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

Course: 2022/2023

Semester: 2nd

PAC: 3 - Distributed-memory Parallel Computing with MPI

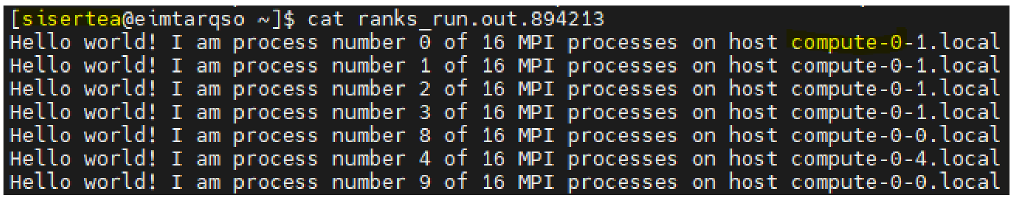
Due Date: 8th May 2023 (24:00h CET)

Submission Format (**THIS IS EXTREMELY IMPORTANT. PLEASE, READ CAREFULLY**): A single PDF containing all the responses including the scripts, screenshots, charts, etc.

Screenshots of executions **will be evaluated only** if they are aligned to the following rules:

- 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
```

Additionally, submitting an archive including files with your implementation is allowed (only ".zip" format.), but keep in mind that only the PDF file will be evaluated.

Grading policy: Brevity and concretion will be highly positive considered to grade the assignment.

Question	Points
1	1
2	1
3	2
4	2
5	2
6	2
Total	10

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.

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`):

```c
#include "mpi.h"
#include <stdio.h>

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    printf("Hello World!\n");

    MPI_Finalize();
    return 0;
}
```

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`):

  ```bash
  #!/bin/bash
  #$ -cwd
  #$ -S /bin/bash
  #$ -N hello_job
  #$ -o $JOB_NAME.out.$JOB_ID
  #$ -e $JOB_NAME.err.$JOB_ID
  #$ -pe orte 6

  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.

1. Attach an screenshot of the execution of the MPI program discussed above, and explain the results.

- What is the difference between the SGE option "-pe orte 6" and the MPI parameter "-np 6"?
- What is the appropriate way of submitting to the queue the program with 2 processes (show your SGE code)?

?

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.

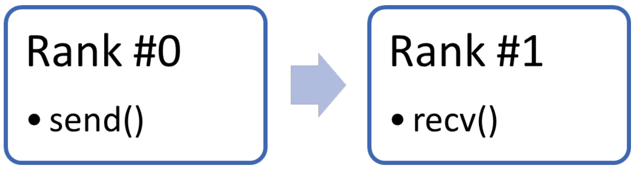
```c
#include "mpi.h"
#include <stdio.h>
#include <unistd.h>

void func1(int rank, char* hostname) {
    printf("Even rank %d on host %s is running function %s\n"
      , rank, hostname, __func__);
}
void func2(int rank, char* hostname) {
    printf("Odd rank %d on host %s is running function %s\n" ,
      rank, hostname, __func__);
}

int main(int argc, char **argv)
{
  int rank, numprocs;
  char hostname[256];

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

  gethostname(hostname,255);

  printf("Hello world! I am process number %d of %d MPI
    processes on host %s\n", rank, numprocs, hostname);

  MPI_Finalize();
  return 0;
}
```

2. What is the result of the execution of the MPI program above (*ranks.c*) with 12 processes? Where did the ranks run? Why?

3 Point to Point 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:

```
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
```

3.1 Blocking and Non-blocking Communications

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 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
```

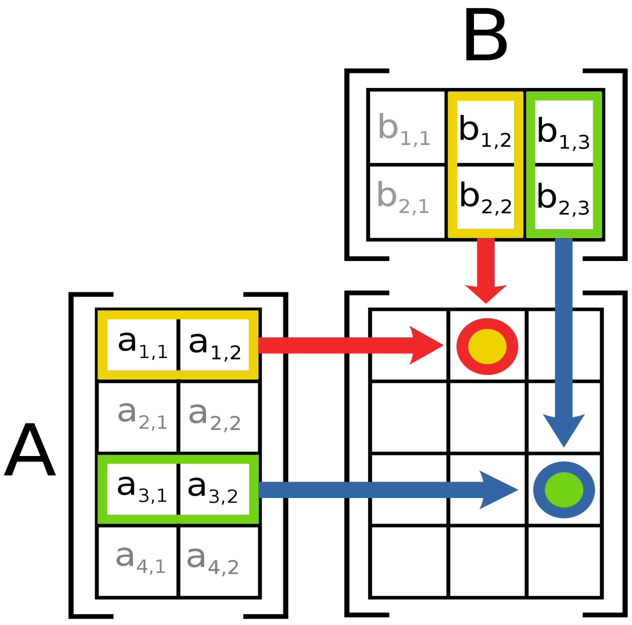
3. With regard to *deadlock.c*:

- Why does it deadlock?
- Can you describe how to prevent the deadlock? Provide and implementation of the previous program that prevents deadlocks. Attach the code, the execution output, and justify it.

?

4 Matrix Multiplication

The dot product operation of two matrices corresponds to the following diagram:



Matrix multiplication diagram.svg:User:BilouSee below., CC BY-SA 3.0 `http://creativecommons.org/licenses/by-sa/3.0`*, via Wikimedia Commons.*

Remember that for two given matrices:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

the result is matrix C with the number of rows of matrix A and the number of columns of the matrix B:

$$\mathbf{C} = \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & a_{11}b_{12} + \cdots + a_{1n}b_{n2} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ a_{21}b_{11} + \cdots + a_{2n}b_{n1} & a_{21}b_{12} + \cdots + a_{2n}b_{n2} & \cdots & a_{21}b_{1p} + \cdots + a_{2n}b_{np} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & a_{m1}b_{12} + \cdots + a_{mn}b_{n2} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{pmatrix}$$

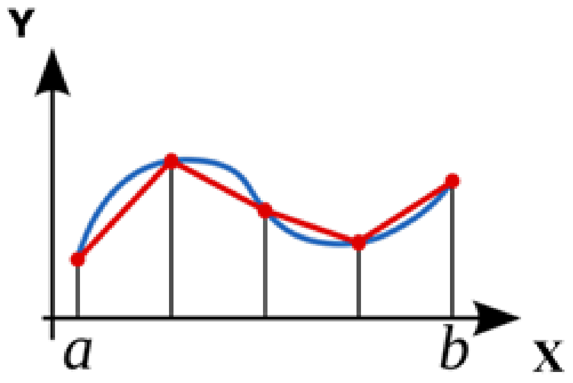
4. Using the code in `matrixmul_template.c`, implement a program, where rank #0 distributes a matrix multiply operation to the rest of ranks.

- Run your code with 8 processes (one producer and seven consumers).
- Briefly explain your code.
- Attach an screenshot of the execution of the prints given in the template.

?

5 Numerical Integration

File *trapezoid.c* provides the code implementing the trapezoidal rule for numerical integration. It is used to approximate the area between the graph of a function y=f(x), two vertical lines, and the X-axis. In the figure below we use n=4 subintervals.



If the endpoints of the subinterval are x_i and x_{i+1}, then the length of the subinterval is h=xi+1–xi. Further, if the lengths of the two vertical segments are $f(x_i)$ and $f(x_{i+1})$, then the area of the trapezoid is: $h/2[f(x_i) + f(x_{i+1})]$. Since we chose the n subintervals so that they would all have the same length, we also know that if the vertical lines bounding the region are $x = a$ and $x = b$, then $h = (b - a)/n$.

If we call the leftmost endpoint x0, and the rightmost endpoint x_n: $x_0 = a$, $x_1 = a + h$, $x_2 = a + 2h, \ldots$, $x_{n-1} = a + (n - 1)h$, $x_n = b$. Thus, the sum of the areas of the trapezoids (total area approximation) is: $h[f(x_0)/2 + f(x_1) + f(x_2) + ... + f(x_{n-1}) + f(x_n)/2]$.

5. Implement a parallel version of *trapezoid.c* using MPI. The serial code provided in *trapezoid.c* uses the fragment function, which can be useful for your implementation.

- Check that your results are equal to the serial version's results.
- Explain your design decisions, paying special attention to the MPI calls.

?

6 MPI + OpenMP

The MPI paradigm is perfectly compatible with intra-node parallelism, for instance, using OpenMP. In our cluster, we can launch jobs combining both approaches. Then, instead of running an MPI process in every "pe", we could reduce the number of processes to one per node, and by default, OpenMP would spread four threads in the node. This code ("hello_world_omp.sge") shows how to request two "exclusive" nodes and initiate an MPI process in each of the assigned nodes:

```
#!/bin/bash
#$ -cwd
#$ -S /bin/bash
#$ -N trapezoid_run
#$ -o $JOB_NAME.out.$JOB_ID
#$ -e $JOB_NAME.err.$JOB_ID
#$ -pe orte 8

cut -f1 -d" " $PE_HOSTFILE > hostfile.$JOB_ID
mpirun -np 2 --hostfile hostfile.$JOB_ID ./hello_mpi_omp
```

The following program implements a "hello world" for each thread of each process ("hello_mpi_omp.c"):

```
#include <omp.h>
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    int rank, numprocs;
    char hostname[256];

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    gethostname(hostname,255);

    int nthreads, tid;
    #pragma omp parallel private(nthreads, tid)
    {
        tid = omp_get_thread_num();
        printf("Hello world! I am thread %d/%d of process
          %d/%d MPI on host %s\n", tid,
          omp_get_num_threads()-1, rank, numprocs-1, hostname);
    }
    MPI_Finalize();
    return 0;
}
```

Giving as a possible result:

```
Hello world! I am thread 1/3 of process 0/1 MPI on host compute-0-1.local
Hello world! I am thread 3/3 of process 0/1 MPI on host compute-0-1.local
Hello world! I am thread 0/3 of process 0/1 MPI on host compute-0-1.local
Hello world! I am thread 2/3 of process 0/1 MPI on host compute-0-1.local
Hello world! I am thread 1/3 of process 1/1 MPI on host compute-0-7.local
Hello world! I am thread 2/3 of process 1/1 MPI on host compute-0-7.local
Hello world! I am thread 3/3 of process 1/1 MPI on host compute-0-7.local
Hello world! I am thread 0/3 of process 1/1 MPI on host compute-0-7.local
```

In order to compile the program, we could use:

```
mpicc -fopenmp hello_mpi_omp.c -o hello_mpi_omp
```

6. With regard to "trapezoid.c":

- Provide a MPI+OpenMP implementation of the code.
- Explain your design decisions (especially those related to `#pragma`).
- Study the performance with different threads/processes configuration, and compare it to the only MPI version. You may use the following template table to plan your experiments ("ppn" stands for "processes per node"):

#nodes	1ppn	2ppn	3ppn	4ppn
1				
2				
3				
4				
5				
6				
7				
8				

Feel free to use charts or other tables to represent and explain your results. Notice that continuing to increase the number of nodes may not be necessary in your case at some point.

?