ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

HPC Lab for CSE

FS 2024
Dr. R. Käppeli,
Dr. A. Schell, M. R. Herde,
C. Humbel, A. Mazzoleni,
N. A. Siegenheim

Project 4

Due date: Monday 29 April 2024, 23:59 (midnight).

This assignment will introduce you to parallel programming using the Message Passing Interface (MPI). You will implement a simple MPI message exchange, compute a process topology, parallelize the computation of the Mandelbrot set, and finally a parallel matrix-vector multiplication used within the power method. You may do this project in groups of students (max four or five). In fact, we prefer that you do so. Please clearly indicate who you worked with in the "Discussed with" field of your report.

If MPI is new to you, we highly recommend the LLNL tutorial[1]. Likewise, we also highly recommend [3, Chap. 8] and [2, Chap. 1-5][2]. The MPI standard document you find on the MPI forum website[3] is very valuable source of information.

For the whole project, the simulations must be run on the Euler (phase 2) nodes with AMD EPYC 7763 CPUs (for which we have privileged access during the lab hours), and we will use the new software stack (LMOD Modules) using the GNU Compiler Collection and OpenMPI MPI library:

```
[user@eu-login-39 ~]$ module load gcc openmpi
[user@eu-login-39 ~]$ mpicc -v
Reading specs from ...
COLLECT_GCC=...
COLLECT_LTO_WRAPPER=...
Target: x86_64-pc-linux-gnu
Configured with: ...
Thread model: posix
Supported LTO compression algorithms: zlib
gcc version 11.4.0 (GCC)
```

However, feel free to try and develop on other available systems (e.g., your workstation or laptop) and compilers, but please make sure to document them in your report if you include results.

While developing/debugging your code, it can be useful to work in an interactive allocation as follows:

```
[user@eu-login-39 ]$ salloc --ntasks=4 --constraint=EPYC_7763
salloc: Pending job allocation 55052348
salloc: job 55052348 queued and waiting for resources
salloc: job 55052348 has been allocated resources
salloc: Granted job allocation 55052348
salloc: Waiting for resource configuration
salloc: Nodes eu-a2p-415 are ready for job
[user@eu-login-39 ]$ mpirun hello_mpi # run hello world in skeleton code
Hello world from rank 3 out of 4 on eu-a2p-415
Hello world from rank 0 out of 4 on eu-a2p-415
Hello world from rank 1 out of 4 on eu-a2p-415
Hello world from rank 2 out of 4 on eu-a2p-415
```

This allocates a job with up to 4 tasks (MPI processes) on Euler VII (phase 2) and 4GB of memory for one hour. In this allocation, we got all tasks assigned to one single node `eu-a2p-415`. By default, `mpirun` will use all available tasks (here 4). To run with less (e.g., for doing a scaling study), use the `-np N` option, where `N` is the desired number of MPI processes. For example:

[1] https://hpc-tutorials.llnl.gov/mpi/
[2] Both books also feature some advanced topics of interest and you find links to the books on the course Moodle page.
[3] https://www.mpi-forum.org/

```
[user@eu-login-39 ]$ mpirun -np 2 hello_mpi # run hello world in skeleton code
Hello world from rank 0 out of 2 on eu-a2p-415
Hello world from rank 1 out of 2 on eu-a2p-415
```

We can also ask Slurm to allocate the job on separate nodes:

```
[user@eu-login-39 ]$ salloc --ntasks=4 --nodes=4 --constraint=EPYC_7763
salloc: Pending job allocation 55052665
salloc: job 55052665 queued and waiting for resources
salloc: job 55052665 has been allocated resources
salloc: Granted job allocation 55052665
salloc: Waiting for resource configuration
salloc: Nodes eu-a2p-[410,415-417] are ready for job
[user@eu-login-39 ]$ mpirun hello_mpi # run hello world in skeleton code
Hello world from rank 0 out of 4 on eu-a2p-410
Hello world from rank 1 out of 4 on eu-a2p-415
Hello world from rank 2 out of 4 on eu-a2p-416
Hello world from rank 3 out of 4 on eu-a2p-417
```

Please see also the example batch scripts in the `hello_mpi` example of the provided skeleton source codes. Last but not least, the Euler wiki and the Slurm documentation are essential resources.
As usual, you find all the skeleton source codes for the project on the course Moodle page.

1 Ring sum using MPI [10 Points]

This task familiarizes you with some basic MPI send/receive functionality and identification of the neighbors in a one-dimensional process layout known as a ring topology. The processes are organized in a circular chain along their MPI ranks, where each process has two neighbors and first and last processes are neighbors as well. The ring sum using MPI algorithm proceeds in the following way: every process initially sends its rank number to a neighbor (in increasing rank direction, except for the first and last processes); then every process sends what it receives from that neighbor. This is done n times, where n is the number of processes. As a result, all ranks will accumulate the sum of all ranks. The first two iterations of the algorithm are illustrated in Fig. 1.

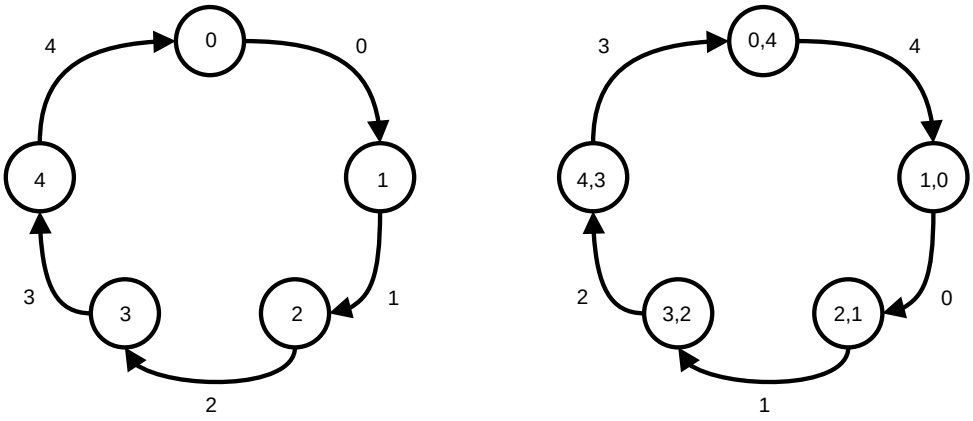


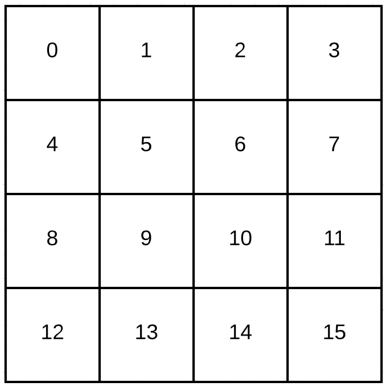
Figure 1: First two iterations of the ring sum algorithm with $n = 5$ MPI processes. The ring sum is equal to 10.

Your task: Implement the ring sum algorithm in the provided skeleton code `ring`. Be careful to avoid any potential deadlock issues in your implementation. In your report, briefly comment on your chosen communication pattern, particularly how you avoid potential deadlock issues.

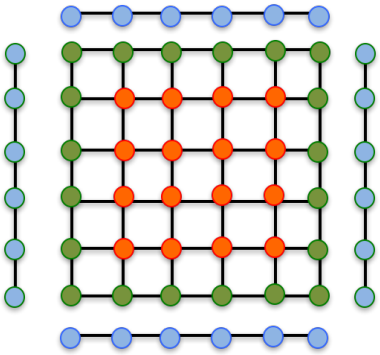
2 Cartesian domain decomposition and ghost cells exchange [20 Points]

The objective of this task is to write an MPI program that partitions a two-dimensional Cartesian domain into a number of sub-domains and to exchange so-called *ghost cells*[4] between neighboring MPI processes. The term ghost cells refers to a copy of remote process' data in the memory space of the current process. This domain decomposition and ghost cells exchange is extensively used in stencil-based kernels such as in the mini-app solving Fisher's equation in Project 3 need data from grid cells residing on another process.

In this task, we consider a two-dimensional and periodic toy domain discretized by 24×24 grid points. This domain is distributed over a 4×4 "grid of processes". Each process holds a local domain discretized by 6×6 grid points, which is extended by one row/column on each side in order to accommodate the copy of its neighbors' borders, i.e., the ghost cells. For simplicity, we ignore the corner cells. The grid of processes and a local domain is illustrated in Fig. 2. The processes are arranged in a so-called Cartesian topology with periodic boundaries, which means that, for example, process with rank 0 is also a neighbor of processes with rank 3 and 12. Therefore, each process has four neighbors and these are often referred to the north, south, east and west neighbor. The exchange of ghost cells between processes of rank 5 and 9 is illustrated in Fig. 3.



(a) Domain distributed over 4×4 "grid of processes".



(b) Local domain discretized by 6×6 grid points extended by one row/column of ghost cells on each side.

Figure 2: Cartesian domain decomposition.

We provide a skeleton code in the `ghost` directory and its compilation and execution on the Euler cluster are shown below:

```
[user@eu-login]$ cd ghost/
[user@eu-login]$ make
[user@eu-login]$ salloc --ntasks=16 --constraint=EPYC_7763
[user@eu-login]$ mpirun -np 16 ./ghost
```

Note: If you are developing on your own machine (or a machine with less than 16 processors), you will need to *oversubscribe* your processors. With OpenMPI, this is achieved with flag `--oversubscribe` (see here for details).
The result of the boundary exchange on rank 9 should be

```
 9.0  5.0  5.0  5.0  5.0  5.0  5.0  9.0
 8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
 8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
 8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
 8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
 8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
 8.0  9.0  9.0  9.0  9.0  9.0  9.0 10.0
 9.0 13.0 13.0 13.0 13.0 13.0 13.0  9.0
```

[4] The terms *guard* or *halo cells* is also used.

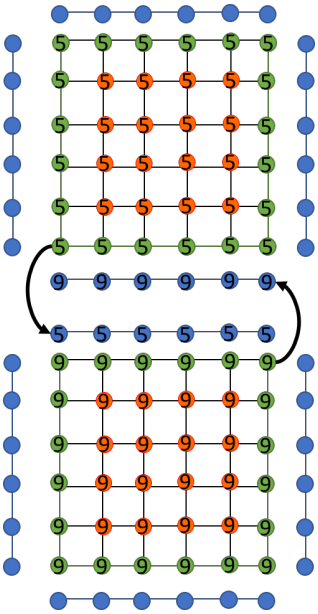


Figure 3: Exchange of ghost cells between processes of rank 5 and 9.

Starting from the provided skeleton code, complete the following tasks:

1. Create a Cartesian two-dimensional MPI communicator (4×4) with periodic boundaries and use it to find your neighboring ranks in all dimensions in a cyclic manner.
   Hint: Use `MPI_Cart_create` and related functions.
2. Create a derived data type for sending a column boundary (east and west neighbors).
   Hint: Use `MPI_Type_vector` and related functions.
3. Exchange ghost cells with the neighboring cells in all directions and verify that correct values are in the ghost cells after the communication phase. Be careful to avoid any potential deadlock issues in your implementation.
4. **Bonus [10 Points]**: Also exchange ghost values with the neighbors in ordinal directions (northeast, southeast, southwest and northwest).

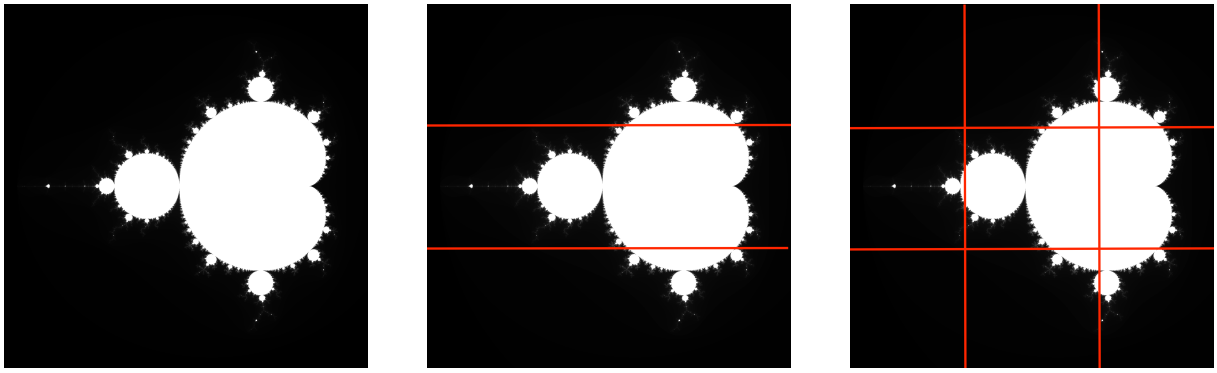
3 Parallelizing the Mandelbrot set using MPI [30 Points]

The goal of this task is to parallelize the Mandelbrot set computation from Project 2 using MPI. The computation of the Mandelbrot set will be partitioned into a set of parallel MPI processes, where each process will compute only its local portion of the Mandelbrot set. Examples of a possible partitioning are illustrated in Fig. 4. After each process completes its own computation, the local domain is sent to the master process that will handle the I/O and create the output image containing the whole Mandelbrot set.

We introduce two structures, that represent the information about the partitioning. These structures are defined in `consts.h`. The structure `Partition` represents the layout of the grid of processes and contains information such as the number of processes in x and y directions and the coordinates of the current MPI process:

```
typedef struct {
    int x;          // x-coordinate of current MPI process in the process grid
    int y;          // y-coordinate of current MPI process in the process grid
    int nx;         // Number of processes in x-direction
    int ny;         // Number of processes in y-direction
    MPI_Comm comm; // (Cartesian) MPI communicator
} Partition;
```

The second structure `Domain` represents the information about the local domain of the current MPI process. It holds information such as the size of the local domain (number of pixels in each dimension)



(a) One partition. (b) Three partitions. (c) Nine partitions.

Figure 4: Possible partitioning of the Mandelbrot set.

and its global indices (index of the first and the last pixel in the full image of the Mandelbrot set that will be computed by the current process):

```
typedef struct {
    long nx;     // Local  domain size in x-direction
    long ny;     // Local  domain size in y-direction
    long startx; // Global domain start index of local domain in x-direction
    long starty; // Global domain start index of local domain in y-direction
    long endx;   // Global domain end   index of local domain in x-direction
    long endy;   // Global domain end   index of local domain in y-direction
} Domain;
```

The skeleton code you find on Moodle is initialized in a way that each process computes the whole Mandelbrot set. Your task is to partition the domain (so that each process only computes an appropriate part), compute the local part of the image, and send the local data to the master process that will create the final complete image. The compilation and execution on the Euler cluster are shown below:

```
[user@eu-login]$ cd mandel/
[user@eu-login]$ make
[user@eu-login]$ salloc --ntasks=16 --constraint=EPYC_7763
[user@eu-login]$ mpirun -np 16 ./mandel_mpi
```

Running the provided benchmark script and creating the performance plot is achieved by submitting the job to the Slurm queuing system. The runtime configuration controling the number of nodes and MPI ranks is specified in the `run_perf.sh` script:

```
[user@eu-login]$ sbatch run_perf.sh
```

This creates a graph `perf.pdf` showing the wall-clock time of each process for multiple runs with varying number of processes ($N_{\text{CPU}} = 1, 2, 4, 8, 16$).

Solve the following tasks:

1. Create the partitioning of the image by implementing the functions `createPartition` and `updatePartition`. You can find a dummy implementation of these functions in `consts.h`.
2. Determine the dimensions and the start/end of the local domain based on the computed partitioning by implementing the function `createDomain`. The function is defined in the `consts.h`.
3. Send the local domain to the master process if `mpi_rank > 0` and receive it at the master process where `mpi_rank == 0`. Compare the output of the parallelized program to that of the sequential program in a graphic and verify that it is correct.
4. Comment the performance observed in the graph `perf.pdf` in your report. Give a suggestion to improve the performance[5].

[5] You don't have to implement your suggestion.

4 Parallel matrix-vector multiplication and the power method [40 Points]

This task is about writing a parallel program to multiply a matrix A by a vector x, and to use this routine in an implementation of the power method[6] to find the largest absolute eigenvalue of a given matrix. A serial `Python` implementation of the power method is provided in the skeleton codes `powermethod.py`. A less experienced parallel programmer has already started the implementation in the skeleton `powermethod_rows.c` and your task is to complete the parallelization. The skeleton code provides four test cases and the exact result is known for the first three cases. Use them to verify your code.

The next task is to study the parallel scalability of your implementation. To this end, use test case three `test_case = 3`, set the matrix size to `n = 10'000` and fix the maximum number of iterations to `niter = 3'000`. Fix the tolerance to a negative value to make sure that earlier convergence does not interfere with the timing measurements (e.g., `tol = -1e-6`). Do the following parallel scaling studies:

1. **Strong scaling**: Run you code for $p = 1, 2, 4, 8, 16, 32, 64$ MPI processes. Plot the runtime and the parallel efficiency as a function of the number of MPI processes.
2. **Weak scaling**: Run you code for $p = 1, 2, 4, 8, 16, 32, 64$ MPI processes and make the matrix size n grow (nearly) proportional to $\sqrt{p}$. Since both total memory and total work scale as n^2, this implies that the memory required per processor and the work done per process will remain constant as you increase p. Plot the runtime and the parallel efficiency as a function of the number of MPI processes/problem size.

Perform the strong and weak scaling study for two MPI process distributions: (i) all MPI processes on one node, and (ii) all MPI processes on different nodes. In your report, provide a short description and interpretation of your parallel scaling studies (and don't forget to include and reference the figures).

Additional notes and submission details

Submit **all the source code files** together with your used build files (e.g., `Makefile`(s)) and other scripts (e.g., batch job scripts) in an archive file (tar or zip) and summarize your results and observations for all sections by writing a detailed LaTeX report. Use the LaTeX template from the webpage and upload the report as a PDF to Moodle.

- Your submission should be a tar or zip archive, formatted like `project_number_lastname_firstname.zip/tgz`. It must 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 report with your name.
  - Follow the provided guidelines for the report.
- Submit your archive file through Moodle.

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

References

[1] Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst, editors. *Templates for the Solution of Algebraic Eigenvalue Problems*. Society for Industrial and Applied Mathematics, January 2000. doi:10.1137/1.9780898719581.

[2] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI*. MIT Press Ltd, November 2014. URL: https://ieeexplore.ieee.org/servlet/opac?bknumber=6981847.

[3] 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.

[6] See, e.g., [1, Algorithm 4.1] or https://en.wikipedia.org/wiki/Power_iteration for more information.