PEC 3

Pablo Riutort Grande

May 8, 2023

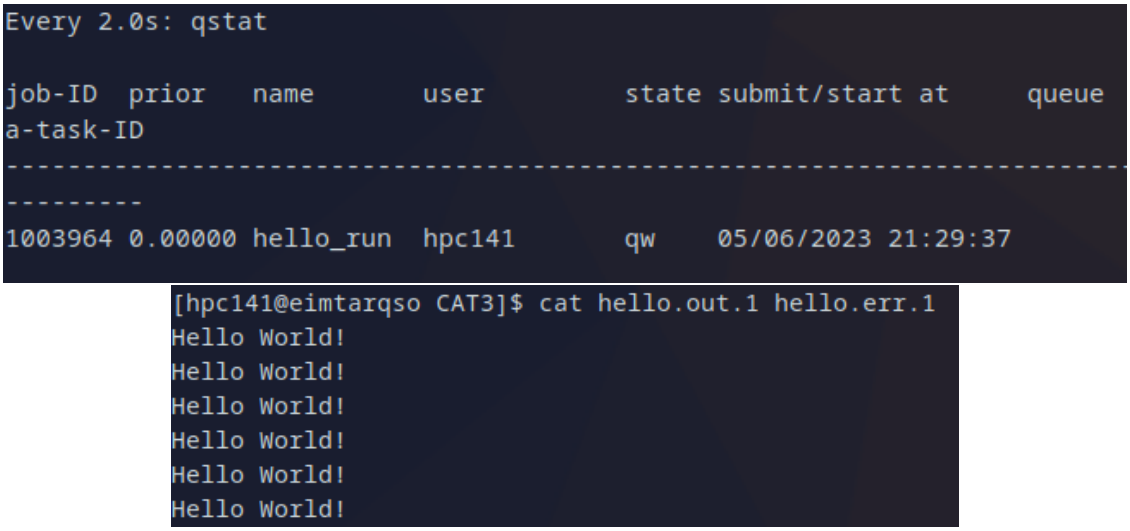
M0.538 - HIGH PERFORMANCE COMPUTING

MU Ingeniería Informática / MU Ingeniería Computacional y Matemática

Estudios de Informática, Multimedia y Telecomunicación

1 MPI Execution Environment

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



Submit ./hello_world to queue

With the provided SGE script we have:

- #$ -pe orte 6: requests the allocation of 6 slots (or processes) for the job using the "orte" parallel environment (Open MPI).
- mpirun -np 6 ./hello executes the "hello" program with 6 processes using the mpirun command.

Each process prints "Hello World!" as output, as expected.

- **What is the difference between the SGE option "-pe orte 6" and the MPI parameter "-np 6"?** "-pe orte" is an SGE option to request a parallel environment, with "orte" we give a specific environment within SGE to work with Open MPI and the number will be the slots to request. "-np", on the other hand, refers to the number of processes to launch by MPI.
- **What is the appropriate way of submitting to the queue the program with 2 processes (show your SGE code)?**

```
#!/bin/bash
#$ -cwd
#$ -S /bin/bash
#$ -N hello_run
#$ -o $JOB_NAME.out.$JOB_ID
#$ -e $JOB_NAME.err.$JOB_ID
#$ -pe orte 2

mpirun -np 2 ./hello
```

Listing 1: Submitting hello_world.c with 2 MPI processes

2 MPI Processes

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

```
#!/bin/bash
#$ -cwd
#$ -S /bin/bash
#$ -N ranks_run
#$ -o ranks.out.12
#$ -e ranks.err.12
#$ -pe orte 12

mpirun -np 12 ./ranks
```

Listing 2: Submitting ranks into the queue with 12 slots

```
[hpc141@eimtarqso 2]$ cat ranks.out.12
Hello world! I am process number 0 of 12 MPI processes on host compute-0-0.local
Hello world! I am process number 1 of 12 MPI processes on host compute-0-0.local
Hello world! I am process number 2 of 12 MPI processes on host compute-0-0.local
Hello world! I am process number 3 of 12 MPI processes on host compute-0-0.local
Hello world! I am process number 8 of 12 MPI processes on host compute-0-2.local
Hello world! I am process number 4 of 12 MPI processes on host compute-0-7.local
Hello world! I am process number 5 of 12 MPI processes on host compute-0-7.local
Hello world! I am process number 9 of 12 MPI processes on host compute-0-2.local
Hello world! I am process number 6 of 12 MPI processes on host compute-0-7.local
Hello world! I am process number 10 of 12 MPI processes on host compute-0-2.local
Hello world! I am process number 11 of 12 MPI processes on host compute-0-2.local
Hello world! I am process number 7 of 12 MPI processes on host compute-0-7.local
```

./ranks submission to the queue with 12 slots

Processes with ranks 0 to 3 are running on the host "compute-0-0.local", processes with ranks 4, 5, 6, and 7 are running on the host "compute-0-7.local" and processes with ranks 8, 9, 10, and 11 are running on the host "compute-0-2.local". The specific distribution of processes across different hosts depends on the configuration and availability of resources in the cluster and the scheduling algorithm of the resource manager.

This distribution may change depending on many factors (cluster's load, resources, etc.), for instance, a change in the SGE script to indicate to use 2 slots (Listing 3) may give the next results Fig. **??**.

```
#$ -pe orte 2

mpirun -np 12 ./ranks
```

Listing 3: Submitting ranks into the queue with 2 slots

The mismatch between the requested slots (2) and the number of processes (12) causes the processes to be oversubscribed on the available slots. As a result, the processes are all executed on "compute-0-6.local", since it is the only host available.

```
[hpc141@eimtarqso 2]$ cat ranks.out.2
Hello world! I am process number 0 of 12 MPI processes on host compute-0-6.local
Hello world! I am process number 2 of 12 MPI processes on host compute-0-6.local
Hello world! I am process number 3 of 12 MPI processes on host compute-0-6.local
Hello world! I am process number 10 of 12 MPI processes on host compute-0-6.local
Hello world! I am process number 7 of 12 MPI processes on host compute-0-6.local
Hello world! I am process number 5 of 12 MPI processes on host compute-0-6.local
Hello world! I am process number 8 of 12 MPI processes on host compute-0-6.local
Hello world! I am process number 1 of 12 MPI processes on host compute-0-6.local
Hello world! I am process number 4 of 12 MPI processes on host compute-0-6.local
Hello world! I am process number 6 of 12 MPI processes on host compute-0-6.local
Hello world! I am process number 9 of 12 MPI processes on host compute-0-6.local
Hello world! I am process number 11 of 12 MPI processes on host compute-0-6.local
```

./ranks submission to the queue with 2 slots

3 Point to Point Communication

With regard to deadlock.c:

- **Why does it deadlock?** Deadlock occurs when both processes (rank 0 and rank 1) execute the send operation first and then attempt to receive and wait for a message that will never arrive.
- **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.**

One possible solution is to use non-blocking communication with "MPI_Isend"(1), "MPI_Irecv"(2) and "MPI_Wait"(3) functions.

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

#define SIZE 100000

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("Rank %d is reported\n", rank);
    if (rank == 0) printf("Rank size: %d\n", size);
        int v1[SIZE];
        int v2[SIZE];
    // request input and status output for MPI_Wait function
    MPI_Request request;
    MPI_Status status;
    if (rank == 0) {
        int partner = 1;
        MPI_Isend(v1, SIZE, MPI_INT, partner, 100, MPI_COMM_WORLD, &request);
        printf("Rank %d sends to %d\n", rank, partner);
        MPI_Irecv(v2, SIZE, MPI_INT, partner, 100, MPI_COMM_WORLD, &request);
        printf("Rank %d receives from %d\n", rank, partner);
        MPI_Wait(&request, &status);
```

```
    }
    else if (rank == 1) {
        int partner = 0;
        MPI_Isend(v1, SIZE, MPI_INT, partner, 100, MPI_COMM_WORLD, &request);
        printf("Rank %d sends to %d\n", rank, partner);
        MPI_Irecv(v2, SIZE, MPI_INT, partner, 100, MPI_COMM_WORLD, &request);
        printf("Rank %d receives from %d\n", rank, partner);
        MPI_Wait(&request, &status);
    }

    MPI_Finalize();
}
```

Listing 4: ideadlock.c: Use of non-blocking function for deadlock prevention

```
#!/bin/bash
#$ -cwd
#$ -S /bin/bash
#$ -N ideadlock_run
#$ -o ideadlock.out.2
#$ -e ideadlock.err.2
#$ -pe orte 2

mpirun -np 2 ./ideadlock
```

Listing 5: SGE script on deadlock prevention

```
job-ID  prior   name       user         state submit/start at     queue                          slots ja-task-ID
-----------------------------------------------------------------------------------------------------------------
1004348 0.50756 ideadlock_ hpc141       r     05/07/2023 17:50:50 all.q@compute-0-0.local            2
```

```
[hpc141@eimtarqso 3]$ cat ideadlock.out.2
Rank 0 is reported
Rank size: 2
Rank 0 sends to 1
Rank 0 receives from 1
Rank 1 is reported
Rank 1 sends to 0
Rank 1 receives from 0
Rank 0 is reported
Rank size: 2
Rank 0 sends to 1
Rank 0 receives from 1
Rank 1 is reported
Rank 1 sends to 0
Rank 1 receives from 0
```

./ideadlock submission to the queue with 2 slots

"MPI_Isend" starts a standard-mode, nonblocking send, "MPI_Irecv" will start a nonblocking receive as well and "MPI_Waits" waits for an MPI send or receive to complete.

4 Matrix multiplication

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

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

#define NRA 1024                 /* number of rows in matrix A */
#define NCA 512                 /* number of columns in matrix A */
#define NCB 512                  /* number of columns in matrix B */

int main (int argc, char *argv[]) {
  int nranks,              /* number of tasks in partition */
    myrank,                /* a task identifier */
    rc;
  double  a[NRA][NCA],           /* matrix A to be multiplied */
    b[NCA][NCB],           /* matrix B to be multiplied */
    c[NRA][NCB];           /* result matrix C */
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &nranks);

  if (nranks < 2) {
    printf("Need at least two MPI ranks. Quitting...\n");
    MPI_Abort(MPI_COMM_WORLD, rc);
    exit(1);
  }

  int i, j;
  if (myrank == 0) {
    printf("MPI has started with %d tasks.\n", nranks);
    for (i = 0; i < NRA; i++)
      for (j = 0; j < NCA; j++)
        a[i][j] = i + j;
    for (i = 0; i < NCA; i++)
      for (j = 0; j < NCB; j++)
        b[i][j] = i * j;

    /* Send matrix data to the worker tasks */
    int rows, dest, offset;
    int chunk = NRA / (nranks - 1);
    for (dest = 1, offset = 0; dest <= nranks - 1; dest++) {
      rows = (dest != nranks - 1) ? chunk : (NRA - offset);
      printf("Sending %d rows to task %d offset=%d\n", rows, dest, offset);
      MPI_Send(&offset, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
      MPI_Send(&rows, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
      MPI_Send(&a[offset][0], rows * NCA, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
      MPI_Send(&b, NCA * NCB, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
      offset += rows;
    }

    /* Receive results from worker tasks */
    for (i = 1; i <= nranks - 1; i++) {
      MPI_Recv(&offset, 1, MPI_INT, i, 1, MPI_COMM_WORLD, &status);
      MPI_Recv(&rows, 1, MPI_INT, i, 1, MPI_COMM_WORLD, &status);
```

```
      MPI_Recv(&c[offset][0], rows * NCB, MPI_DOUBLE, i, 1, MPI_COMM_WORLD, &
    status);
      printf("Received results from task %d\n", i);
    }

    printf("Done.\n");
  }

  if (myrank > 0) {
    int offset, rows;
    MPI_Recv(&offset, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    MPI_Recv(&a, rows * NCA, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
        MPI_Recv(&b, NCA * NCB, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);

    /* Perform matrix multiplication */
    for (i = 0; i < rows; i++) {
      for (j = 0; j < NCB; j++) {
        c[i][j] = 0.0;
        for (int k = 0; k < NCA; k++) {
          c[i][j] += a[i][k] * b[k][j];
        }
      }
    }

    /* Send the results back to rank 0 */
    MPI_Send(&offset, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    MPI_Send(&c, rows * NCB, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
  }

  MPI_Finalize();
  return 0;
}


```

Listing 6: Matrix multiply operation across rest of ranks

- **Run your code with 8 processes (one producer and seven consumers).**

```
#!/bin/bash
#$ -cwd
#$ -S /bin/bash
#$ -N matrix_multiply
#$ -o matrix.out
#$ -e matrix.err
#$ -pe orte 8

# Run the MPI program
mpirun -np 8 ./matrix_multiply
```

Listing 7: SGE script for matrixmul

- **Briefly explain your code.**
  This code demonstrates a parallel matrix multiplication using MPI (Message Passing Interface).
  If the rank is 0 (the producer), it initializes matrices A and B with some values. Then, the

```
job-ID  prior   name       user         state submit/start at     queue                          slots ja-task-ID
-----------------------------------------------------------------------------------------------------------------
1004476 0.52295 matrix_mul hpc141       qw    05/07/2023 20:04:44                                 8
```

Sending code into queue

producer sends data to worker tasks by dividing the workload.
It calculates the number of rows each worker will process based on the rank and the total number of ranks. Sends the offset (starting row index), number of rows, matrix A, and matrix B to each worker using MPI_Send.
After sending the data, the producer receives the results from each worker with MPI_Recv.
For the workers (with rank > 0), each receives the offset, number of rows and the matrices using MPI_Recv and performs the matrix multiplication for its assigned rows of matrix A (calculated with offset) and matrix B (in full), and stores the result in matrix C.

- **Attach an screenshot of the execution of the prints given in the template.**

```
[hpc141@eimtarqso 4]$ cat matrix.out
MPI has started with 8 tasks.
Sending 146 rows to task 1 offset=0
Sending 146 rows to task 2 offset=146
Sending 146 rows to task 3 offset=292
Sending 146 rows to task 4 offset=438
Sending 146 rows to task 5 offset=584
Sending 146 rows to task 6 offset=730
Sending 148 rows to task 7 offset=876
Received results from task 1
Received results from task 2
Received results from task 3
Received results from task 4
Received results from task 5
Received results from task 6
Received results from task 7
Done.
```

Execution of the ./matrixmul program into the queue

5 Numerical Integration

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.

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

double f(double x) {
    return (4.0 / (1.0 + x * x));
}
```

```
8
9  double fragment(double a, double b, long long unsigned int num_fragments, double h
       ) {
10     double est, x;
11     long long unsigned int i;
12
13     est = (f(a) + f(b)) / 2.0;
14     for (i = 1; i <= num_fragments - 1; i++) {
15         x = a + i * h;
16         est += f(x);
17     }
18     est = est * h;
19
20     return est;
21 }
22
23 int main(int argc, char** argv) {
24
25     int rank, size;
26     long long unsigned int n = 100000000000;
27     double a = 0.0, b = 1.0, h = 0.0;
28     double result = 0.0, total_result = 0.0;
29
30     MPI_Init(&argc, &argv);
31     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
32     MPI_Comm_size(MPI_COMM_WORLD, &size);
33
34     h = (b - a) / n;
35
36     long long unsigned int local_n = n / size;
37     double local_a = a + rank * local_n * h;
38     double local_b = local_a + local_n * h;
39     double local_result = fragment(local_a, local_b, local_n, h);
40
41     MPI_Reduce(&local_result, &total_result, 1, MPI_DOUBLE, MPI_SUM, 0,
       MPI_COMM_WORLD);
42
43     if (rank == 0) {
44         printf("Result: %.20f\n", total_result);
45     }
46
47     MPI_Finalize();
48
49     exit(0);
50 }
```

Listing 8: Parallel version of trapezoid.c

- **Check that your results are equal to the serial version's results.** First execution of this exercise was naive and did not take into account a 60 min. timeout so after a first execution the output file was empty: In order to avoid this caveat, the serial trapezoid program

```
[hpc141@eimtarqso 5]$ cat trapezoid.out
[hpc141@eimtarqso 5]$ _
```

Execution of the serial verion of trapezoid.c in the queue

was edited by reducing its "n" param in order of 5 [9].

```
//long long unsigned int n=100000000000;
//Reduce n in order of 5
long long unsigned int n=1000000;
```

Listing 9: Trapezoid edit "n"

With this change we can see some results in the output file after seding the program to the queue [10] [??].

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

mpirun -np 8 ./trapezoid
```

Listing 10: SGE script for serial trapezoid

```
[hpc141@eimtarqso 5]$ cat trapezoid.out
Result: 3.141592653589792671f
Result: 3.141592653589792671f
Result: 3.141592653589792671f
Result: 3.141592653589792671f
Result: 3.141592653589792671f
Result: 3.141592653589792671f
Result: 3.141592653589792671f
Result: 3.141592653589792671f
[hpc141@eimtarqso 5]$
```

Execution of the corrected serial verion of trapezoid.c in the queue

When we compare this results with the parallel version we see they are the same [??].

```
[hpc141@eimtarqso 5]$ cat trapezoid_parallel.out
Result: 3.14159265359416650654
```

Execution of parallel verion of trapezoid.c in the queue

- **Explain your design decisions, paying special attention to the MPI calls.** For this version we have several MPI calls
  - MPI_Init: Initializes the MPI environment as we saw erarlier [8].
  - MPI_Comm_rank and MPI_Comm_size also initializes the MPI environment by determining a rank of the processes and a size associated to the communicator group. This is needed for the processes to communicate [6, 7]
  - MPI_Reduce: After the computation of the local fragment this function reduces the local results into a total result. "MPI_SUM" will calculate the sum of all locals across processes [5].

– MPI_Finalize: Terminates the execution environment [4].

With these MPI calls we can divide the computation among the processes. Each process is assigned a local range to compute its own fragment of the integral. The workload is distributed evenly among the processes (local_n = n / size) where "n" is the total number of fragments.

6 MPI + OpenMP

With regard to "trapezoid.c":

- **Provide a MPI+OpenMP implementation of the code.**

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

double f(double x) {
    return (4.0 / (1.0 + x * x));
}

double fragment(double a, double b, long long unsigned int num_fragments,
    double h) {
    double est, x;
    long long unsigned int i;

    est = (f(a) + f(b)) / 2.0;

    #pragma omp parallel for reduction(+:est)
    for (i = 1; i <= num_fragments - 1; i++) {
        x = a + i * h;
        est += f(x);
    }

    return est * h;
}

int main(int argc, char** argv) {

    int rank, size;
    long long unsigned int n = 100000000000;
    double a = 0.0, b = 1.0, h = 0.0;
    double result = 0.0, total_result = 0.0;

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

    h = (b - a) / n;

    long long unsigned int local_n = n / size;
    double local_a = a + rank * local_n * h;
    double local_b = local_a + local_n * h;
    double local_result = fragment(local_a, local_b, local_n, h);

    MPI_Reduce(&local_result, &total_result, 1, MPI_DOUBLE, MPI_SUM, 0,
    MPI_COMM_WORLD);

```

```
    if (rank == 0) {
        printf("Result: %.20f\n", total_result);
    }

    MPI_Finalize();

    exit(0);
}
```

Listing 11: MPI+OpenMP implementation

```
#!/bin/bash
#$ -cwd
#$ -S /bin/bash
#$ -N trapezoid_$SLOTS_$THREADS
#$ -o trapezoid_$SLOTS_$THREADS.out
#$ -e trapezoid_$SLOTS_$THREADS.err
#$ -pe orte $SLOTS
#$ -pe openmp $THREADS

export OMP_NUM_THREADS=$THREADS
/usr/bin/time -f %e -o results/$SLOTS_$THREADS.txt mpirun -np $SLOTS -x
    OMP_NUM_THREADS ./trapezoid_parallel_openmp
```

Listing 12: Base SGE script

```
#!/bin/sh

for slots in {1,2,3,4,5,6,7,8}; do
for threads in {1,2,3,4}; do
  sge=trapezoid_parallel.sge
  cp $sge aux.sge
  aux=aux.sge
  sed -i 's/$THREADS/'$threads'/g' $aux
  sed -i 's/$SLOTS/'$slots'/g' $aux
  qsub $aux
done;
done;
```

Listing 13: run.sh to build SGE scripts

- **Explain your design decisions (especially those related to #pragma).**
  The code is basically the same as before (trapezoid with parallel implementation) but a new pragma sentence was added in order to make use of OpenMP.
  "#pragma omp parallel for" is chosen to parallelize the "for" loop across threads. "reduction(+:est)" indicates the variable "est" needs to be reduced accross all threads with addition operation.

```
    #pragma omp parallel for reduction(+:est)
    for (i = 1; i <= num_fragments - 1; i++) {
```

Listing 14: Pragma added in for loop

  This adding allows the program to parallelize the for loop and distribute the work across nodes with the parallel implementation in previous exercise.

- **Study the performance with different threads/processes configuration, and compare it to the only MPI version.**

#nodes	1ppn	2ppn	3ppn	4ppn
1	0.06	0.08	0.08	0.09
2	0.04	0.07	0.07	0.07
3	0.03	0.06	0.07	0.09
4	0.03	0.06	0.08	0.08
5	0.06	0.07	0.19	0.12
6	0.04	0.19	0.11	0.13
7	0.05	0.09	0.24	0.14
8	0.06	0.10	0.13	0.18

Table 1: Performance experiments for processes per node (ppn)

With the script 12 we are able to store the execution times of each execution of the program in the queue, in Table 1 we leave the results of each.

7 References

1. Open MPI. (2022). *MPI_Isend(3): Starts a nonblocking send.* Retrieved from https://www.open-mpi.org/doc/v4.0/man3/MPI_Isend.3.php
2. Open MPI. (2023). *MPI_Irecv(3): Starts a nonblocking receive.* Retrieved from https://www.open-mpi.org/doc/v4.1/man3/MPI_Irecv.3.php
3. Open MPI. (2021). *MPI_Wait(3): Waits for an MPI request to complete.* Retrieved from https://www.open-mpi.org/doc/v3.1/man3/MPI_Wait.3.php
4. Open MPI. (2022). *MPI_Finalize(3): Terminates MPI execution environment.* Retrieved from https://www.open-mpi.org/doc/v4.0/man3/MPI_Finalize.3.php
5. Open MPI. (2023). *MPI_Reduce(3): Reduces values on all processes in a communicator.* Retrieved from https://www.open-mpi.org/doc/v4.1/man3/MPI_Reduce.3.php
6. Open MPI. (2022). *MPI_Comm_rank(3): Determines the rank of the calling process in the communicator.* Retrieved from https://www.open-mpi.org/doc/v4.0/man3/MPI_Comm_rank.3.php
7. Open MPI. (2021). *MPI_Comm_size(3): Determines the size of the group associated with a communicator.* Retrieved from https://www.open-mpi.org/doc/v3.1/man3/MPI_Comm_size.3.php
8. Open MPI. (2021). *MPI_Init(3): Initializes the MPI execution environment.* Retrieved from https://www.open-mpi.org/doc/v3.1/man3/MPI_Init.3.php