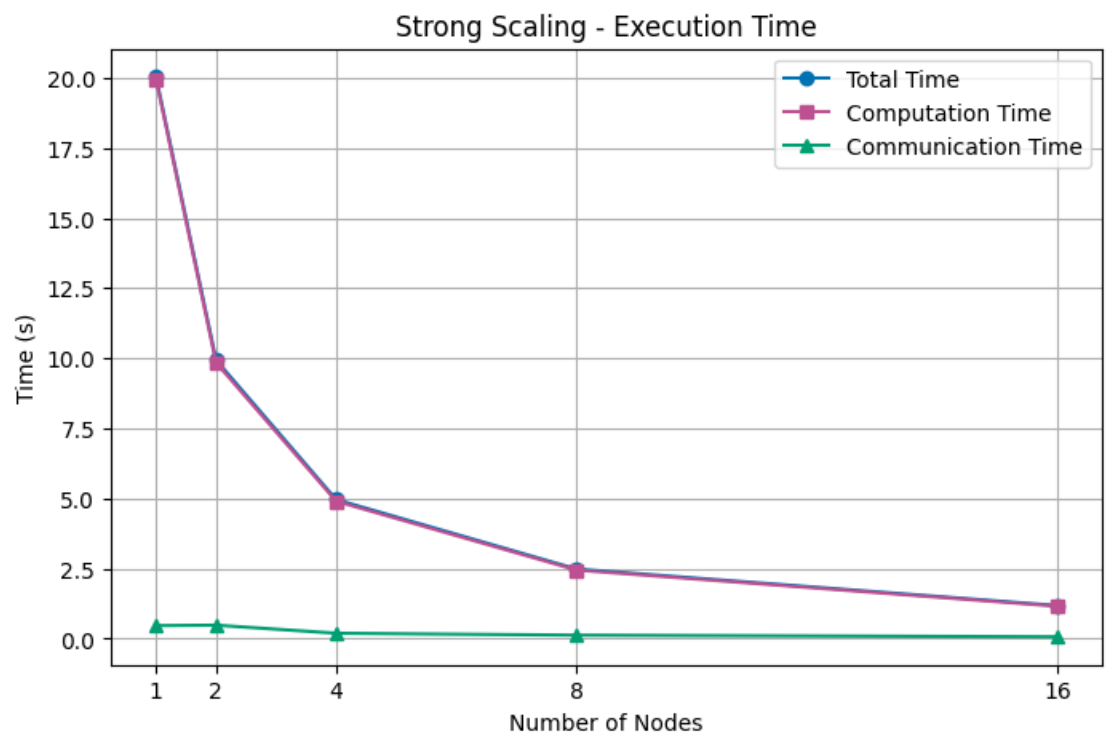
Fixed workload per node; problem size grows proportionally with number of nodes.



- Ideally, execution time should stay constant, and in the plot, the curve is almost flat.
- Efficiency remains close to 1, indicating excellent weak scaling performance.

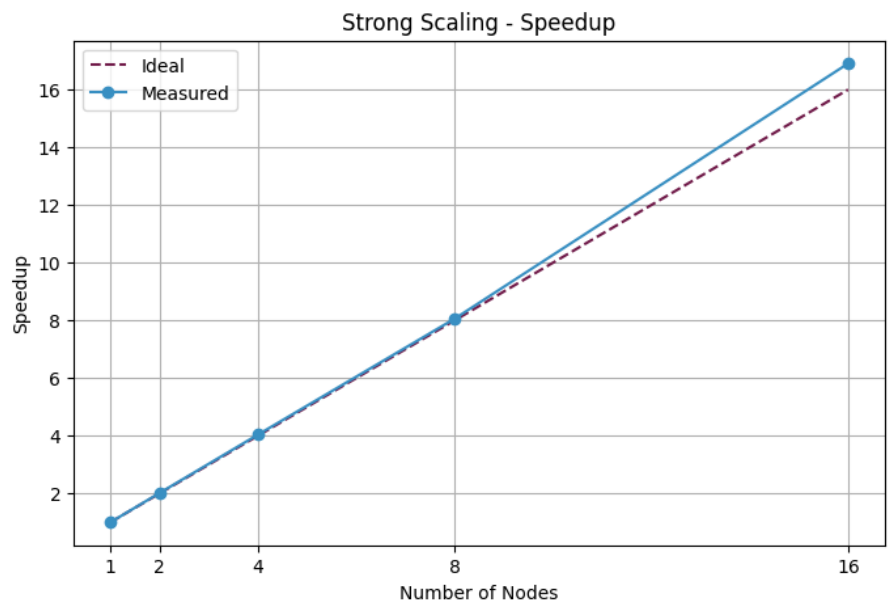
MPI Strong Scaling - Grid 15000x15000

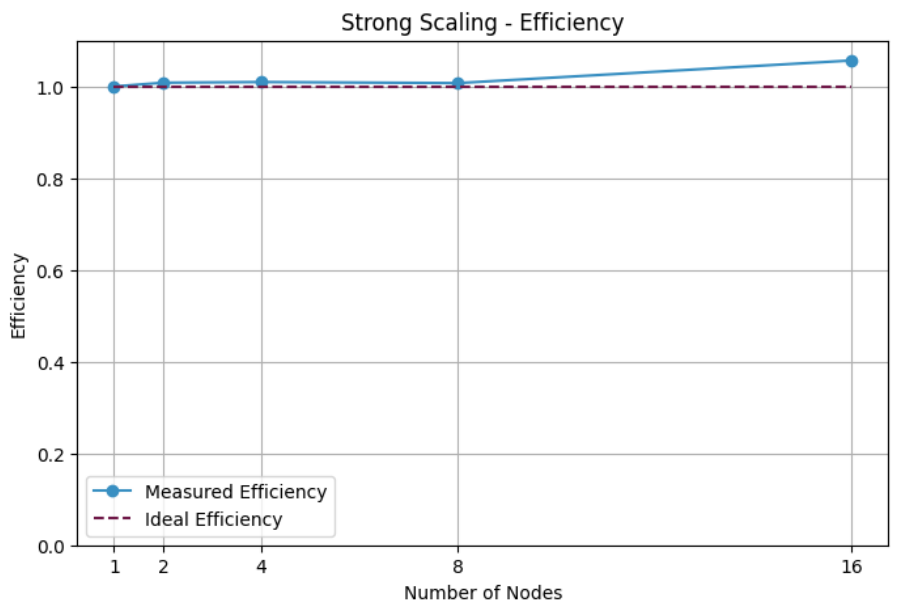
Fixed total problem size, increasing the number of nodes. Goal: measure how execution time decreases with more resources.



- Execution time decreases as more nodes are used.

MPI Strong Scaling - Grid 15000x15000





- A **super-linear speedup** is observed: the measured performance exceeds the ideal line at 16 nodes.
- Consequently, the **efficiency surpasses 1.0 (100%)** in the same region.
- This indicates an additional performance gain beyond simple parallelization.

Speedup Analysis: Explaining Super-Linearity

Hypothesis: A Memory Hierarchy Effect

The key is the relationship between the problem size per node and the cache capacity of each node.

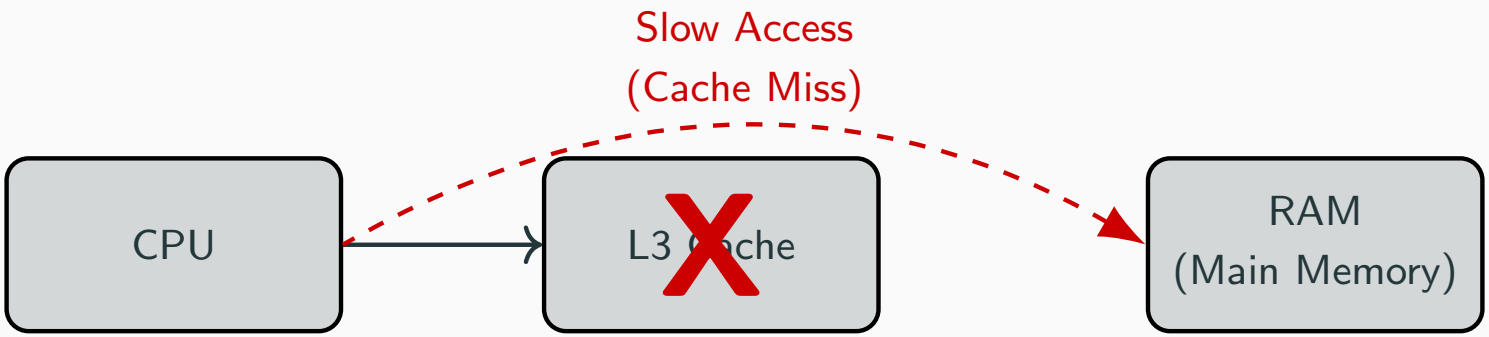
- **Step 1: Calculate the total problem memory size**
  For a 15000x15000 grid, with two copies (old & new plane) and 'doubles' (8 bytes), we need: *15000 x 15000 x 8 bytes/point x 2 grids* $\approx$ 3.6 GB, which in binary units is $\approx$ **3.35 GiB**.

- **Step 2: Determine the cache capacity per node**
  A Leonardo node has 2 sockets, each with 105 MiB of L3 cache: *2 sockets x 105 MiB/socket* = **210 MiB of L3 Cache per Node**

- **Step 3: Compare the memory per node to the cache**
  At 8 nodes: 3.35 GiB / 8 $\approx$ 429 MiB ($>>$ 210 MiB of cache)
  At 16 nodes: 3.35 GiB / 16 $\approx$ 215 MiB ($\approx$ 210 MiB of cache)

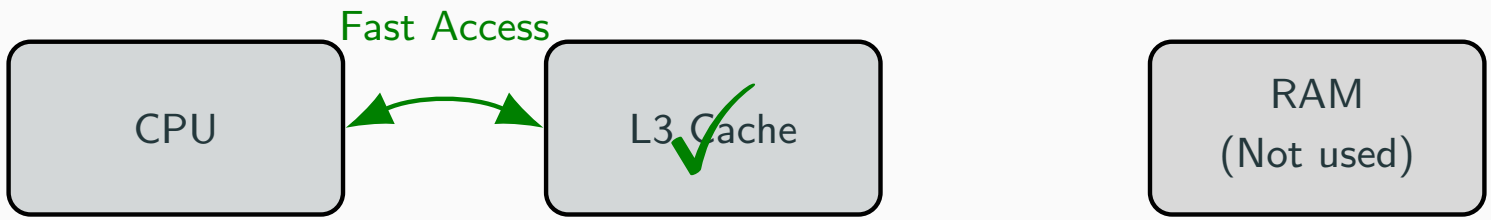
The Impact: "Memory-Bound" Regime (1-8 Nodes)

- The grid patch per node is **larger** than the L3 cache.
- The CPU must constantly fetch data from the **slow main RAM**.
- This costly "cache miss" limits the overall performance.

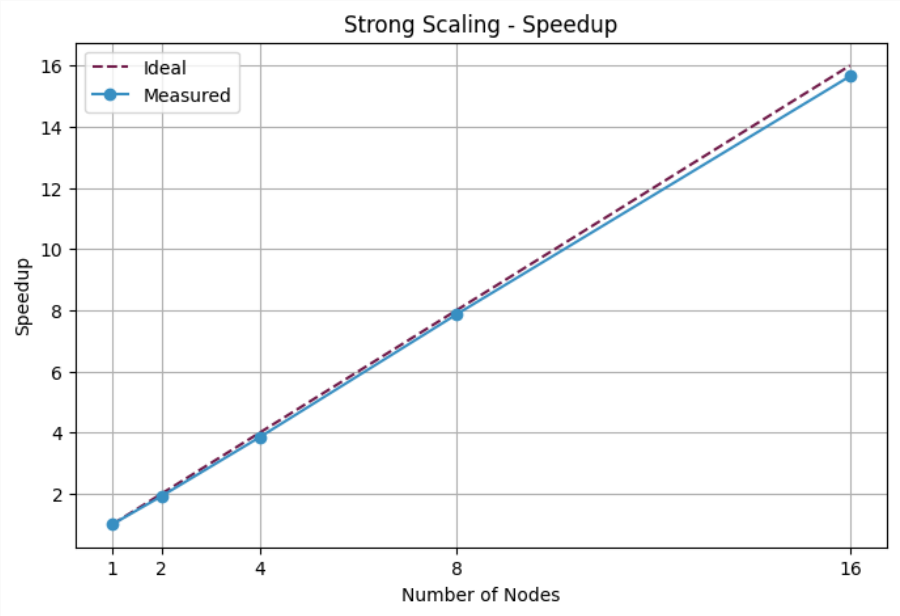


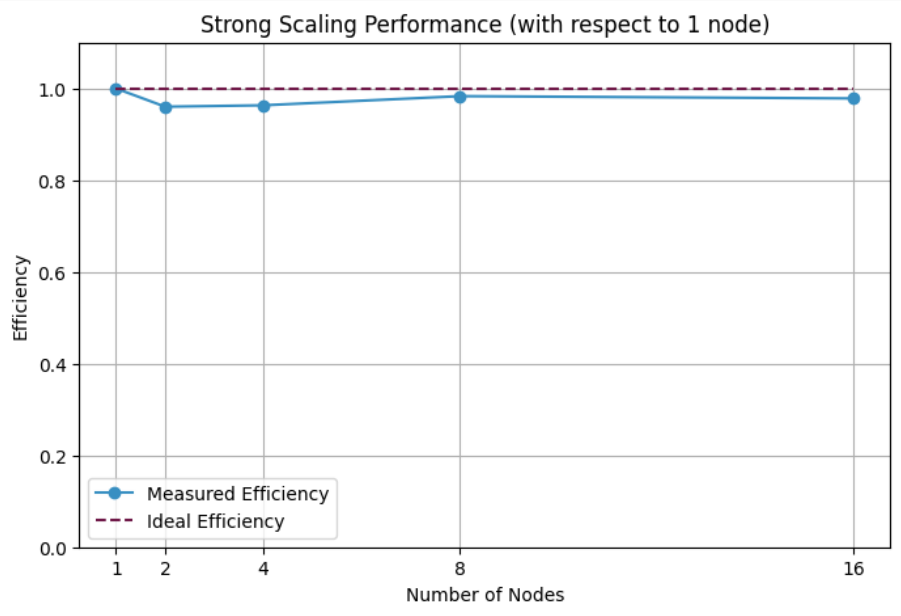
The Impact: "Cache-Bound" Regime (16 Nodes)

- The grid patch per node **fits inside** the L3 cache.
- The CPU finds almost all data in the **extremely fast L3 cache**.
- This "cache hit" accelerates performance, leading to the superlinear speedup.



Strong Scaling - Grid 30000x30000





- With the much larger grid, the **super-linear speedup vanishes**. Scaling is now sub-linear.
- **Reason: The problem is too large to fit into the L3 cache**, even with 16 nodes.
- The system remains in the **"Memory-Bound" regime** for all node counts, confirming our previous analysis.

Discussion & Conclusion

Discussion & Conclusions

Achievements

- The integration of **MPI + OpenMP** efficiently exploits both inter-node communication and intra-node parallelism.
- The code demonstrated excellent performance in both scaling tests: **strong and weak scaling**, confirming its effectiveness for large-scale workloads.

Discussion & Conclusions

Achievements

- The integration of **MPI + OpenMP** efficiently exploits both inter-node communication and intra-node parallelism.
- The code demonstrated excellent performance in both scaling tests: **strong and weak scaling**, confirming its effectiveness for large-scale workloads.

Possible Improvements

- Exploit the **first-touch memory policy** to ensure faster memory access.
- Use **MPI derived datatypes** to simplify east/west communications and reduce manual buffer handling.

Questions?

Thank you!