Università della Svizzera italiana

Institute of Computational Science ICS

High Performance Computing **2019**

Instructor: Prof. Olaf Schenk TA: Juraj Kardos, Radim Janalik, Aryan Eftekhari

Mini-project 5 — MPI

Due date: 26 November 2019, 23:55

Parallel Programming with MPI

This assignment will introduce you to parallel programming using MPI. You will implement simple message exchange, compute a process topology and parallelize computation of the Mandelbrot set.

1. Exchange of the ghost cells *(40 Points)*

The goal of this exercise is to exchange ghost cells between neighboring processes. The term “ghost cell" refers to a copy of remote process' data in the memory space of the current process. The ghost data are often needed for local computations, therefore, before proceeding with the computation, the processes need to exchange their ghost cells first. In this simple example, each process has a 6×6 local domain which is extended by 1 row/column from each side in order to accommodate the copy of its neighbors' borders (a.k.a “ghost cells"). The local domain is illustrated in red and green in Figure 1. The green cells are the local data that will be communicated to other processes. On the other hand, the ghost cells, containing the copy of the remote processes' data, are illustrated with a blue color. The local domain extended by ghost cells has therefore size $(6+2) \times (6+2)$ (we will ignore the corners). In order to easily inspect the result of the communication, we initialize the local domain of each process by its `mpi_rank`.

Furthermore, we will assume we have a $n \times n$ grid of processes organized in a Cartesian topology, as shown in Figure 2. We also assume cyclic borders, which means that, for example, process 0 is also a neighbor of process 12 and 3. Each process has 4 neighbors, sometimes referred to as neighbor on north, south, east and west. For testing purposes, always assume we are launching 16 processes. The exchange of the borders between process 5 and 9 is illustrated in Figure 3.

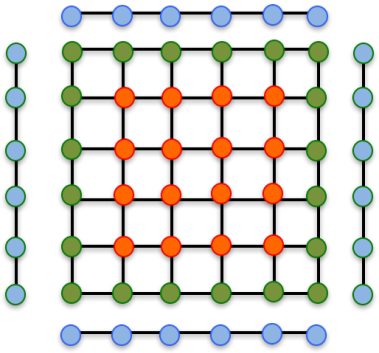

Figure 1. Local domain

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 2. Processor grid

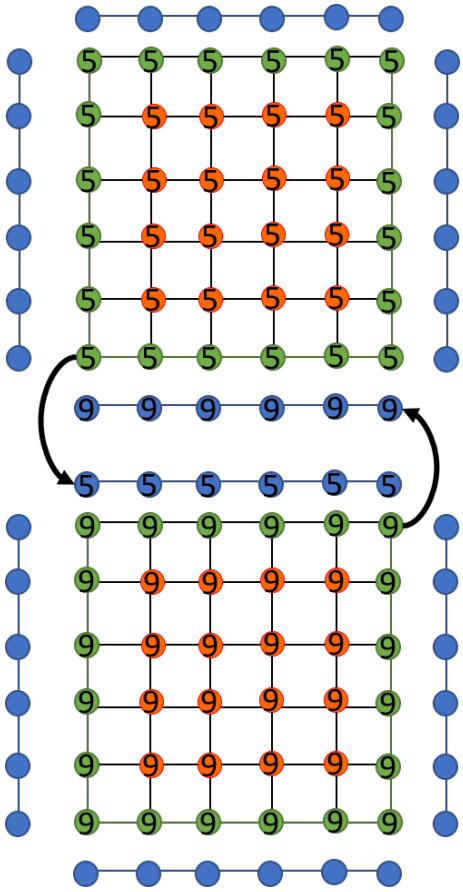

Figure 3. Exchange of the ghost cells

Compilation and execution on the cluster (please compile and run on the compute node):

```
$ module load openmpi
$ make
$ salloc -N 1 -n 16
$ mpirun -np 16 ./ghost
```

Compilation and execution on localhost (you will probably need to oversubscribe your CPU, see here):

```
$ make
$ cat my-hostfile
localhost slots=16
$ mpirun --hostfile my-hostfile -np 16 ./ghost
```

Solve the following tasks:

1. Create a Cartesian 2D communicator (4×4) with periodic boundaries and use it to find your neighboring ranks in all dimensions in a cyclic manner.
2. Create a derived data type for sending a column border (east and west neighbors).
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.

2. Parallelizing the Mandelbrot Set with MPI *(60 Points)*

In this exercise, you will parallelize Mandelbrot set computation using MPI. The computation of the Mandelbrot set will be partitioned between 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 Figure 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.

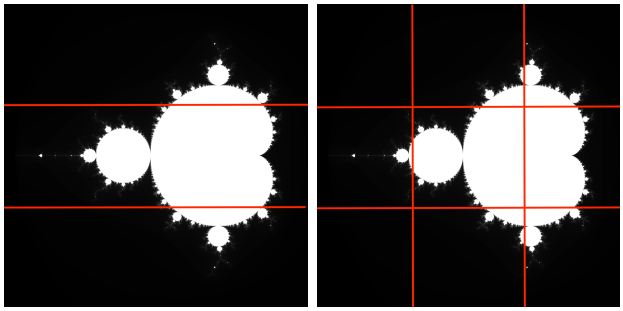

Figure 4. Possible partitionings of the Mandelbrot set

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

```
typedef struct
{
    int x; // Coordinates of the current MPI process within the processor grid
    int y;
    int nx; // Dimensions of the processor grid
    int ny;
    MPI_Comm comm;
} 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) 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; // Size of the local domain
    long ny;
    long startx; // Beggining of the local domain (global index)
    long starty;
    long endx; // End of the local domain (global index)
    long endy;
} Domain;
```

The code you will find on iCorsi or at the GitHub repository is initialized in a way that each process computes the whole Mandelbrot set. Your task will be to partition the domain (so that each process computes only 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.

Compilation and execution (please compile and run on the compute node):

```
$ module load openmpi
$ make
$ salloc -N 1 -n 20
$ mpirun -np <tasks> -npernode <tasks per node> ./mandel_mpi
```

In order to run the MPI jobs, you need to specify one more parameter to salloc (`-n` or `-ntasks`). Otherwise, you would not be able to run more than one task per node. See `man salloc` for more info. Excerpt from the manual: "-n, –ntasks=<number>: salloc does not launch tasks, it requests an allocation of resources and executes some command. This option advises the Slurm controller that job steps run within this allocation will launch a maximum of number tasks and sufficient resources are allocated to accomplish this. The default is one task per node."

Running the benchmark and creating the performance plot (please compile and run on the compute node):

```
$ module load openmpi
$ make
$ salloc -N 1 -n 20 --exclusive
$ ./run_perf.sh
$ ./plot_perf.sh # this needs to be run at the login node
```

Solve the following tasks:

1. Create partitioning of the image by implementing a body of functions `Partition createPartition(int mpi_rank, int mpi_size)` and `Partition updatePartition(Partition p, int mpi_rank)`. You can find a dummy implementation of these functions in `consts.h`.

2. Determine dimensions and the start/end of the local domain based on the computed partitioning by implementing function `Domain createDomain(Partition p)`. 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 those of the sequential program in a graphic and verify that it is correct.

4. Analyze the performance in the graph `perf.ps`. It shows computational time of each process for multiple runs with varying number of processes. Comment on the benefits of MPI parallelization and load balancing you have observed. How does the performance change when running on 1 compute node vs. multiple nodes?

5. **Bonus:** How does the performance change when you run each process with multiple OpenMP threads?

Submission:

Submission: Submit the source code files in an archive file (tar, zip, etc) and show the TA the results. Furthermore, summarize your results and the observations for all exercises by writing an extended Latex summary. Use the Latex template from the webpage and upload the extended Latex summary as a PDF to iCorsi.