Exercise 1: MPI-Parallelization of a Jacobi Solver

- **Handout** on Thursday 14.03.2024, 3.30pm.
- **Handin** until Thursday 18.04.2024, 2pm, group submission via TUWEL.
- Include the name of all group members in your documents.
- Do not include binary executables **but** build instructions (for GNU/Linux) and a Makefile.
- Include any scripts you used to generate the plots from your recorded data.
- Submit your report (including plots/visualizations) as one `.pdf`-file per task.
- Submit everything as one `.zip`-file.

General information

- Use (and assume) the `double` precision floating point representation.
- Test your MPI-parallel implementation on your local machine before you benchmark on the cluster.
- Compare the results (i.e., residual and error) with your serial implementation to ensure a correct implementation.
- Use only a small number of Jacobi iterations when benchmarking the performance your code: convergence is **not** required during benchmarking.

MPI-Parallel Stencil-Based Jacobi Solver

In this excercise, your task is to parallelize a stencil-based Jacobi solver for Poisson's equation in a two-dimensional domain:

$$-(u_{xx} + u_{yy}) = f$$

Square domain:

$$\Omega = [0,1] \times [0,1]$$

Boundary conditions:

$$\begin{aligned} \frac{\partial u(x,0)}{\partial n} &= 0 \\ \frac{\partial u(x,1)}{\partial n} &= 0 \\ u(0,y) &= u_W \\ u(1,y) &= u_E \end{aligned}$$

The problem is discretized with a regular grid spacing $h_x = h_y = h = \frac{1}{N-1}$ using a central difference scheme for the second-order derivates (*5-point star-shaped stencil*) resulting in a sparsly populated LSE

$$A_h u_h = b_h$$

where A_h is the system matrix, b_h is the RHS and u_h is the solution to the LSE.

Domain Decomposition

In this exercise your main task is to decompose the finite-difference grid into domain regions such that multiple MPI-processes can independently perform an Jacobi iteration on each region. The decoupling of the regions is achieved by introducing a *ghost layer* of grid points which surrounds each region:

- The values in the ghost layer of a region are **not updated during an iteration**.
- Instead, **after an iteration is finished** the updated values for the ghost layer are received from the neighbouring regions, and the boundary layer is **sent to the neighouring regions** (see Figure below).

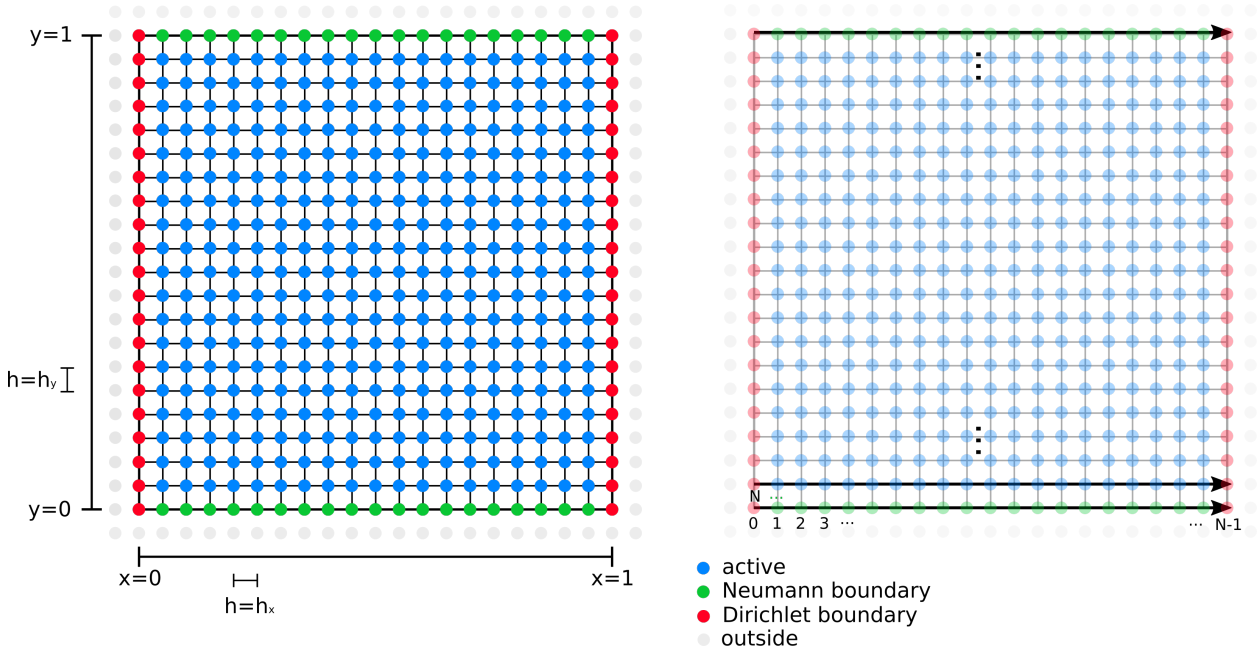

Figure 1: images/unitsquare.png

Task 1: Questions (3 points)

1. Describe the advantages/disadvantages of a two-dimensional decomposition (compared to a one-dimensional decomposition).
2. Discuss if the decomposition of the domain changes the order of computations performed during a single Jacobi iteration (i.e., if you expect a numerically identical result after each iteration, or not).
3. An extension to the ghost layer approach would be to use a wider layer (of more than one ghost cell). This allows to perform multiple independent iterations before a communication of the ghost layers has to happen. Comment in which situation (w.r.t. the available bandwidth or latency between MPI-processes) multiple independent iterations are potentially advantageous.
4. How big is the sum of all L2 caches for 2 nodes of the IUE-cluster link

Task 2: One-Dimensional Decomposition (4 points)

As a starting point for your implementation, you can

- use the source code distributed with this exercise in the folder `src`, or
- use your own serial implementation of the Jacobi solver from NSSCI/Exercise2 (supporting a Gaussian source region is optional)

Your MPI-implementation should be callable like this (identical to NSSCI/Exercise2, beside the new first argument `1D`):

```
./solverMPI 1D benchmarkX 250 100 0.0 1.0
```

The command line arguments are (in order):

- `1D`: decomposition mode, value values are `1D` (task2) and `2D` (task3)
- `benchmarkX`: a name (string) for the simulation (which has to be used as filename for the `.csv` output).
- `250`: resolution N (number of grid points along one axis of the domain).
- `100`: number of iterations the Jacobi solver should perform.
- `0.0`: value u_W for the Dirichlet boundary condition on the left (west) side.

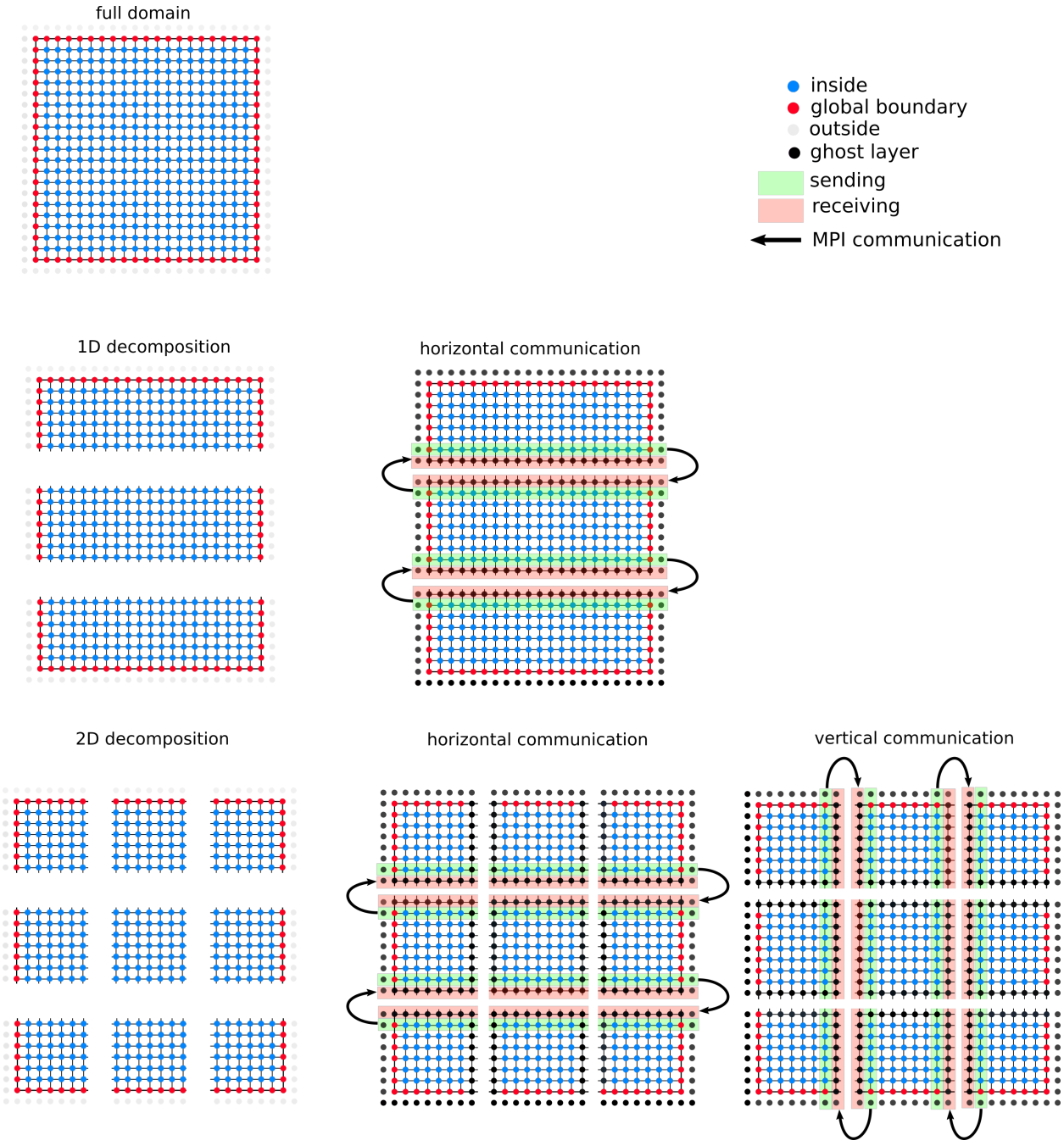

Figure 2: Decomposition

- `1.0`: value u_E for the Dirichlet boundary condition on the right (east) side.

Your task is to implement a one-dimensional decomposition using a ghost layer and MPI-communication to update the ghost layers. Create a program `solverMPI` which is callable like this:

```
# example call (on your machine, or on the cluster via a slurm script) using 4 MPI-processes
mpirun -np 4 ./solverMPI 1D benchmarkX 250 100 0.0 1.0
```

Further and more specifically, your program should

- use $\bar{u}_h = \mathbf{0}$ as initial approximation to u, and (after finishing all iterations)
- print the Euclidean $\| \cdot \|_2$ and Maximum ‘ $\| \cdot \|_\infty$ norm of the residual $\| A_h \bar{u}_h - b_h \|$ to the console,
- print the runtime for the total number of iterations of the Jacobi method to the console, and
- save the solution vector as two dimensional array using the `.csv`-file format; use the name of the simulation (first command line argument) as filename,
- **produce the same results as a serial run**.

Finally, benchmark the parallel performance of your program `solverMPI` using 1 node of the IUE-Cluster for the 4 different resultions $\{125, 250, 1000, 2000\}$ using between 1 and 40 MPI-processes. More specifically, you should

- create a plot of the parallel speedup and a plot of the parallel efficiency for each of the 4 different resolutions, and
- discuss your results in detail.

Notes on MPI:

- On your local machine, you can also launch MPI-runs using `mpirun`, e.g. for a Ubuntu system using OpenMPI

  ```
  sudo apt-get install build-essential
  sudo apt-get install openmpi-bin openmpi-common libopenmpi-dev
  mpic++ -std=c++17 -O3 -Wall -pedantic -march=native -ffast-math main.cpp -o solverMPI
  mpirun -np 2 ./solverMPI 1D benchmarkX 64 1000 0.0 1.0
  mpirun -np 8 --oversubscribe ./solverMPI 1D benchmarkX 64 1000 0.0 1.0
  ```

- The use of `MPI_Cart_create`, `MPI_Cart_coords`, and `MPI_ Cart_shift` for setup of the communication paths is recommended.
- Your implementation should work for any number of MPI-processes (e.g., 1,2,3,4,...) and also utilize this number of processes for the decomposition.
- If you compile for a benchmark run, the following compiler arguments are recommended: `-O3 -Wall -pedantic -march=native -ffast-math`

Task 3: Two-Dimensional Decomposition (3 points)

Extend your program from Task 2 by implementing support for two-dimensional decomposition using a ghost layer and MPI-communication to update the ghost layers.

```
# using 4 MPI-processes and a 1D decomposition
mpirun -np 4 ./solverMPI 1D benchmarkX 250 100 0.0 1.0
# using 4 MPI-processes and a 2D decomposition
mpirun -np 4 ./solverMPI 2D benchmarkX 250 100 0.0 1.0
```

- Your implementation should work for and also utilize any number of processes for the 2D decomposition.
  - If a the 2D composition is not possible with the supplied number of processes (i.e., a prime number), your program should resort to a 1D decomposition.
- Ensure a correct implementation by comparing your results to a serial run.
- Benmarking Task 3 on the cluster is **not** required.

Working with the IUE-Cluster

- Your login credentials (username/password) will be provided individually; you will be asked to change your initial password upon first login.
- You need to enable a "TU Wien VPN" connection.
- You can login to the cluser using `ssh` and your credentials.
- The cluster has a *login node* (the one you `ssh` to); all other nodes of the cluster are used to run the "jobs" you submit.
- The *login node* must only be used to compile your project and **never** to perform any benchmarks or MPI-runs (beside minimal lightweight tests of for the MPI configuration)
- You can transfer files and folders between your local machine and the cluster using `scp`
- You also need to load the modules you require in your job subsmission scripts (see examples provided in this repo).
- All nodes of the cluster operate on a shared file system (all your files on the cluster are also available when executing jobs).

Submitting jobs on the cluster

- Once you successfully compiled your program on the login node of the cluster, you are read to submit jobs.
- On our cluster job submissions are handled by the workload manager `SLURM` (a very common choice).
- A job essentially is a shell-script which contains a call of your executable (see examples in this repo).
- Additionally a job-script specifies its demands w.r.t. to resources (number of nodes, expected runtime, ...)
- After you submitted the job, it is up to the `SLURM` scheduler to queue it into the waiting list and to inform you once the job has completed.
- The "results" of your job are
  1. Potential output files which your program produced during execution.
  2. The recording of the stdout/stderr which was recorded while your job executed (e.g., `slurm-12345.out`)

Useful Resources

- link: SLURM Intro
- link: SLURM Intro(youtube playlist)
- link: OpenMPI Docs
- link: `scp`
- link: Environment Modules