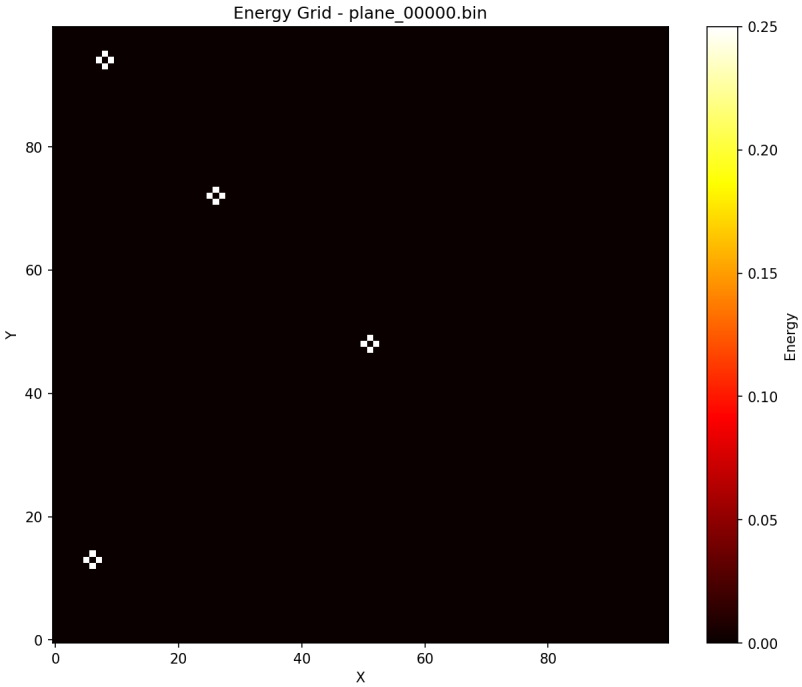
Distributed Stencil Performance Analysis

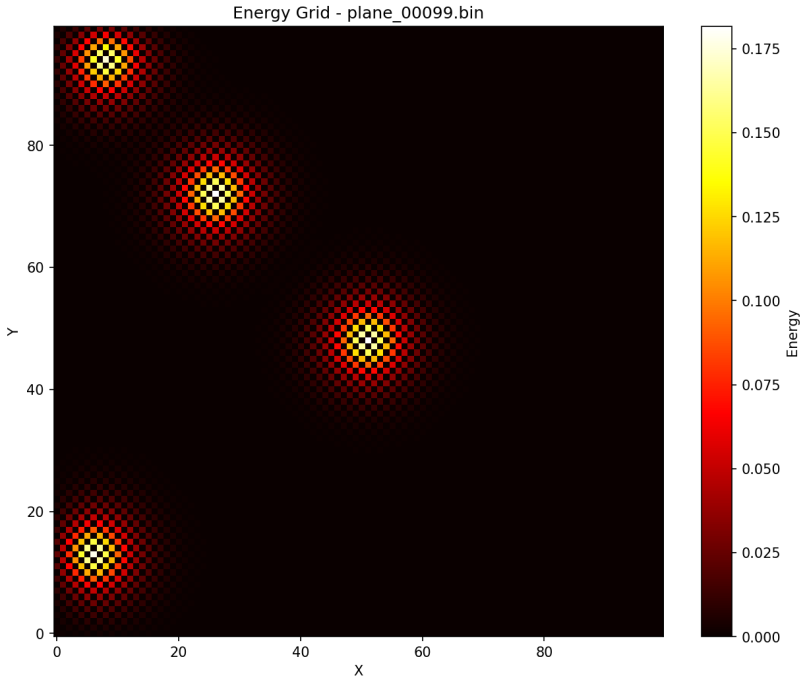
Christian Faccio

High Performance Computing course

October 10, 2025

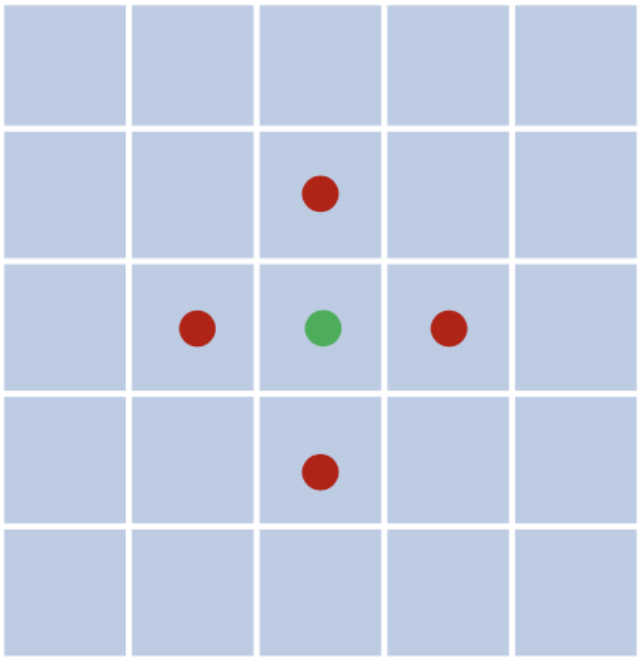
Energy distribution in a 2D domain





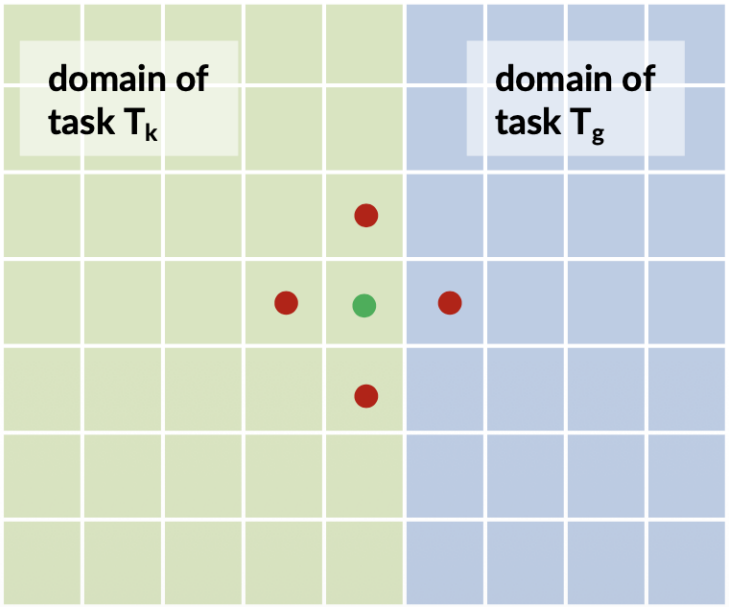
Serial Code

1. **Initialization**
   - Set default values
   - (opt) Read value from cmd line
   - Allocate memory
   - Initialize sources
2. **Main loop**
   - (opt) Inject energy
   - Update planes
   - (opt) Output energy stats
   - Swap planes
3. **Output energy stats**
4. **Free memory**



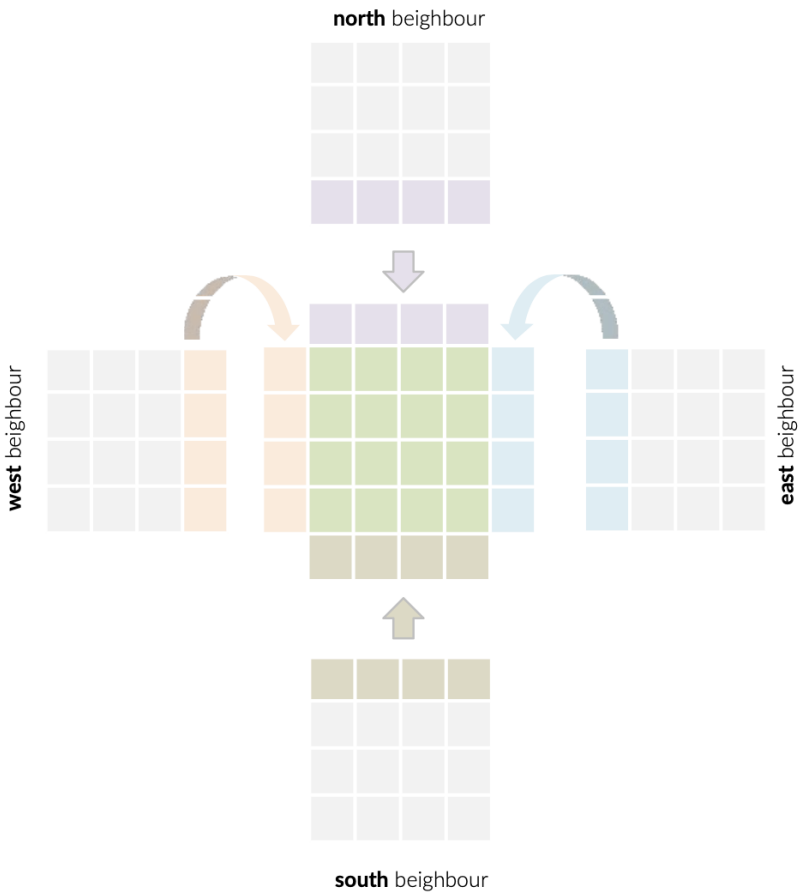
Parallel Code

1. **Initialization**
   - Set default values
   - (opt) Read value from cmd line
   - *Domain decomposition*
   - Allocate memory
   - Initialize sources
2. **Main loop**
   - (opt) Inject energy
   - *Exchange halos (MPI)*
   - *Update planes (OpenMP)*
   - (opt) Output energy stats
   - Swap planes
3. **Output energy stats**
4. **Free memory**



Halo exchange

- Each process manages a sub-domain of the global domain
- Necessity to use values from neighboring sub-domains
- **Solution**: exchange of *halo* regions (ghost cells)



MPI

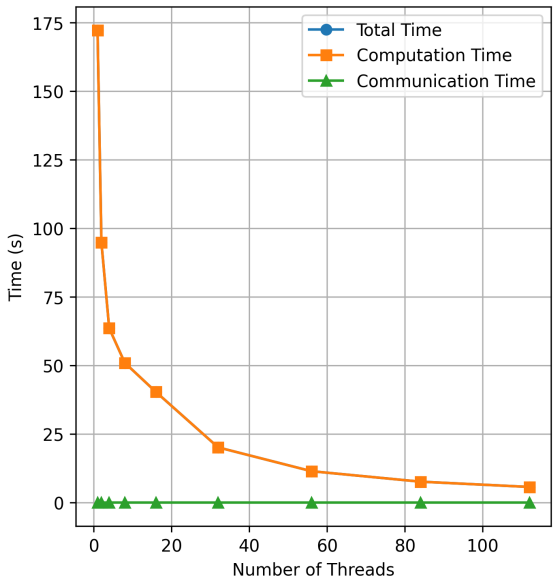
- Separate **context** -> `STENCIL_WORLD`
- Simple yet effective **domain decomposition** between processes
- `MPI_Bcast` to share tasks with sources
- `MPI_Isend` and `MPI_Irecv` to exchange halos
- `MPI_Reduce` to gather energy statistics

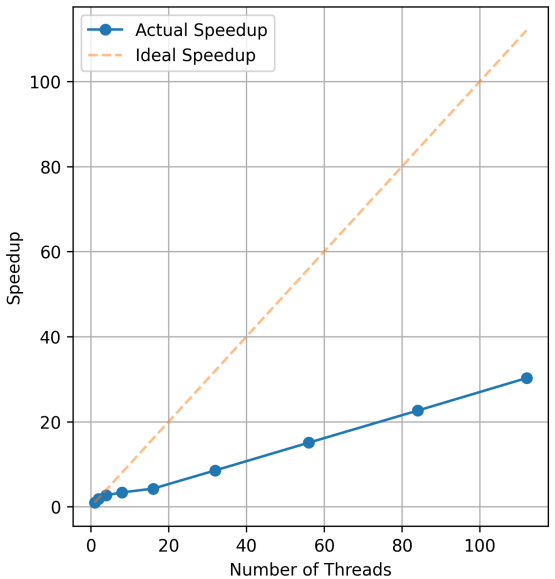
OpenMP

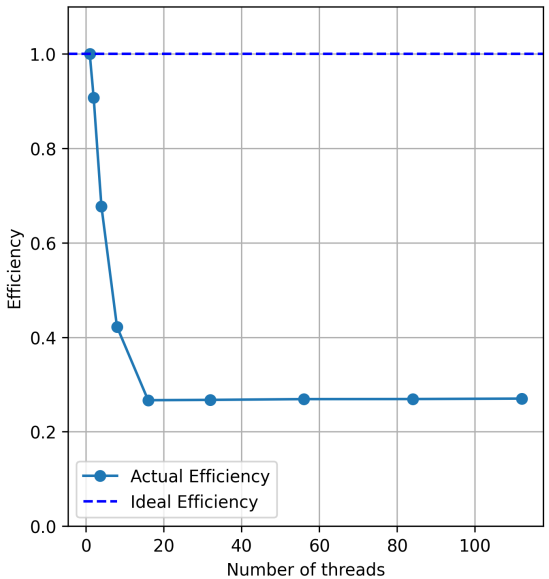
- `#pragma omp parallel for` to parallelize the loops that iterate through the 2D domain
- `schedule(static)` to divide iterations in contiguous chunks
- `reduction(+:totenergy)` to avoid data races when summing energy
- `#pragma GCC unroll 4` to unroll inner loop
- `First touch` policy for memory allocation
- `export OMP_PLACES=cores`
- `export OMP_PROC_BIND=close`

Threads Scaling

- 1 MPI process
- `for threads in 1 2 4 8 16 32 56 84 112; do`
- `-x 16000 -y 16000 -n 500 -f 50 -e 4 -E 1.0 -p 0`







Scalability

Scalability is the capability of the code to efficiently use different amount of computational resources when

- **Weak scaling**: the problem size per resource is constant;
- **Strong scaling**: the total problem size is constant.

We can measure it using the **speedup**:

$$S = \frac{T_{ref}}{T}$$

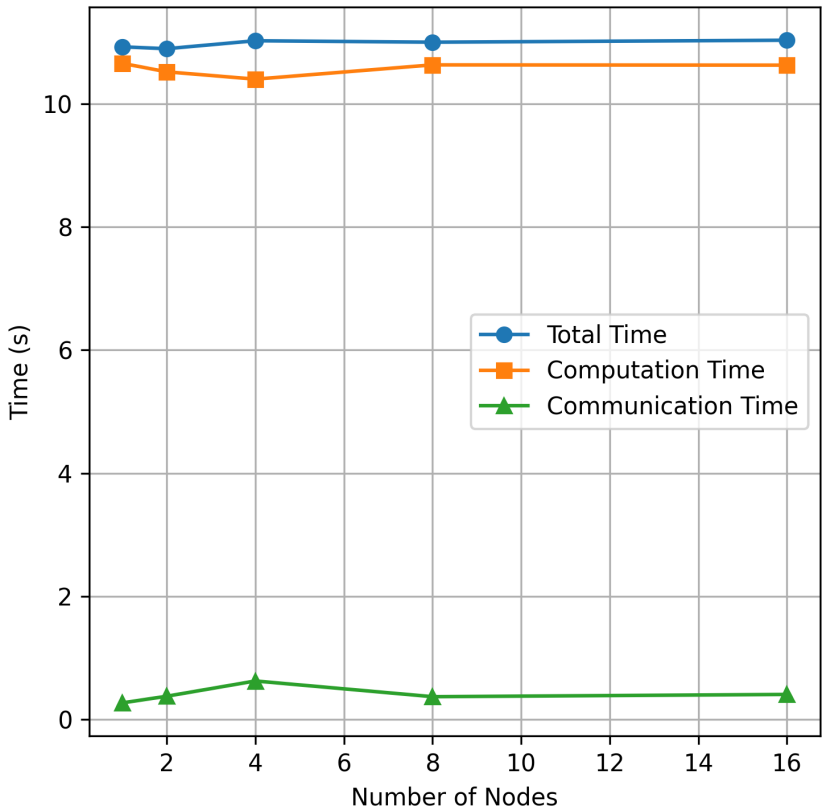
where T_{ref} is the execution time of a reference case and T is the execution time of the case we want to analyze.
The **efficiency** is used to measure how the speedup scales with the number of resources:

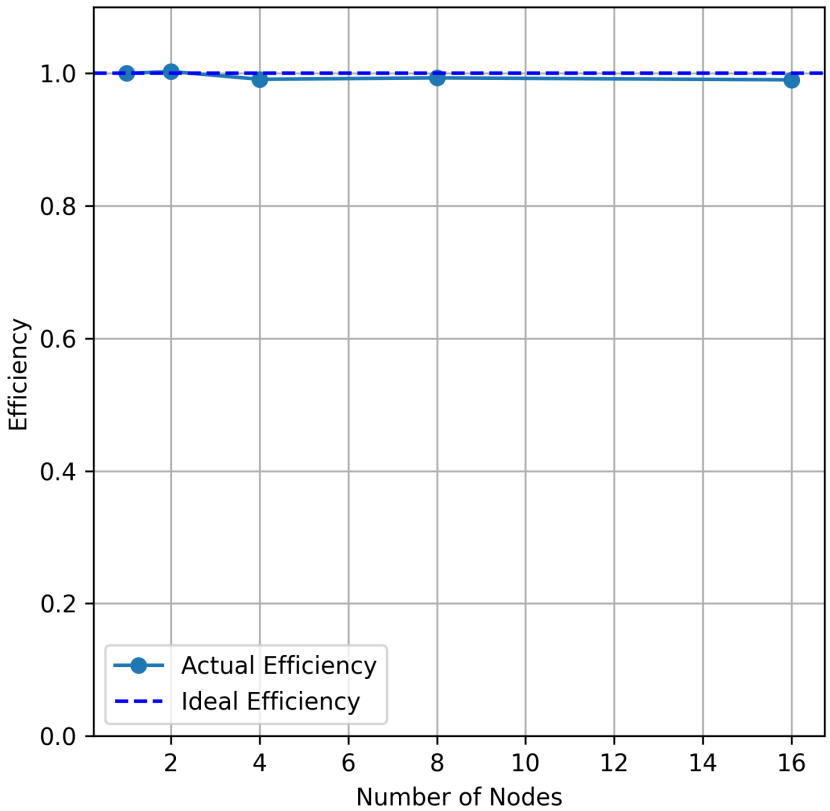
$$E = \frac{S}{P}$$

where P is the number of resources (e.g. processes).

Weak Scaling

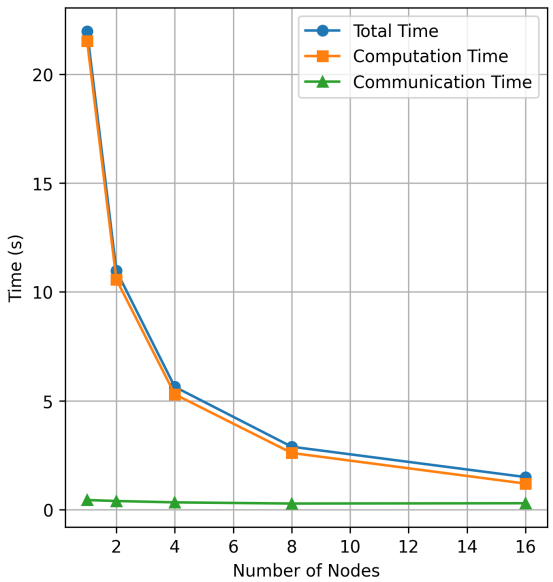
- 8 MPI processes per node
- 14 OpenMP threads per process
- `LOCAL_SIZE=4000`
- `for nodes in 1 2 4 8 16; do`

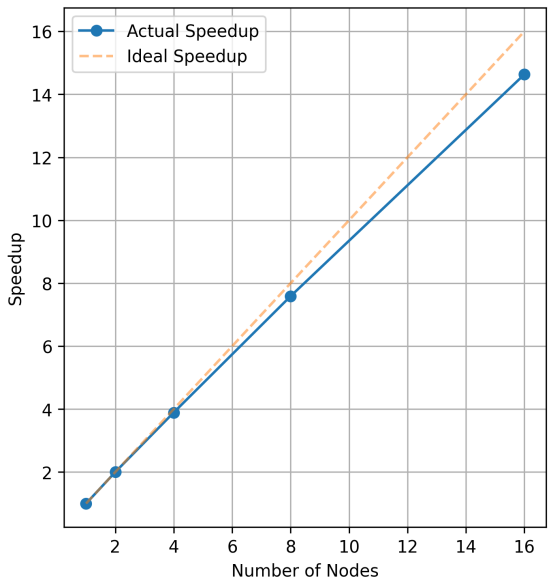


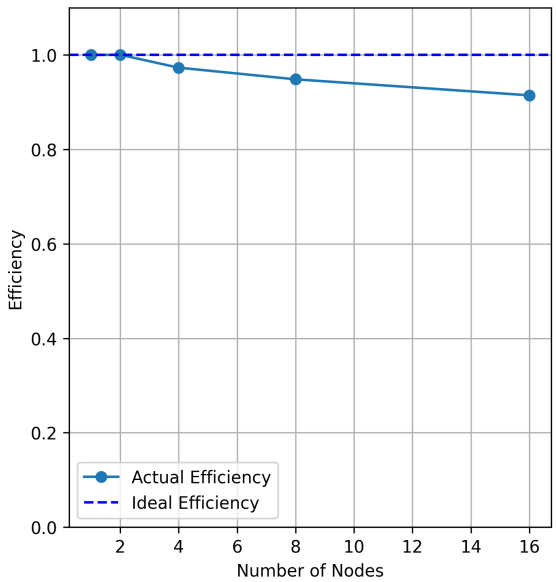


Strong Scaling

- 8 MPI processes per node
- 14 OpenMP threads per process
- `GRID_SIZE=16000`
- `for nodes in 1 2 4 8 16; do`







Conclusions

We can comment on the results now:

- **Good** scaling properties;
- Further optimizations possible;
- Higher communication time with more processes in strong scaling;

A good future work could be the implementation of **MPI derived data types**.