Course instructor: Sergio Iserte (sisertea@uoc.edu)

Course: 2024/2025

Semester: 1st

PAC: 3 - Distributed-memory Parallel Computing with MPI

Due Date: 23rd Nov 2024 (23:59h CET)

Submission Format

PLEASE, READ IT CAREFULLY AS NO APPEALS WILL BE ADMITTED IF IT IS NOT FOLLOWED

- **A single PDF**
  - that includes all the responses and
  - the necessary scripts, screenshots, charts, etc., while
  - excluding the PAC content (only exercise statements can be included, if needed).
- **Screenshots will be evaluated only if**
  - There is a prompt from the `eimtarqso` cluster with your username.
  - Results are in output files generated by the resource management system.
  - Processes run on compute nodes. Never on the frontend (`eimtarqso`).

For example:

```
[sisertea@eimtarqso ~]$ cat ranks_run.out.894213
Hello world! I am process number 0 of 16 MPI processes on host compute-0-1.local
Hello world! I am process number 1 of 16 MPI processes on host compute-0-1.local
Hello world! I am process number 2 of 16 MPI processes on host compute-0-1.local
Hello world! I am process number 3 of 16 MPI processes on host compute-0-1.local
Hello world! I am process number 8 of 16 MPI processes on host compute-0-0.local
Hello world! I am process number 4 of 16 MPI processes on host compute-0-4.local
Hello world! I am process number 9 of 16 MPI processes on host compute-0-0.local
```

Grading Policy

Question	%
1	20
2	20
3	30
4	30
Total	100

Tip: Traditionally, the cluster is expected to be saturated near the due date since a lot of students have procrastinated the PAC. Take advantage of this situation and run your jobs as soon as possible to find the cluster idle. Once it is saturated, jobs are not likely to run and some exercises could not be completed.

1 MPI Execution Environment

The goal of this part of the assignment is to become familiar with the environment and be able to compile and execute MPI programs. We will use the following MPI program, which implements the typical *hello world* program (`hello.c`):

```
1  #include "mpi.h"
2  #include <stdio.h>
3
4  int main(int argc, char **argv) {
5      MPI_Init(&argc, &argv);
6      printf("Hello World!\n");
7
8      MPI_Finalize();
9      return 0;
10 }
```

You will note that you need to include the library *mpi.h*, as well as, it is required to start MPI programs with `MPI_Init` and end them with `MPI_Finalize`. These calls are responsible for interfacing with the MPI runtime for creating and destroying the MPI environment.

The following items illustrate the main necessary steps to compile and execute the *hello world* using MPI:

- Compile the program hello.c with mpicc (more details can be found in the documentation):

```
$ mpicc hello.c -o hello
```

- Execute the hello world MPI program through SGE. A sample script for the execution of the hello world program using six MPI processes is provided below (`hello.sge`):

```
1  #!/bin/bash
2  #$ -cwd
3  #$ -S /bin/bash
4  #$ -N hello_job
5  #$ -o $JOB_NAME.out.$JOB_ID
6  #$ -e $JOB_NAME.err.$JOB_ID
7  #$ -pe orte 6
8
9  mpirun -np 6 ./hello
```

The "-pe orte 6" option indicates that it is necessary 6 cores for the execution of the job (these will be assigned to more than one node in the UOC cluster). The mpirun command interfaces with the MPI runtime to start the execution of the program. The "-np 6" option indicates the MPI runtime to start 6 processes and the name of the program (binary) is provided at the end of the command. You can explore additional mpirun options by yourself.

2 MPI Processes

The code shown below includes the typical library calls in MPI programs. Note that the variable "rank" is essential in the execution of MPI programs as it allows identifying the number of MPI processes within a communicator (in this case `MPI_COMM_WORLD`). Please check the documentation for additional information.

```
1  #include "mpi.h"
2  #include <stdio.h>
3  #include <unistd.h>
4
5  int main(int argc, char **argv)
6  {
7    int rank, numprocs;
8    char hostname[256];
9
10   MPI_Init(&argc,&argv);
11   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12
13   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
14
15   gethostname(hostname,255);
16
17   printf("Hello world! I am process number %d of %d MPI
       processes on host %s\n", rank, numprocs, hostname);
18
19   MPI_Finalize();
20   return 0;
21 }
```

1. Run *ranks.c* without over-subscription in **32 processors** and answer the questions:

1. Where did the ranks run? Why?
2. Can you run each rank in a different node? Because of hardware limitations, spawn **one rank per node** within eight nodes. Attach a screenshot of the output.

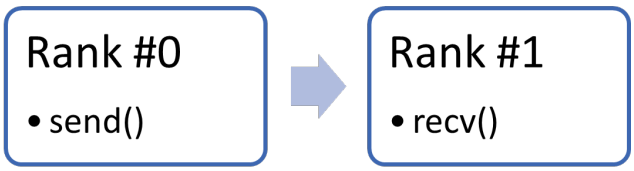
3 MPI Communication

This section introduces the basic communication mechanisms which allow MPI processes to exchange data.

Another example of an MPI program (*mpi_send_recv.c*) is provided along with this assignment description. This program is designed to be executed with only 2 MPI processes and is responsible for sending messages between the two MPI processes.

An output example is shown below:

Blocking communications, those which we are using in this CAT, are synchronized by the send/receive buffers. Particularly, processes sending data are blocked until data in the




```
Started rank #1
Started rank #0
Sending message to rank #1
Received message from rank #1
Finishing rank #0
Received message from rank #0
Sending message to rank #0
Finishing rank #1
```

send buffer is emptied; while processes receiving data are blocked until the received buffer is filled. These completions depend on the message size and the system buffer size.

Blocking communications are prone to deadlocks. For example, the MPI program *deadlock.c* implements a simple data exchange between two ranks. You can compile and run this program using the following commands (remember to run it on the compute nodes submitting the job to the queue):

```
mpicc deadlock.c -o deadlock
mpirun -np 2 ./deadlock
```

2. With regard to *deadlock.c*:

1. Why does it deadlock?
2. Explain what happens when running the code with `SIZE = 1024`? How could you provide the original behavior for any `SIZE`?
3. You may use one of the following approaches to prevent the original deadlock:
   - **0**: Non-blocking communications.
   - **1**: "Chain" sending operations (MPI_Sendrecv).
   - **2**: Only MPI_Send and MPI_Recv functions.

   Provide a deadlock-free version of the original code, and briefly describe it, based on one of those alternatives depending on your `eimtarqso` user numeric part modulo 3.
4. Attach and describe a screenshot of the trace visualized with `Paraver` and provide a brief description of it. Remember that you can extract traces with the commands in Listing 1. Thus, open `Paraver`, and load the ".prv" file. Load also the MPI configuration in "<<<paraver installation path>>>/cfgs/mpi/views/MPI_calls.cfg". Finally, click on the button "new single timeline window".
   Notice that when using `Extrae` we can have errors related to PAPI libraries that can be ignored in this exercises.

Listing 1: "Commands to load `Extrae`."

```
cd <<<working dir>>>
export EXTRAE_HOME=/share/apps/extrae
cp /share/apps/extrae/share/example/MPI/extrae.xml .
source /share/apps/extrae/etc/extrae.sh
export EXTRAE_CONFIG_FILE=extrae.xml
export LD_PRELOAD=/share/apps/extrae/lib/libmpitrace.so
mpirun -np 4 ./a.out
```

4 Example: Parallel Stencil Computation

To compute a parallel stencil, you may follow these general steps:

1. Understand the Stencil Operation: A stencil operation involves updating the value of a grid point based on the values of its neighboring grid points. The specific computation depends on the stencil pattern, which defines the relative positions and weights of the neighboring points.
2. Partition the Grid: Divide the grid into smaller subdomains, also known as patches or blocks, to enable parallel processing. Each subdomain will be assigned to a different processing unit or thread.

3. Handle Boundary Conditions: Determine how to handle the boundary points of each subdomain. Boundary conditions may require additional communication or special treatment to ensure correct computation.

4. Distribute Data: Distribute the required data to each processing unit or thread. This can be done by assigning each unit a portion of the grid and providing the necessary data for the stencil operation.

5. Perform Local Computations: Each processing unit or thread performs the stencil computation on its assigned subdomain. It updates the values of the grid points based on the stencil pattern and the values of neighboring points.

6. Exchange Boundary Data: If boundary points depend on neighboring subdomains, exchange the boundary data between adjacent processing units. This ensures that each unit has the necessary data for the correct computation of stencil points near the subdomain boundaries.

7. Synchronize: Synchronize the processing units or threads to ensure that all computations are completed before moving on to the next iteration or step.

8. Repeat: If the stencil operation requires multiple iterations, repeat the steps above until the desired convergence or accuracy is achieved.

In this example, we will draw our attention to points 1-5. Furthermore, we describe a fixed scenario of a $8x8$ matrix with a boundary thickness of 1 (see Figure 1).

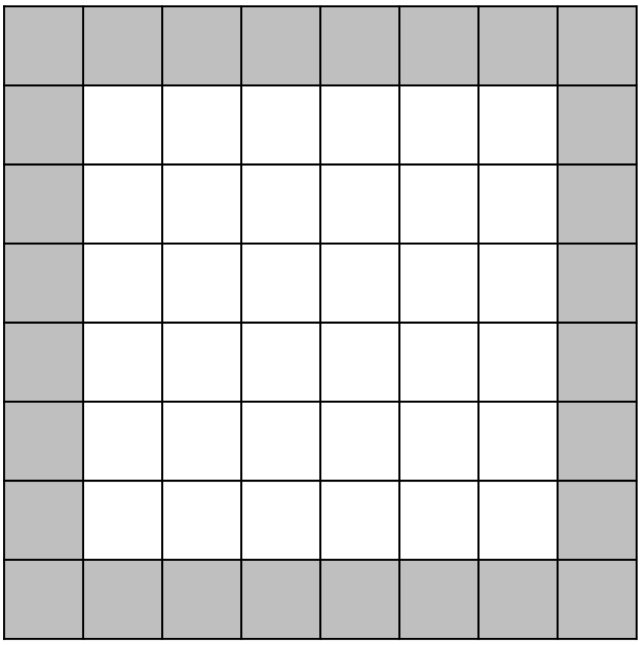


Figure 1: Problem grid with the boundary.

Figure 2 illustrates the computation of the stencil for each of the first positions of the grid, where the adjacent positions are involved. For instance, it could be computed as:

$$x_{new}[i][j] = (x[i][j+1] + x[i][j-1] + x[i+1][j] + x[i-1][j])/4$$

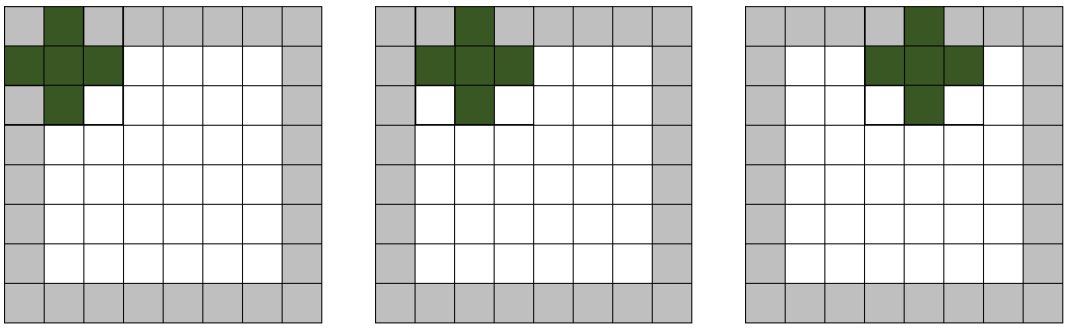


Figure 2: Stencil computation sequence.

In order to distribute the problem among the processes (in this case four), we would have the partitions showcased in Figure 3.

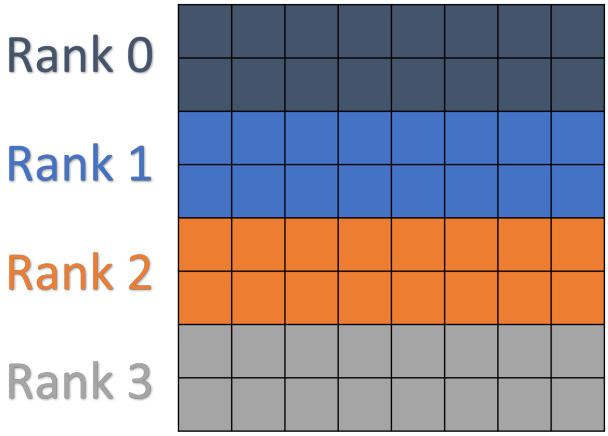


Figure 3: Domain partition.

However, in the limits of the subdomains, we need the data in other processes. To handle this difficulty, we define halos which will contain the values of these adjacent points as Figure 4 depicts.

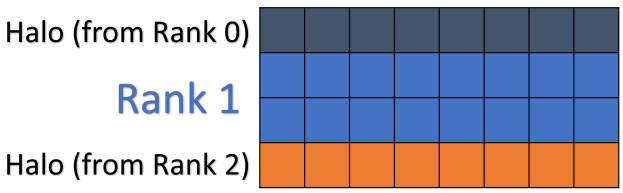


Figure 4: Subdomain with halo.

In this regard, we need to communicate the halos before the stencil computation. For instance, Figure 5 illustrates how rank 0 sends its second row and receive in its third row, the second row of rank 1. All the ranks execute this method, although we are assuming that the domain is not periodic; that is, the top process (rank 0) only sends and receives data from the one under it (rank 1) and the bottom process (rank $size - 1$) only sends and

receives data from the one above it (rank $size - 2$).

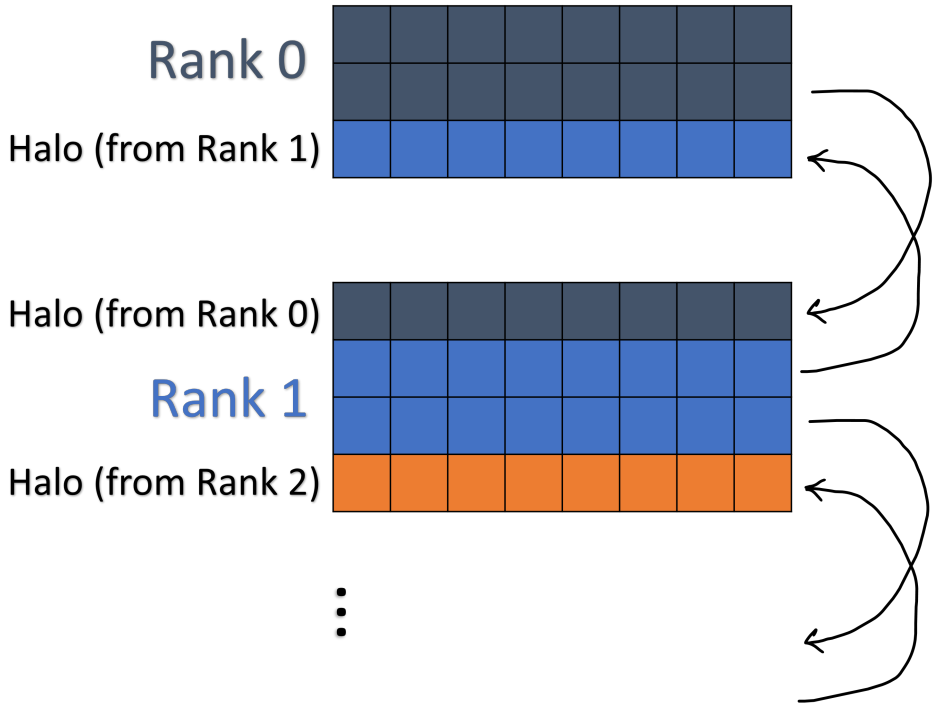


Figure 5: Halo exchange.

-1.00	-1.00	-1.00	-1.00	-1.00	-1.00	-1.00	-1.00
3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00
3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00

Table 1: Rank 3 subdomain initialized

3. Considering, for simplicity, a grid of *double* and the following constraints

- $Rows = 8$
- $Columns = 8$
- $MPI\ processes = 4$
- $Halo\ thickness = 1$

Write a code where each process calculates its own subdomain filled with the rank identifier and -1 in the halos. For instance, rank 3 must be initialized as Table 1 shows. Then, the processes will exchange their halos, and the stencil computation will take place, updating the value position with the average of values of its adjacent. Finally, the original subdomain will be updated with the new values.
Provide:

- A screenshot of the execution showing the status of the grid after:
  - Initialization.
  - Halo exchange.
  - Stencil computation.

  Attach only the grids of the rank corresponding to your `eimtarqso` user numeric part modulo 4.
- Briefly explain your code related to the communication (halo exchange).

5 MPI One-sided Communications

In this section, **we will learn the very basics of one-sided communications** (OSC) in MPI. Without exploring all the possibilities, we will grasp an intuition, and we will be able to write some programs.

OSC functions provide an interface to Remote Memory Access (RMA) communication methods that allow a process to specify all communication parameters for both sending and receiving data. This contrasts with two-sided communication, where both the sender and receiver must participate explicitly.

RMA simplifies the programming model by reducing synchronization and can improve performance by reducing communication overhead and latency (i.e., in irregular communication patterns).

An RMA example could be similar to a chef ringing the bell in a kitchen. She *puts* the dish over the counter and continues cooking. When the waiter is available, he will take the dish to bring it to the table.

RMA communications follow this process:

- **Initialize the communication memory region *window***: Collectively, ranks must define a region of their memory that will be available for the rest. This region is called

window and is handled by the output of this function:

```
int MPI_Win_create(void *base, MPI_Aint size, int
  disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)
```

- **base**: Initial address of window (offset).
- **size**: Size of the window in bytes.
- **disp_unit**: Local unit size for displacements in bytes.
- **info**: Handle to an MPI_Info object.
- **comm**: Handle to a communicator.
- **win** (output): Handle to the window created by the call.

Due to operations within windows are non-blocking, the synchronization has to be defined explicitly.

- **Start a synchronization (*epoch*)**: For a given window, typically, two synchronization calls are necessary: one to initiate a communication epoch and another to complete all communications initiated after the first call, thus ending the epoch.

  There are three RMA synchronization mechanisms, although, in this course, we will focus only on *fence*.

  It is important to note that epochs are defined per window. Consequently, a single process can participate in multiple epochs simultaneously associated with different windows.

```
int MPI_Win_fence(int assert, MPI_Win win)
```

  - **assert**: Conditions for the fence.
  - **win**: Handle to the window.

- **Perform the communication**: In this course, we will learn the autodescriptive RMA operations *get* and *put*. In this regard, we are able to read remote data and to write in a remote memory address, respectively.

```
int MPI_Get(void *origin_addr, int origin_count,
  MPI_Datatype origin_datatype, int target_rank, MPI_Aint
  target_disp, int target_count, MPI_Datatype
  target_datatype, MPI_Win win)
```

  - **origin_addr:** Address of the local buffer to store the data.
  - **origin_count:** The number of entries in the origin buffer.
  - **origin_datatype:** The datatype of the entry.
  - **target_rank:** The rank of the target.
  - **target_disp:** Target offset.
  - **target_count:** Number of entries in the target buffer.
  - **target_datatype:** Datatype of the entry in the target buffer.

- **win**: Handle to the window.

```
int MPI_Put(const void *origin_addr, int origin_count,
  MPI_Datatype origin_datatype, int target_rank, MPI_Aint
  target_disp, int target_count, MPI_Datatype
  target_datatype, MPI_Win win)
```

- **origin_addr:** Address of the local buffer for "sending" the data.
- **origin_count:** The number of entries in the origin buffer.
- **origin_datatype:** The datatype of the entry.
- **target_rank:** The rank of the target.
- **target_disp:** Target offset.
- **target_count:** Number of entries in the target buffer.
- **target_datatype:** Datatype of the entry in the target buffer.
- **win**: Handle to the window.

- **Stop the epoch**: Since communication operations are non-blocking, in order to ensure the completion of the transfer, a synchronization call is required.

  In this course, we only use `MPI_Win_fence` to synchronize ranks.

- **Free the window**: Collectively, ranks have to free the window. This operation requires that all RMA communications on the window are completed, so a synchronization call must be made before this call may be made:

```
int MPI_Win_create(void *base, MPI_Aint size, int
  disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)
```

- **win**: handle to the window.

4. Using RMA, we want to implement a program based on the code in *rma.c* where:

- The odd indexes of "v1" in rank #0 contain the values from rank #1 in those indexes.
- As a result you should print from rank #0 something like: {`value_0_rank0, value_1_rank1, value_2_rank0, value_3_rank1, ...`}
- Use the numeric part of your `eimtarqso` user to define the `USERID` in the code.
- Execute the program only with 2 processes (`mpirun -np 2 ./a.out`).