Università
della
Svizzera
italiana

Institute of
Computing
CI

High-Performance Computing, Fall 2025
Lecturer: Dr. A. Eftekhari
Assistants: V. Karpenko, L. Migliari, D. Vega

Project 5
Parallel Nonlinear PDE using MPI and HPC with Python

Due date: See iCorsi submission

In this project, we will continue learning about parallel programming with MPI which was introduced in project 4. Particularly, we will use the ghost cells exchange between neighboring processes towards building an MPI parallel solver for Fisher's equation that we discussed and parallelized with OpenMP in project 3. Furthermore, we will extend this project with several tasks related to High-Performance Computing with Python. The Python programming language is very popular in scientific computing because of the benefits it offers for fast code development.
As usual, you find all the skeleton source codes for the project on the course iCorsi page. We highly recommend to review all the provided skeleton codes before starting any serious implementation of the project tasks.

1 Parallel Space Solution of a nonlinear PDE using MPI [60 points]

This sub-project discusses domain decomposition for an MPI parallel solver of a nonlinear PDE that we discussed in detail in project 3. In project 3, we have added OpenMP to the parallel space solution of a nonlinear PDE mini-application, so that we could use all cores on one compute node on the Rosa cluster. The goal of this exercise is now to utilize MPI (Message Passing Interface), enabling the use of multiple compute nodes. Unlike the serial and OpenMP versions, where a single process handles all the data, the MPI version distributes the computational domain across multiple processes (ranks). Each rank handles a sub-domain, allowing for *domain decomposition*. Each process can access only its sub-domain's data and not the data from other processes. For computations using the five-point finite-difference stencil, each process needs data from neighboring grid points. If these points lie on the boundary of a process's sub-domain, the necessary values must be obtained from adjacent processes. Therefore, before each iteration, all MPI processes exchange these boundary values — known as *ghost*, *guard* or *halo cells* — storing them in boundary buffers. This exchange ensures that each process has the necessary data to compute the next iteration.
You can find an initial incomplete version of the MPI code in the directory `mini_app`. The source code is almost equivalent to the serial/OpenMP version that you have already implemented in project 3. There are some comments below and in the code that will guide you through the implementation.

1.1 Initialize/finalize MPI and welcome message [5 Points]

Initialize and finalize the MPI environment in `main.cpp`. Replace the welcome message with one that (i) informs the user that the code is using MPI and (ii) indicates the number of processes:

```
[user@icslogin01 ~]$ srun --ntasks=4 --nodes=1 --time=00:05:00 --reservation=hpc-monday --pty bash -i
srun: job 13542 queued and waiting for resources
srun: job 13542 has been allocated resources
[user@icsnodeXX mini_app]$ module load openmpi
[user@icsnodeXX mini_app]$ make
[user@icsnodeXX mini_app]$ mpirun ./main 128 100 0.005
================================================================================
Welcome to mini-stencil!
version   :: C++ MPI
processes :: 4
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.005
```

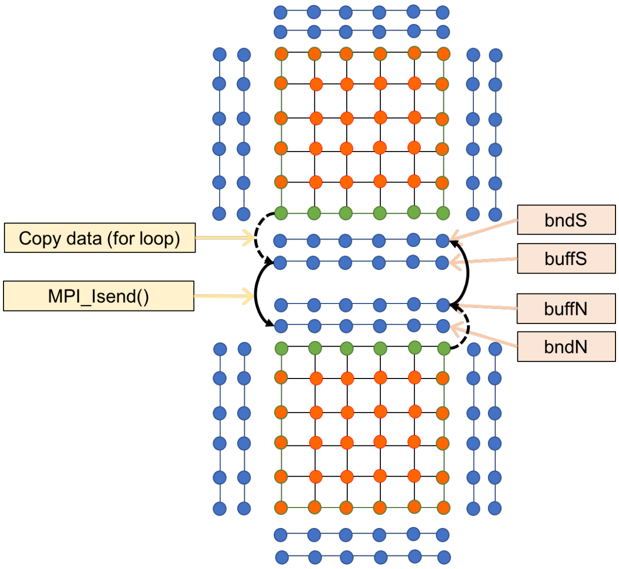

Figure 1: Ghost cell exchange: The bottom process copies the north row (green) into buffer `buffN`, and sends it to its north neighbor (top process). The top process receives the ghost cells from its south neighbor (bottom process) into `bndS`. Likewise, the top process copies the south row (green) into buffer `buffS` and sends it to its south neighbor (bottom process). The bottom process receives the ghost cells from its north neighbor (top process) into `bndN`.

```
iteration :: CG 300, Newton 50, tolerance 1e-06
================================================================================
--------------------------------------------------------------------------------
simulation took 0.0614173 seconds
1513 conjugate gradient iterations, at rate of 24634.8 iters/second
300 newton iterations
--------------------------------------------------------------------------------
### 4, 128, 100, 1513, 300, 0.0614173 ###
Goodbye!
```

For readability, only one process should output this message.
Note: You can use the `-reservation=hpc-monday` or `-reservation=hpc-thursday` for better priority.

1.2 Domain decomposition [10 Points]

Design a domain decomposition strategy that is compatible with any square grid size and any number of MPI processes[1]. Implement this strategy by storing the decomposition details in the `data::Discretization` and `data::SubDomain` structs. In your report, explain your chosen method of domain decomposition. Discuss why this method was selected and analyze its implications on the performance of the application. Consider aspects such as load balancing, communication overhead, and computational efficiency in your analysis.
Hint: Consider the number of data that must be communicated by a generic process with its neighbors. See the short discussion in [1, Sec. 10.4.1].

1.3 Linear algebra kernels [5 Points]

Parallelize the relevant linear algebra kernels `hpc_XXX` in `linalg.cpp` using MPI. Please briefly explain in your report which functions you modified and which you did not.

[1] We naturally exclude the practically irrelevant case where the number of MPI processes exceeds the grid size.

1.4 The diffusion stencil: Ghost cells exchange [10 Points]

Implement the ghost cell exchange between neighboring processes. See Fig. 1 for an illustration. Use *non-blocking* point-to-point communication with the objective to overlap computation and communication. Explain your approach in your report.
Hint: Copy the corresponding cell data into the *send buffers* `buffN`, `buffS`, `buffE` and `buffW`. Similarly, receive the ghost cell data into the *receive buffers* `bndN`, `bndS`, `bndE` and `bndW`.

1.5 Implement parallel I/O [10 Points]

Implement the output of the computed solution with MPI-I/O.
Hint: Study the MPI-I/O demo in the provided skeleton codes.

1.6 Strong scaling [10 Points]

How does your code scale for different resolutions? Plot the time to solution for $N_{\text{CPU}} = 1, 2, 4, 8, 16$ processes across resolutions of $n \times n$, where $n = 64, 128, 256, 512, 1024$. Interpret your results and compare them to the OpenMP implementation of Project 3 in your report.

1.7 Weak scaling [10 Points]

How does code scale for constant work by process ratio? Plot the time to solution as the total problem size and the number of processes $N_{\text{CPU}} = 1, 2, 4, 8, 16$ increase proportionally, to maintain a constant workload per process, and the base resolutions $n \times n$ and $n = 64, 128, 256$. Interpret your results and compare them to the OpenMP implementation of Project 3 in your report.

2 Python for High-Performance Computing [25 points]

Python is increasingly used in High-Performance Computing (HPC) projects, serving various roles such as a high-level interface to existing applications and libraries, an embedded interpreter, or even for direct implementation. Its popularity in scientific computing has surged due to its flexibility and ease of use. Users now commonly use Python not only to prototype codes at small scales but also to develop parallel production codes. This shift is partly replacing traditional compiled HPC languages such as C/C++ and Fortran for certain applications. However, when adopting Python for such purposes, it is crucial to monitor performance to meet the rigorous demands of HPC environments. Similar bindings exist for other popular languages such Julia[2] (see MPI.jl[3]).
We highly recommend to (partly) watch the course High-Performance Computing with Python[4] held July 02–04, 2019 at CSCS. In particular, we will use the package MPI for Python (`mpi4py`) for using MPI within Python. To get started, please watch the Introduction to MPI[5] lesson of the CSCS course. Although the lessons use mostly IPython/Jupyter notebooks, we will use plain Python scripts.
ICS Cluster environment setup instructions: In order to run mpi4py on the ICS Cluster you will need to install a custom environment using anaconda that has all of the libraries that you will need to run the code. To set up this environment you will need the text file `project5_conda_env.txt` that is located in the hpc-python directory.
To set up the environment navigate to the directory containing the `project5_conda_env.txt` file and run the following commands:

```
[user@icslogin01 hpc_python]$ source /apps/miniconda3/bin/activate
[user@icslogin01 hpc_python]$ conda create --name project5_env --file
 ↪  project5_conda_env.txt
[user@icslogin01 hpc_python]$ conda init bash
[user@icslogin01 hpc_python]$ exit
```

[2] https://julialang.org/
[3] https://juliaparallel.org/MPI.jl/stable/
[4] https://www.youtube.com/watch?v=JYX4TQ_fCqY&list=PL1tk5lGm7zvQ-EzsiTZ6Xv1SxZs74epzg
[5] https://www.youtube.com/watch?v=XeyspDaKjMM

```
[uname@personal_computer]$ ssh rosa
(base) [userm@icslogin01 hpc_python]$ conda activate project5_env
```

Please note that if you have the OpenMPI module loaded in your current session on the cluster you will have problems running the code. Please start a new session for running Python MPI code and do not load the OpenMPI module.
For Python, we refer to the documentation

- `https://docs.python.org/3/`

The documentation for `mpi4py` can be found here

- `https://mpi4py.readthedocs.io/en/stable/index.html`

Remember to use the help function within a Python interpreter:

```
>>> from mpi4py import MPI
>>> help(MPI)
```

In order to get started, we begin with a simple Python MPI program `hello.py`:

```
from mpi4py import MPI

# get comm, size, rank & host name
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
proc = MPI.Get_processor_name()

# hello
print(f"Hello world from processor {proc}, rank {rank} out of {size} processes")
```

Run the script as follows:

```
(base) [user@icslogin01 hpc_python]$ salloc --nodes=4 --ntasks=8 --ntasks-per-node=2
(base) [user@icslogin01 hpc_python]$ conda activate project5_env
(project5_env) [user@icslogin01 hpc_python]$ mpiexec -n 8 python hello.py
Hello world from processor icsnode33, rank 0 out of 8 processes
Hello world from processor icsnode33, rank 1 out of 8 processes
Hello world from processor icsnode34, rank 2 out of 8 processes
Hello world from processor icsnode34, rank 3 out of 8 processes
Hello world from processor icsnode36, rank 7 out of 8 processes
Hello world from processor icsnode35, rank 4 out of 8 processes
Hello world from processor icsnode36, rank 6 out of 8 processes
Hello world from processor icsnode35, rank 5 out of 8 processes
```

Now that everything is set up and working, we can get started!
Note: If the warning about `-ntasks-per-node` appears but the tasks are distributed correctly across nodes, it can be safely ignored. This warning does not impact the performance or execution of the job.

2.1 Sum of ranks: MPI collectives [5 Points]

With MPI for Python's collective communication methods, write a script that computes the sum of all ranks:

- using the pickle-based communication of generic Python objects, i.e. the *all-lowercase* methods;

- using the fast, near C-speed, direct array data communication of buffer-provider objects, i.e. the method names starting with an *uppercase* letter.

2.2 Ghost cell exchange between neighboring processes [5 Points]

Write a script that creates a two-dimensional, periodic Cartesian topology and implements ghost cell exchange (as in Project 4):

- use the method `MPI.Compute_dims` , a convenience function similar to MPI's `MPI_Dims_create` ;
- create a Cartesian topology using MPI for Python;
- determine the neighboring processes;
- output the topology: rank, Cartesian coordinates in decomposition, East/West/North/South neighbors;
- for each process, exchange its rank with the four east/west/north/south neighbors.

Verify that you obtain the expected result.

2.3 A self-scheduling example: Parallel Mandelbrot [15 Points]

In this task, you are asked to implement one of the most common parallel patterns: the *manager-worker* pattern. The basic idea is that one process, known as the manager, is responsible for delegating work to other processes, known as the workers. This is particularly useful in problems where the amount of work per worker is difficult to estimate and the workers don't have to communicate with each other in order to do their work. As a particular example, we again consider the Mandelbrot set. Note that this is only meant as an illustration of this fundamental type of parallel algorithm, and not really as the best way to parallelize the computation of the Mandelbrot set.

The manager decomposes the Mandelbrot set into a number of (rectangular) patches. Computing the Mandelbrot (sub)set on a particular patch will be called a task. The manager then delegates these tasks to the workers. Once a worker is done computing a particular task, he sends the patch back to the manager. Therewith, the worker signals to the manager that he is available to work on a new task. The manager then sends the worker another task to work on. This process is repeated until no more tasks remain, i.e. all the patches of the Mandelbrot set have been computed. Finally, the manager combines all the patches from the workers and outputs the Mandelbrot set.

The skeleton codes for this sub-project are located in the folder `hpc_python/ManagerWorker` available through the course iCorsi page. Begin by familiarizing yourself with the `mandelbrot_task.py` module. It contains two classes. First, the class `mandelbrot` , which decomposes the Mandelbrot set computation in a series of subsets or patches, produces a list of tasks, and combines the tasks' patches together. Second, the `mandelbrot_patch` class, which holds a subset or patch of the Mandelbrot set and contains a method `do_work` that performs the actual computation. This part is already fully implemented for your convenience. However, feel free to try out different implementations, e.g., domain decompositions, etc.

Complete the following:

- Implement the manager-worker algorithm in the skeleton code `manager_worker.py`.
- Add a scaling study using 2-16 workers for a `4001x4001` domain and split the workload into 50 and 100 tasks.

The program can be called as follows:

```
(project5-env) [user@icslogin01 ManagerWorker]$ mpiexec -n 4 python
↪  manager_worker.py 4001 4001 100
```

3 Quality of the Report [15 Points]

Each project will have 100 points (out of 15 point will be given to the general written quality of the report).

Additional notes and submission details

Submit the source code files (together with your used `Makefile`) in an archive file (tar, zip, etc.) and summarize your results and the observations for all exercises by writing an extended Latex report. Use the Latex template from the webpage and upload the Latex summary as a PDF to iCorsi.

- Your submission should be a gzipped tar archive, formatted like project_number_lastname_firstname.zip or project_number_lastname_firstname.tgz. It should contain:
  - All the source codes of your solutions.
  - Build files and scripts. If you have modified the provided build files or scripts, make sure they still build the sources an run correctly. We will use them to grade your submission.
  - project_number_lastname_firstname.pdf, your write-up with your name.
  - Follow the provided guidelines for the report.
- Submit your .tgz through iCorsi.

Code of Conduct and Policy

- Do not use or otherwise access any on-line source or service other than the iCorsi system for your submission. In particular, you may not consult sites such as GitHub Co-Pilot or ChatGPT.
- You must acknowledge any code you obtain from any source, including examples in the documentation or course material. Use code comments to acknowledge sources.
- Your code must compile with a standard-configuration C/C++ compiler.

Please follow these instructions and naming conventions. Failure to comply results in additional work for the TAs, which makes the TAs sad...

References

[1] Georg Hager and Gerhard Wellein. Introduction to high performance computing for scientists and engineers. *Chapman & Hall/CRC Computational Science*, July 2010. URL: http://dx.doi.org/10.1201/EBK1439811924, doi:10.1201/ebk1439811924.