Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

HPC Lab for CSE, Spring Semester 2021
Lecturer: Dr. R. Käppeli, Prof. O. Schenk
Assistants: A. Dolfini, L. Gaedke-Merzhaeuser
T. Holt, R. Janalik, I. Labarca, D. Pasadakis,
G. Pollak, M. Sudwoj

Project 6 – High-Performance Computing with Python
Due date: 20 May 2021 12pm (midnight)

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 the 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. The performance of pure Python programs is often suboptimal, but there are ways to make them faster and more efficient. In this project you will learn various ways to optimize and parallelize Python programs, particularly in the context of scientific and high-performance computing. Additionally, we will introduce it through some examples. We will also look at alternative algorithms, e.g, using self-scheduling techniques for parallelizing the computation of, e.g., the Mandelbrot set.

You may do this project in groups of two or three students. In fact, we prefer that you do so.

1. Parallel Space Solution of a nonlinear PDE using MPI [in total 35 points]

This subproject 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 miniapplication, so that we could use all cores on one compute node on the Euler cluster . The goal of this exercise is now to use MPI, so that we are able to use multiple compute nodes. In both the serial and the OpenMP versions, there was only one process that had all the data. In the MPI version, we now have multiple processes (ranks). The computational grid on which we compute will be divided into equal subgrids, and each subgrid will be assigned to a process. Each process will only have access to its own subgrid and cannot access the data that belongs to other processes. In order to compute the new value at a given grid-point, the values at all its neighboring grid-points will be needed. In case this grid-point is on the boundary of a process' subgrid, we would need to get the value from the neighboring process that has these data. So, before each iteration, all the MPI processes first exchange the "ghost cells" and save these values from their respective neighbors to the boundary buffers (bndN, bndS, bndE, bndW). After the exchange, each process has all the data it needs to compute the next iteration.

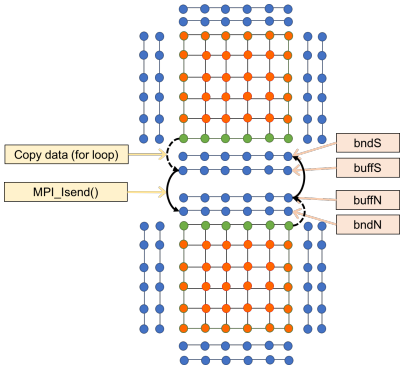

Figure 1: Ghost cell exchange: copy the north (south) row to buffN (buffS), send buffN (buffS) to the neighbor, receive to bndS (bndN).

You can find an initial incomplete version of the MPI code in the directory `pde-miniapp`. The source code is almost equivalent to the OpenMP version that you have already implemented in project 3. In this subproject you should modify the same files as before. There are some comments below that will guide you during the implementation process.

Hints

Note that at the beginning the initial version of the code is incomplete and it is your task to add the missing MPI functionality. As a first step you need to initialize MPI and fillin the missing parts. When you finalize the ghost cells

exchange in (`operators.cpp`), you can check the functionality by looking at the resulting final image. You might observe that only parts of the image are correct, or parts of the image are flipped in n-s or e-w directions. Think about why this happens. It will help you to find what is wrong in your code.

For the testing of your final MPI code, you can use the same parameters as for the OpenMP version, for example

```
[user@eu-login]$  ./main 128 100 0.005
```

Don't forget that you have to use `mpirun` to launch the MPI application.

```
[user@eu-login]$ module load new gcc/6.3.0 open_mpi/3.0.0 python/3.6.0
[user@eu-login]$ cd pde-miniapp/
[user@eu-login]$ make
[user@eu-login]$ bsub -n 4 -W 00:30 mpirun ./main 128 100 0.005
```

Below we will list several steps that you need to take in order to finalize this project.

1.1. Initialize and finalize MPI [5 Points]

In the file `main.cpp` we need to add the initial MPI code so that the MPI environment is initialized with `MPI_Init`. During `MPI_Init`, all of MPI's global and internal variables are constructed. For example, a communicator is formed around all of the processes that were spawned, and unique ranks are assigned to each process. You also need to add MPI code so that each process has its own rank as well. In particular we need to add the following:

- Initialize MPI.
- Get current rank and number of ranks.
- Finalize MPI.

1.2. Create a Cartesian topology [5 Points]

You need to generate a 2D domain decomposition (MPI communicator) of a given grid depending on the number of ranks similar to project 4 ("Ghost cells exchange between neighboring processes"). In the file `data.cpp`:

- Create the dimensions of the decomposition depending on the number of ranks (using "MPI_Dims_create").
- Create a *non-periodic* Cartesian topology for the grid of domains (using "MPI_Cart_create").
- Identify the coordinates of the current rank in the domain decomposition (using "MPI_Cart_coords").
- Identify the neighbors of the current rank: east, west, north and south directions (using "MPI_Cart_shift").

1.3. Extend the linear algebra functions [5 Points]

Implement the dot product and the norm computation where a vector is distributed over all ranks. Think about why this is only necessary for these two functions and not the others?
In the file `linalg.cpp`:

- Add a collective operation to compute the dot product (using "MPI_Allreduce").
- Add a collective operation to compute the norm (using "MPI_Allreduce").

1.4. Exchange ghost cells [10 Points]

Use point-to-point communication to exchange ghost cells amongst neighbours.
In the file `operators.cpp`:

- Add point-to-point communication for all neighbours in all directions.
- Use *non-blocking* communication (using "MPI_Irecv" and "MPI_Isend").
- Try to overlap computation and communication.

Be careful to send first row / last row / column. Before you send the data, copy it to the send buffers (buffN, buffS, buffE, buffW), and receive it in the ghost cells (bndN, nbdS, bndE, bndW). Because you copy the data to a 1D array first, you don't need any custom data types for send and receive. Look at Figure 1 for an illustration of the ghost cells exchange.

1.5. Scaling experiments [10 Points]

How does it scale at different resolutions? Consider both weak- and strong-scaling for this project and use iters/sec as your measurement. Plot, the time to solution using 1-32 MPI ranks on 1 compute node for the grid sizes:

- 128×128
- 256×256
- 512×512
- 1024×1024

And then repeat the same measurement for 1 process per node. Analyze and interpret your results and compare it to the OpenMP implementation of Project 3. *Hint:* you can use the `span[ptile=n]` option to access different nodes.

2. Python for High-Performance Computing (HPC) [in total 50 points]

Python is increasingly used in high-performance computing projects. It can be used either as a high-level interface to existing HPC applications and libraries, as an embedded interpreter, or directly. In this project, we will show how Python can be used on parallel architectures to parallelize the nonlinear PDE solver project using NumPy to explore the productivity gains made possible by Python for HPC.

In recent years the Python programming language has become more and more popular in scientific computing for various reasons. Users not only implement prototypes for numerical experiments on small scales, but also develop parallel production codes, thereby partly replacing compiled languages such as C or C++. However, when following this approach it is crucial to pay special attention to performance. This tutorial course teaches "High-Performance Computing with Python" approaches to use Python efficiently and reasonably in an HPC environment. We recommend to have an initial look at this course which was held from July 02–04, 2019 at CSCS:

- `https://www.youtube.com/watch?v=JYX4TQ_fCqY&list=PL1tk5lGm7zvQ-EzsiTZ6Xv1SxZs74epzg`

We will use the package MPI for Python (`mpi4py`) for using MPI within Python. Begin by watching the lesson of the CSCS course on MPI:

- `https://www.youtube.com/watch?v=XeyspDaKjMM`

Although the lessons use mostly IPython/Jupyter notebooks, we will use plain Python scripts. 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)
```

For Python, we refer to the documentation

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

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

```
from mpi4py import MPI

# get comm, size & rank
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

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

In order to run the script, first load the following module on the Euler cluster :

```
[user@eu-login]$ module load new gcc/6.3.0 open_mpi python hdf5
```

Submit the script to be run in parallel to the queue with

```
[user@eu-login]$ bsub -n 4 -W 00:05 mpirun python3 hello.py
```

The output of the script (see the `lsf.o*` file) should look like (up to the order):

```
Hello world from rank 0 out of 4 processes
Hello world from rank 1 out of 4 processes
Hello world from rank 2 out of 4 processes
Hello world from rank 3 out of 4 processes
```

Now that everything is set up and working, we can get started!

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. Domain decomposition: Create a Cartesian topology [5 Points]

Write a script that computes a 2D *periodic* process distribution depending on the number of processes, and creates a Cartesian topology:

- 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 neighbouring processes,
- output the topology: rank, Cartesian coordinates in decomposition, East/West/North/South neighbours.

2.3. Exchange rank with neighbours [5 Points]

Next, we are going to exchange data within the periodic Cartesian topology from the previous task.
For each process, exchange its rank with the four east/west/north/south neighbours. Verify that you obtain the expected result.

2.4. Parallel space solution of a nonlinear PDE using Python and MPI

In this task, we are going to complete a Python implementation of the previously introduced nonlinear PDE using C/C++ and MPI example. You can find the `pde_miniapp_py` code on the usual git or Moodle repositories. The code largely follows the structure of the previous C/C++ implementation. You can run the skeleton code with

```
[user@eu-login]$ bsub -n 4 -W 00:05 mpirun python3 main.py 128 100 0.005 verbose
```

In this example, the simulation is run with four MPI processes, a grid of size 128^2, 100 time steps until a final time of 0.005. You can draw the solution with the `draw.py` script:

```
[user@eu-login]$ python3 draw.py
```

You can adapt this script for debugging purposes.

2.4.1. Change linear algebra functions [5 Points]

In the `linalg` module (`linalg.py`):

- Complete the dot product computation, method `hpc_dot`.
- Complete the norm computation, method `hpc_norm2`.

Python is an interpreted language and therefore executes loops much slower than compiled languages do. Conveniently, many popular Python libraries are actually implemented in efficient compiled languages such as C. Try writing these functions firstly using a standard for-loop implementation as you would do in C, and then using numpy functions. Please briefly comment on the speed difference observed between the different methods. Your submitted code should contain only the faster of the two implementations (you may comment-out the slower implemetation).

2.4.2. Exchange ghost cells [5 Points]

The solution data are contained within the `Field` class in the `data` module (`data.py`). The class holds one-dimensional (1D) Numpy arrays `bdryN/E/S/W`, supposed to contain ghost points from neighbouring processes, and 1D Numpy buffer arrays `buffN/E/S/W`, supposed to store/buffer data to be sent to neighbouring processes. Communication is started by calling the `exchange_startall` method, and waiting until the communication is completed is handled by the `exchange_waitall` method.
Complete the following tasks:

- Implement the `exchange_startall` method using `Isend` and `Irecv` methods to initiate send and receive of operations.
  Remark: make sure that you understand the difference between `mpi4py`'s *all-lowercase* and first letter *uppercase* methods.
- Implement the `exchange_waitall` routine.
- Verify that you obtain results that are consistent with your C/C++ implementation.

2.4.3. Scaling experiments [5 Points]

Repeat the scaling experiments from 1.5 using 1-32 MPI ranks. Analyze and interpret your results, also in comparison to the behavior of the C++ implementation. *Remark:* The Python version is expected to be significantly slower.

2.5. A self-scheduling example: Parallel Mandelbrot [20 Points]

In this task, you are asked to implement one of the most common parallel algorithm prototypes: the *self-scheduling*, or *manager-worker*, or *master-slave*[1], algorithm. 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 consider the Mandelbrot set again. 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 image.

The skeleton codes for this subproject are located in the folder `hpc-python/ManagerWorker` available through the lecture git/ Moodle repository. 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 2,4,8, and 16 workers (or more if the Euler cluster allows) splitting the workload once into 50 and once into 100 tasks.

The program can be called as follows:

```
[user@eu-login]$ bsub -n 4 -W 00:05 mpirun python3 manager_worker.py 4001 4001 100
```

3. Task: Quality of the Report [15 Points]

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

Additional notes and submission details

We only accept submissions using our Latex template and C/C++ code. Please submit the source code files (together with your used `Makefile`) in an archive file (tar, zip, etc.), and summarize your results and observations for all exercises by writing an extended Latex report. Use the Latex template provided on the webpage and upload the Latex summary as a PDF to Moodle .

[1] This traditional naming convention has recently come under fire for being politically incorrect.

- 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;
  - your write-up with your name project_number_lastname_firstname.pdf.
- Submit your .zip/.tgz through Moodle .