UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

PARALLEL AND DISTRIBUTED COMPUTING

2022/2023

Professori	*Studente*
Giuliano Laccetti	Vincenzo Tramo
Anna Mele	

Indice

Esercizio uno

1 Esercizio uno - Grid Printer

1.1 Obiettivo

Dati P processi e un numero positivo p, crea una griglia di $p \times q$ processi dove $p \times q = P$ in cui ogni processo stampa le proprie coordinate sullo standard output. Precondizioni:

- $p <= P$

1.2 Codice sorgente C

```c
#include <mpi.h>
#include <stdio.h>
#include <stdbool.h>
#include <malloc.h>
#include <stdlib.h>
#include <ctype.h>

/*
 *  GRID PRINTER - EXERCISE 1
 *
 *  Given P processes and a positive integer p, creates a grid of p x q
 *  processes, where p x q = P, in which each process prints its
 *  coordinates on standard output.
 *
 *  Preconditions:
 *  - p <= P
 */

int get_number_of_comm_grid_rows_command_line(char **argv, unsigned int total_processes);
MPI_Comm *create_bidimensional_grid(unsigned int rows, unsigned int columns);
bool is_number(const char *s);

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

    int number_of_processes;
    MPI_Comm_size(MPI_COMM_WORLD, &number_of_processes);

    int pid_comm_world;
    MPI_Comm_rank(MPI_COMM_WORLD, &pid_comm_world);

    int p = (pid_comm_world == 0)
        ? get_number_of_comm_grid_rows_command_line(argv, number_of_processes)
        : 0;

    MPI_Bcast(&p, 1, MPI_INT, 0, MPI_COMM_WORLD);

    const int q = number_of_processes / p;

    const MPI_Comm *comm_grid = create_bidimensional_grid(p, q);

    int pid_comm_grid;
    MPI_Comm_rank(*comm_grid, &pid_comm_grid);

    int *coordinates = (int*)malloc(sizeof(int) * 2);
    MPI_Cart_coords(*comm_grid, pid_comm_grid, 2, coordinates);

    printf("[PROCESSOR %d] Grid coordinates (%d, %d)\n", pid_comm_grid, coordinates[0], coordinates[1])
```

```c
    MPI_Finalize();
    return 0;
}

MPI_Comm *create_bidimensional_grid(
    const unsigned int rows,
    const unsigned int columns
) {
    static const int N_DIMS = 2;
    MPI_Comm *comm_grid = (MPI_Comm*) malloc(sizeof(MPI_Comm));

    int *dims = (int*)malloc(sizeof(int) * N_DIMS);
    dims[0] = (int)rows;
    dims[1] = (int)columns;

    const int *periods = (int*)calloc(N_DIMS, sizeof(int));
    const int reorder = 0;

    MPI_Cart_create(MPI_COMM_WORLD, N_DIMS, dims, periods, reorder, comm_grid);
    return comm_grid;
}

int get_number_of_comm_grid_rows_command_line(char **argv, const unsigned int total_processes) {
    const static int GRID_COMM_ROWS_ARGV_INDEX = 2;
    const static char *GRID_COMM_ROWS_ARGV_NAME = "<grid_comm_rows>";
    if (!is_number(argv[GRID_COMM_ROWS_ARGV_INDEX])) {
        fprintf(stderr, "%s must be a positive number!\n", GRID_COMM_ROWS_ARGV_NAME);
        exit(EXIT_FAILURE);
    }
    int grid_comm_rows = atoi(argv[GRID_COMM_ROWS_ARGV_INDEX]);
    if (grid_comm_rows < 1) {
        fprintf(stderr, "%s must be greater than one!\n", GRID_COMM_ROWS_ARGV_NAME);
        exit(EXIT_FAILURE);
    }
    if (grid_comm_rows > total_processes) {
        fprintf(
                stderr,
                "%s can't be greater than the number of processes %d!\n",
                GRID_COMM_ROWS_ARGV_NAME, total_processes
        );
        MPI_Finalize();
        exit(EXIT_FAILURE);
    }
    return grid_comm_rows;
}

bool is_number(const char *s) {
    if (s == NULL) return false;
    for (int i = 0; s[i] != '\0'; i++) {
        if (i == 0 && s[i] == '-' && isdigit(s[i + 1])) continue;
        if (!isdigit(s[i])) {
            return false;
        }
    }
    return true;
}
```

1.3 Lo Script `start_pbs.sh`

```
#!/bin/bash

if [ "$#" -ne 2 ]; then
  echo "Correct usage: $0 <processes> <grid_comm_rows>"
  exit 1
fi

N_CPU=$1
COMM_GRID_ROWS=$2

qsub -v N_CPU="$N_CPU",COMM_GRID_ROWS="$COMM_GRID_ROWS" ./grid_printer.pbs
```

1.4 Il File PBS

```
#!/bin/bash

#PBS -q studenti
#PBS -l nodes=8:ppn=8
#PBS -N grid_printer
#PBS -o grid_printer.out
#PBS -e grid_printer.err

sort -u $PBS_NODEFILE > hostlist
NCPU=$(wc -l hostlist)

echo ----------------------------
echo 'This job is allocated on '${NCPU}' cpu(s)'
echo 'Job is running on node(s):'
cat hostlist
PBS_O_WORKDIR=$PBS_O_HOME/mpi_cartesian_topology_exercises/exercise1
echo ------------------------------------------
echo PBS: qsub is running on $PBS_O_HOST
echo PBS: originating queue is $PBS_O_QUEUE
echo PBS: executing queue is $PBS_QUEUE
echo PBS: working directory is $PBS_O_WORKDIR
echo PBS: execution mode is $PBS_ENVIRONMENT
echo PBS: job identifier is $PBS_JOBID
echo PBS: job name is $PBS_JOBNAME
echo PBS: node file is $PBS_NODEFILE
echo PBS: current home directory is $PBS_O_HOME
echo PBS: PATH = $PBS_O_PATH
echo -----------------------------------------

/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/grid_printer \
$PBS_O_WORKDIR/grid_printer.c -std=c99
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -n $N_CPU \
$PBS_O_WORKDIR/grid_printer $COMM_GRID_ROWS
```

1.5 Esempio di output

Eseguendo il programma con lo script `start_pbs.sh` ponendo il numero di processi uguale a 8 e le numero di righe della topologia cartesiana di processi uguale a 2:

```
$ ./start_pbs.sh 8 2
```

otteniamo il seguende output:

```
----------------------------
This job is allocated on 8 hostlist cpu(s)
Job is running on node(s):
wn273.scope.unina.it
```

```
wn274.scope.unina.it
wn275.scope.unina.it
wn276.scope.unina.it
wn277.scope.unina.it
wn278.scope.unina.it
wn279.scope.unina.it
wn280.scope.unina.it
------------------------------------------
PBS: qsub is running on ui-studenti.scope.unina.it
PBS: originating queue is studenti
PBS: executing queue is studenti
PBS: working directory is /homes/DMA/PDC/2022/TRMVCN99Y/mpi_cartesian_topology_exercises/exercise1
PBS: execution mode is PBS_BATCH
PBS: job identifier is 4006272.torque02.scope.unina.it
PBS: job name is grid_printer
PBS: node file is /var/spool/pbs/aux//4006272.torque02.scope.unina.it
PBS: current home directory is /homes/DMA/PDC/2022/TRMVCN99Y
PBS: PATH = /usr/lib64/openmpi/1.2.7-gcc/bin:/usr/kerberos/bin:/opt/exp_soft/unina.it/intel/composer_xe
-----------------------------------------
[PROCESSOR 7] Grid coordinates (1, 3)
[PROCESSOR 1] Grid coordinates (0, 1)
[PROCESSOR 0] Grid coordinates (0, 0)
[PROCESSOR 2] Grid coordinates (0, 2)
[PROCESSOR 6] Grid coordinates (1, 2)
[PROCESSOR 4] Grid coordinates (1, 0)
[PROCESSOR 3] Grid coordinates (0, 3)
[PROCESSOR 5] Grid coordinates (1, 1)
```

Esercizio due

2 Esercizio due - Uniform Distributor Numbers

2.1 Obiettivo

Dati P processi e un vettore V di dimensione N, distribuisce equamente gli elementi di V tra i processi. Precondizioni:

- $N >= P$

2.2 Differenti versioni

Sono state implementate differenti versioni dello stesso algoritmo, precisamente tre che si differenziano su come la comunicazione avviene:

- **Versione 1**: viene utilizzato `MPI_Scatter()`

- **Versione 2**: viene utilizzato `MPI_ISend()` e `MPI`$_Recv()$**`Versione 3`** : $vieneutilizzato$`MPI_Scatterv()`

2.3 Codice sorgente C

```c
#include <mpi.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>
#include <stdbool.h>
#include <ctype.h>

/*
 *  UNIFORM DISTRIBUTOR NUMBERS - EXERCISE 2
 *
 *  Given P processes and a vector V of dimension N, equally distributes the elements of V among the pr
 *
 *  Preconditions:
 *  - N >= P
 */

#define MAX_RANDOM_NUMBER 11

enum {
    UNIFORM_DISTRIBUTOR_NUMBERS_V1 = 1,
    UNIFORM_DISTRIBUTOR_NUMBERS_V2 = 2,
    UNIFORM_DISTRIBUTOR_NUMBERS_V3 = 3,
};

typedef struct {
    int size;
    int *data;
} IntVector;

/* VERSION 1: with MPI_Scatter */
IntVector* uniform_distributor_numbers_v1(MPI_Comm comm, int *numbers, int total_numbers);

/* VERSION 2: with MPI_Isend() and MPI_Recv() */
IntVector* uniform_distributor_numbers_v2(MPI_Comm comm, int *numbers, int total_numbers);

/* VERSION 3: with MPI_Scatterv() */
IntVector* uniform_distributor_numbers_v3(MPI_Comm comm, int *numbers, int total_numbers);

int* generate_random_numbers(unsigned int total, unsigned int max);
```

```c
void print_vector(IntVector *vector);
void print_vector_s(int *vector, int size);
void check_arguments(int argc, char **argv);
bool is_number(const char *s);
const int get_size_vector_command_line(char **argv, int total_processes);
const int get_uniform_distributor_strategy_command_line(char **argv);

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

    int total_processes;
    MPI_Comm_size(MPI_COMM_WORLD, &total_processes);

    int this_pid;
    MPI_Comm_rank(MPI_COMM_WORLD, &this_pid);

    const int size_vector = get_size_vector_command_line(argv, total_processes);
    int *global_vector = generate_random_numbers(size_vector, MAX_RANDOM_NUMBER);

    if (this_pid == 0) {
        printf("Global Vector:\n - Size: %d\n - Numbers: ", size_vector);
        print_vector_s(global_vector, size_vector);
    }

    const int strategy = get_uniform_distributor_strategy_command_line(argv);
    IntVector *local_vector = NULL;

    MPI_Barrier(MPI_COMM_WORLD);
    const double t0 = MPI_Wtime();

    switch (strategy) {
        case UNIFORM_DISTRIBUTOR_NUMBERS_V1:
            local_vector = uniform_distributor_numbers_v1(MPI_COMM_WORLD, global_vector, size_vector);
            break;
        case UNIFORM_DISTRIBUTOR_NUMBERS_V2:
            local_vector = uniform_distributor_numbers_v2(MPI_COMM_WORLD, global_vector, size_vector);
            break;
        case UNIFORM_DISTRIBUTOR_NUMBERS_V3:
            local_vector = uniform_distributor_numbers_v3(MPI_COMM_WORLD, global_vector, size_vector);
            break;
    }

    const double t1 = MPI_Wtime();

    double total_time = t1 - t0;
    double max_total_time;
    MPI_Reduce(&total_time, &max_total_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

    if (this_pid == 0) {
        printf("[TOTAL TIME] %e seconds.\n", max_total_time);
    }

    printf("[PID %d]: ", this_pid);
    print_vector(local_vector);

    MPI_Finalize();

    return 0;
}
```

```c
/* with MPI_Scatter */
IntVector* uniform_distributor_numbers_v1(MPI_Comm comm, int *numbers, int total_numbers) {
    int total_processes;
    MPI_Comm_size(comm, &total_processes);

    int this_pid;
    MPI_Comm_rank(comm, &this_pid);

    const int number_of_locations_per_process = total_numbers / total_processes;
    const int rest = total_numbers % total_processes;

    const int linear_vector_size = total_numbers + (rest != 0 ? (total_processes - rest) : 0);
    int *linear_vector = (int*)calloc(linear_vector_size, sizeof(int));
    int offset_linear_vector = 0;

    if (this_pid == 0) {
        for (int pid = 0; pid < total_processes; pid++) {
            int offset_rest_start = 0, offset_rest_end = 0;

            if (rest != 0) {
                if (pid == 0) {
                    offset_rest_end = 1;
                } else if (pid <= rest) {
                    offset_rest_start = pid;
                    offset_rest_end = ((pid < rest) ? 1 : 0);
                } else {
                    offset_rest_start = rest;
                    offset_linear_vector++;
                }
            }

            const int start_index = (pid * number_of_locations_per_process) + offset_rest_start;
            const int end_index = (start_index + number_of_locations_per_process) + offset_rest_end;

            memcpy(
                &linear_vector[start_index + offset_linear_vector],
                &numbers[start_index],
                sizeof(int) * (end_index - start_index)
            );
        }
    }

    const int locations_size = number_of_locations_per_process + ((rest == 0) ? 0 : 1);
    int *my_vector = (int*)calloc(locations_size, sizeof(int));
    MPI_Scatter(
        linear_vector,
        locations_size,
        MPI_INT,
        my_vector,
        locations_size,
        MPI_INT,
        0,
        comm
    );

    free(linear_vector);

    IntVector *int_vector = (IntVector*)malloc(sizeof(IntVector));
    int_vector->data = my_vector;
    int_vector->size = this_pid < rest ? number_of_locations_per_process + 1 : number_of_locations_per_
    return int_vector;
```

```c
}

/* with MPI_Isend() and MPI_Recv() */
IntVector* uniform_distributor_numbers_v2(MPI_Comm comm, int *numbers, int total_numbers) {
    int total_processes;
    MPI_Comm_size(comm, &total_processes);

    int this_pid;
    MPI_Comm_rank(comm, &this_pid);

    const int number_of_locations_per_process = total_numbers / total_processes;
    const int rest = total_numbers % total_processes;

    static const int TAG_NUM = 80;

    if (this_pid == 0) {
        for (int pid = 0; pid < total_processes; pid++) {

            const int offset_rest_start = (rest != 0 && pid <= rest)
                ? pid
                : rest;

            const int start_index = (pid * number_of_locations_per_process) + offset_rest_start;

            MPI_Request req;
            MPI_Isend(
                &numbers[start_index],
                pid >= rest
                    ? number_of_locations_per_process
                    : number_of_locations_per_process + 1,
                MPI_INT,
                pid,
                pid + TAG_NUM,
                comm,
                &req
            );
            MPI_Request_free(&req);
        }
    }

    int *my_vector = (int*)calloc(number_of_locations_per_process + 1, sizeof(int));
    MPI_Recv(
        my_vector,
        this_pid >= rest
            ? number_of_locations_per_process
            : number_of_locations_per_process + 1,
        MPI_INT,
        0,
        this_pid + TAG_NUM,
        comm,
        MPI_STATUS_IGNORE
    );

    IntVector *int_vector = (IntVector*)malloc(sizeof(IntVector));
    int_vector->data = my_vector;
    int_vector->size = this_pid >= rest ? number_of_locations_per_process : number_of_locations_per_pro
    return int_vector;
}

/* with MPI_Scatterv() */
IntVector* uniform_distributor_numbers_v3(MPI_Comm comm, int *numbers, int total_numbers) {
```

```
    int total_processes;
    MPI_Comm_size(comm, &total_processes);

    int this_pid;
    MPI_Comm_rank(comm, &this_pid);

    const int number_of_locations_per_process = total_numbers / total_processes;
    int rest = total_numbers % total_processes;

    int *sendcounts = (int*)malloc(sizeof(int) * total_processes);
    int *displs = (int*)calloc(total_processes, sizeof(int));
    int total_data_send = 0;

    if (this_pid == 0) {
        for (int pid = 0; pid < total_processes; pid++) {
            sendcounts[pid] = number_of_locations_per_process;

            if (rest > 0) {
                sendcounts[pid]++;
                rest--;
            }

            displs[pid] = total_data_send;
            total_data_send += sendcounts[pid];
        }
    }

    int *my_vector = (int*)calloc(number_of_locations_per_process + 1, sizeof(int));
    MPI_Scatterv(
        numbers,
        sendcounts,
        displs,
        MPI_INT,
        my_vector,
        number_of_locations_per_process + 1,
        MPI_INT,
        0,
        comm
    );

    IntVector *int_vector = (IntVector*)malloc(sizeof(IntVector));
    int_vector->data = my_vector;
    int_vector->size = sendcounts[this_pid];

    free(sendcounts);
    free(displs);

    return int_vector;
}

void print_vector(IntVector *vector) {
    for (int i = 0; i < vector->size; i++)
        printf("%d ", vector->data[i]);
    printf("\n");
}

void print_vector_s(int *vector, int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", vector[i]);
    printf("\n");
}
```

```c
int* generate_random_numbers(const unsigned int total, const unsigned int max) {
    srand(time(0));
    int *random_numbers = malloc(sizeof(int) * total);
    for (int i = 0; i < total; i++) {
        random_numbers[i] = rand() % max;
        if (random_numbers[i] % 2 == 0)
            random_numbers[i] = -random_numbers[i];
    }
    return random_numbers;
}

const int get_uniform_distributor_strategy_command_line(char **argv) {
    const static int UNIFORM_DISTRIBUTOR_STRATEGY_ARGV_INDEX = 3;
    const static char *SIZE_VECTOR_ARGV_NAME = "<strategy>";
    if (!is_number(argv[UNIFORM_DISTRIBUTOR_STRATEGY_ARGV_INDEX])) {
        fprintf(stderr, "%s must be a number in {1, 2, 3}!\n", SIZE_VECTOR_ARGV_NAME);
        MPI_Finalize();
        exit(EXIT_FAILURE);
    }
    const int strategy = atoi(argv[UNIFORM_DISTRIBUTOR_STRATEGY_ARGV_INDEX]);
    if (strategy != UNIFORM_DISTRIBUTOR_NUMBERS_V1 && strategy != UNIFORM_DISTRIBUTOR_NUMBERS_V2 && str
        fprintf(stderr, "%s must be a number in {1, 2, 3}!\n", SIZE_VECTOR_ARGV_NAME);
        MPI_Finalize();
        exit(EXIT_FAILURE);
    }
    return strategy;
}

const int get_size_vector_command_line(char **argv, int total_processes) {
    const static int SIZE_VECTOR_ARGV_INDEX = 2;
    const static char *SIZE_VECTOR_ARGV_NAME = "<total_random_numbers>";
    if (!is_number(argv[SIZE_VECTOR_ARGV_INDEX])) {
        fprintf(stderr, "%s must be a positive number!\n", SIZE_VECTOR_ARGV_NAME);
        MPI_Finalize();
        exit(EXIT_FAILURE);
    }
    const int size_vector = atoi(argv[SIZE_VECTOR_ARGV_INDEX]);
    if (size_vector < total_processes) {
        fprintf(stderr, "%s must be greater than total processes %d!\n", SIZE_VECTOR_ARGV_NAME, total_p
        MPI_Finalize();
        exit(EXIT_FAILURE);
    }
    return size_vector;
}

bool is_number(const char *s) {
    if (s == NULL) return false;
    for (int i = 0; s[i] != '\0'; i++) {
        if (i == 0 && s[i] == '-' && isdigit(s[i + 1])) continue;
        if (!isdigit(s[i])) {
            return false;
        }
    }
    return true;
}

void check_arguments(int argc, char **argv) {
    if (argc != 4) {
        fprintf(stderr, "Correct usage: %s <total_random_numbers> <strategy_number>\n", argv[0]);
        MPI_Finalize();
```

```
        exit(EXIT_FAILURE);
    }
}
```

2.4 Lo Script `start_pbs.sh`

Il corretto utilizzo dello script `start_pbs.sh` è:

```
$ ./start_pbs.sh <processes> <total_random_numbers> <strategy>
```

dove `strategy` è la strategia da utilizzare (1, 2 o 3). I numeri vengono generati casualmente ma mostrati sullo stdout prima di stampare il risultato. Implementazione:

```bash
#!/bin/bash

if [ "$#" -ne 3 ]; then
  echo "Correct usage: $0 <processes> <total_random_numbers> <strategy>"
  exit 1
fi

N_CPU=$1
TOTAL_RANDOM_NUMBERS=$2
STRATEGY=$3

qsub -v N_CPU="$N_CPU",TOTAL_RANDOM_NUMBERS="$TOTAL_RANDOM_NUMBERS",STRATEGY="$STRATEGY" ./uniform_dist
```

2.5 Il File PBS

```bash
#!/bin/bash

#PBS -q studenti
#PBS -l nodes=8:ppn=8
#PBS -N uniform_distributor_numbers
#PBS -o uniform_distributor_numbers.out
#PBS -e uniform_distributor_numbers.err

sort -u $PBS_NODEFILE > hostlist
NCPU=$(wc -l hostlist)

echo ----------------------------
echo 'This job is allocated on '${NCPU}' cpu(s)'
echo 'Job is running on node(s):'
cat hostlist
PBS_O_WORKDIR=$PBS_O_HOME/mpi_cartesian_topology_exercises/exercise2
echo -------------------------------------------
echo PBS: qsub is running on $PBS_O_HOST
echo PBS: originating queue is $PBS_O_QUEUE
echo PBS: executing queue is $PBS_QUEUE
echo PBS: working directory is $PBS_O_WORKDIR
echo PBS: execution mode is $PBS_ENVIRONMENT
echo PBS: job identifier is $PBS_JOBID
echo PBS: job name is $PBS_JOBNAME
echo PBS: node file is $PBS_NODEFILE
echo PBS: current home directory is $PBS_O_HOME
echo PBS: PATH = $PBS_O_PATH
echo -----------------------------------------

/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/uniform_distributor_numbers $PBS_O_WORKDIR/unifo
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -n $N_CPU $PBS_O_WORKDIR/uniform_distribut
```

2.6 Esempi di output

2.6.1 Strategia 1 con MPI_Scatter()

Avviando lo script `start_pbs.sh` settando il numero di processi uguale a 8, il numero totale di numeri 20 (generati casualmente) e utilizzando la strategia 1 (con `MPI_Scatter()`):

```
$ ./start_pbs.sh 8 20 1
```

otteniamo il seguente output:

```
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
4006273.torque02          ...butor_numbers TRMVCN99Y              0 R studenti
[TRMVCN99Y@ui-studenti exercise2]$ qstat
[TRMVCN99Y@ui-studenti exercise2]$ ls
start_pbs.sh                     uniform_distributor_numbers.c    uniform_distributor_numbers.out
uniform_distributor_numbers  uniform_distributor_numbers.err  uniform_distributor_numbers.pbs
[TRMVCN99Y@ui-studenti exercise2]$ cat uniform_distributor_numbers.out
-----------------------------
This job is allocated on 8 hostlist cpu(s)
Job is running on node(s):
wn273.scope.unina.it
wn274.scope.unina.it
wn275.scope.unina.it
wn276.scope.unina.it
wn277.scope.unina.it
wn278.scope.unina.it
wn279.scope.unina.it
wn280.scope.unina.it
------------------------------------------
PBS: qsub is running on ui-studenti.scope.unina.it
PBS: originating queue is studenti
PBS: executing queue is studenti
PBS: working directory is /homes/DMA/PDC/2022/TRMVCN99Y/mpi_cartesian_topology_exercises/exercise2
PBS: execution mode is PBS_BATCH
PBS: job identifier is 4006273.torque02.scope.unina.it
PBS: job name is uniform_distributor_numbers
PBS: node file is /var/spool/pbs/aux//4006273.torque02.scope.unina.it
PBS: current home directory is /homes/DMA/PDC/2022/TRMVCN99Y
PBS: PATH = /usr/lib64/openmpi/1.2.7-gcc/bin:/usr/kerberos/bin:/opt/exp_soft/unina.it/intel/composer_xe
-----------------------------------------
Global Vector:
 - Size: 20
 - Numbers: 1 3 -2 -10 5 3 -8 -10 7 3 3 5 9 5 7 -6 -2 3 -4 -4

[PID 0]: 1 3 -2
[PID 1]: -10 5 3
[PID 3]: 3 3 5
[PID 2]: -8 -10 7
[PID 5]: 7 -6
[PID 7]: -4 -4
[PID 4]: 9 5
[PID 6]: -2 3

[TOTAL TIME] 3.812695e-02 seconds.
```

2.6.2 Strategia 2 con MPI_ISend() e MPI_Recv()

Avviando lo script `start_pbs.sh` settando il numero di processi uguale a 8, il numero totale di numeri 24 (generati casualmente) e utilizzando la strategia 2 (con `MPI_ISend()` e `MPI_Recv()`):

```
$ ./start_pbs.sh 8 24 2
```

otteniamo il seguente output:

```
-----------------------------
This job is allocated on 8 hostlist cpu(s)
Job is running on node(s):
wn273.scope.unina.it
wn274.scope.unina.it
wn275.scope.unina.it
wn276.scope.unina.it
wn277.scope.unina.it
wn278.scope.unina.it
wn279.scope.unina.it
wn280.scope.unina.it
------------------------------------------
PBS: qsub is running on ui-studenti.scope.unina.it
PBS: originating queue is studenti
PBS: executing queue is studenti
PBS: working directory is /homes/DMA/PDC/2022/TRMVCN99Y/mpi_cartesian_topology_exercises/exercise2
PBS: execution mode is PBS_BATCH
PBS: job identifier is 4006274.torque02.scope.unina.it
PBS: job name is uniform_distributor_numbers
PBS: node file is /var/spool/pbs/aux//4006274.torque02.scope.unina.it
PBS: current home directory is /homes/DMA/PDC/2022/TRMVCN99Y
PBS: PATH = /usr/lib64/openmpi/1.2.7-gcc/bin:/usr/kerberos/bin:/opt/exp_soft/unina.it/intel/composer_xe
-----------------------------------------
Global Vector:
 - Size: 24
 - Numbers: -6 1 -4 9 0 9 -4 -10 0 -6 -10 -8 7 -2 7 -2 -2 -4 -8 7 -2 0 -10 7

[PID 0]: -6 1 -4
[PID 1]: 9 0 9
[PID 3]: -6 -10 -8
[PID 2]: -4 -10 0
[PID 5]: -2 -2 -4
[PID 7]: 0 -10 7
[PID 4]: 7 -2 7
[PID 6]: -8 7 -2

[TOTAL TIME] 1.116204e-02 seconds.
```

2.6.3 Strategia 3 con `MPI_Scatterv()`

Avviando lo script `start_pbs.sh` settando il numero di processi uguale a 8, il numero totale di numeri 31 (generati casualmente) e utilizzando la strategia 3 (con `MPI_Scatterv()`):

```
$ ./start_pbs.sh 8 31 3
```

otteniamo il seguente output:

```
Global Vector:
 - Size: 31
 - Numbers: -10 -8 -10 -10 7 -4 -6 -10 -2 0 1 0 -10 0 -2 -4 -8 9 1 -2 -10 7 9 9 0 9 7 1 -10 0 7

[PID 1]: 7 -4 -6 -10
[PID 3]: -10 0 -2 -4
[PID 2]: -2 0 1 0
[PID 5]: -10 7 9 9
[PID 7]: -10 0 7
[PID 6]: 0 9 7 1
[PID 4]: -8 9 1 -2
[PID 0]: -10 -8 -10 -10

[TOTAL TIME] 3.859615e-02 seconds.
```

Esercizio tre

3 Esercizio tre - Matrix Partitioning Blocks

3.1 Obiettivo

Dati P processi e una matrice M di dimensione $R \times C$, crea una topologia cartesiana di processi di dimensione $p \times q$. Identifica poi $p \times q$ sottoblocchi rettangolari della matrice M e li assegna ad ogni processo che ha le corrispondenti coordinate sulla topologia cartesiana. Precondizioni:

- $R \times C \geq P$

3.2 Codice Sorgente C

```c
#include <mpi.h>
#include <malloc.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>

/*
 *  MATRIX PARTITIONING BLOCKS - EXERCISE 3
 *
 *  Given P processes and a matrix M of dimensions RxC, creates a Cartesian topology
 *  of processes of dimensions pxq. It then identifies pxq sub-blocks of the matrix M
 *  and assigns them to each process that has the corresponding coordinates.
 *
 *  Preconditions:
 *  - RxC >= P
 */

typedef struct {
    int rows;
    int columns;
    char *data;
} MatrixBlock;

MatrixBlock* partition_and_distribute_matrix_blocks(
    const MPI_Comm *comm_grid,
    int matrix_rows,
    int matrix_columns,
    char *global_matrix
);

MPI_Comm *create_bidimensional_grid(unsigned int rows, unsigned int columns);

void print_matrix(int rows, int columns, const char *matrix);

int get_matrix_columns_command_line(char **argv);
int get_matrix_rows_command_line(char **argv);
int get_number_of_comm_grid_rows_command_line(char **argv, unsigned int total_processes);
int get_number_of_rows_stdin(unsigned int number_of_processes);
bool is_number(const char *s);
void check_arguments(int argc, char **argv);

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

    check_arguments(argc, argv);
    const int COLS = get_matrix_columns_command_line(argv);
```

```c
    const int ROWS = get_matrix_rows_command_line(argv);

    int total_processes;
    MPI_Comm_size(MPI_COMM_WORLD, &total_processes);

    if (ROWS * COLS < total_processes) {
        fprintf(stderr, "The number of total processes is greater than %d x %d!\n", ROWS, COLS);
        MPI_Finalize();
        exit(EXIT_FAILURE);
    }

    int pid_comm_world;
    MPI_Comm_rank(MPI_COMM_WORLD, &pid_comm_world);

    int comm_grid_rows = (pid_comm_world == 0) ? get_number_of_comm_grid_rows_command_line(argv, total_

    MPI_Bcast(&comm_grid_rows, 1, MPI_INT, 0, MPI_COMM_WORLD);

    const int comm_grid_columns = total_processes / comm_grid_rows;

    const MPI_Comm *comm_grid = create_bidimensional_grid(comm_grid_rows, comm_grid_columns);

    int pid_comm_grid;
    MPI_Comm_rank(*comm_grid, &pid_comm_grid);

    char global_matrix[ROWS * COLS];
    if (pid_comm_grid == 0) {
        for (int i = 0; i < ROWS * COLS; i++) {
            global_matrix[i] = (char)i;
        }
    }

    MatrixBlock *local_matrix_block = partition_and_distribute_matrix_blocks(comm_grid, ROWS, COLS, glo
    const int block_rows = local_matrix_block->rows;
    const int block_columns = local_matrix_block->columns;
    const char *local_matrix = local_matrix_block->data;

    if (pid_comm_grid == 0) {
        printf("Global matrix: \n");
        print_matrix(ROWS, COLS, global_matrix);
    }

    MPI_Barrier(*comm_grid);

    printf("\n[LOCAL MATRIX] Rank = %d \n", pid_comm_grid);
    print_matrix(block_rows, block_columns, local_matrix);
}

void print_matrix(int rows, int columns, const char *matrix) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            printf("%3d ",(int)matrix[i * columns + j]);
        }
        printf("\n");
    }
    printf("\n");
}

void check_arguments(int argc, char **argv) {
    if (argc != 5) {
        fprintf(stderr, "Correct usage: %s <grid_comm_rows> <matrix_rows> <matrix_columns>\n", argv[0])
```

```c
        exit(EXIT_FAILURE);
    }
}

MPI_Comm *create_bidimensional_grid(
    const unsigned int rows,
    const unsigned int columns
) {
    static const int N_DIMS = 2;
    MPI_Comm *comm_grid = (MPI_Comm*)malloc(sizeof(MPI_Comm));

    int *dims = (int*)malloc(sizeof(int) * N_DIMS);
    dims[0] = (int)rows;
    dims[1] = (int)columns;

    const int *periods = (int*)calloc(N_DIMS, sizeof(int));
    const int reorder = 0;

    MPI_Cart_create(MPI_COMM_WORLD, N_DIMS, dims, periods, reorder, comm_grid);
    return comm_grid;
}

MatrixBlock* partition_and_distribute_matrix_blocks(
    const MPI_Comm *comm_grid,
    const int matrix_rows,
    const int matrix_columns,
    char *global_matrix
) {
    int pid_comm_grid;
    MPI_Comm_rank(*comm_grid, &pid_comm_grid);

    int dims[2], periods[2], coords[2];
    MPI_Cart_get(*comm_grid, 2, dims, periods, coords);
    const int comm_grid_rows = dims[0];
    const int comm_grid_columns = dims[1];
    const int total_processes = comm_grid_rows * comm_grid_columns;

    const int block_rows = matrix_rows / comm_grid_rows;
    const int block_columns = matrix_columns / comm_grid_columns;

    if (pid_comm_grid == 0) {
        for (int i = 0; i < matrix_rows * matrix_columns; i++) {
            global_matrix[i] = (char)i;
        }
    }

    char *local_matrix = (char*)malloc(sizeof(char) * block_rows * block_columns);

    MPI_Datatype blocktype_not_resized;
    MPI_Datatype blocktype_resized;

    MPI_Type_vector(block_rows, block_columns, matrix_columns, MPI_CHAR, &blocktype_not_resized);
    MPI_Type_create_resized( blocktype_not_resized, 0, sizeof(char), &blocktype_resized);
    MPI_Type_commit(&blocktype_resized);

    int displs[total_processes];
    int send_counts[total_processes];
    for (int i = 0; i < comm_grid_rows; i++) {
        for (int j = 0; j < comm_grid_columns; j++) {
            displs[i * comm_grid_columns + j] = i * matrix_columns * block_rows + j * block_columns;
            send_counts[i * comm_grid_columns + j] = 1;
```

```
        }
    }

    MPI_Scatterv(global_matrix, send_counts, displs, blocktype_resized, local_matrix, block_rows * bloc

    MatrixBlock *matrix_block = (MatrixBlock*)malloc(sizeof(MatrixBlock));
    matrix_block->columns = block_columns;
    matrix_block->rows = block_rows;
    matrix_block->data = local_matrix;
    return matrix_block;
}

int get_number_of_rows_stdin(const unsigned int number_of_processes) {
    int rows;

    bool is_valid_input = false;
    do {
        printf("Enter the number 'p' of rows: ");
        scanf("%d", &rows);

        if (rows > number_of_processes) {
            printf("\nThe number of rows can't be greater than the number of processes!\n\n");
        } else {
            is_valid_input = true;
        }
    } while(!is_valid_input);

    return rows;
}

int get_matrix_columns_command_line(char **argv) {
    const static int MATRIX_COLUMNS_ARGV_INDEX = 4;
    const static char *MATRIX_COLUMNS_ARGV_NAME = "<matrix_columns>";
    if (!is_number(argv[MATRIX_COLUMNS_ARGV_INDEX])) {
        fprintf(stderr, "%s must be a positive number!\n", MATRIX_COLUMNS_ARGV_NAME);
        exit(EXIT_FAILURE);
    }
    const int columns = atoi(argv[MATRIX_COLUMNS_ARGV_INDEX]);
    if (columns < 1) {
        fprintf(stderr, "%s must be greater than one!\n", MATRIX_COLUMNS_ARGV_NAME);
        exit(EXIT_FAILURE);
    }
    return columns;
}

int get_matrix_rows_command_line(char **argv) {
    const static int MATRIX_ROWS_ARGV_INDEX = 3;
    const static char *MATRIX_ROWS_ARGV_NAME = "<matrix_rows>";
    if (!is_number(argv[MATRIX_ROWS_ARGV_INDEX])) {
        fprintf(stderr, "%s must be a positive number!\n", MATRIX_ROWS_ARGV_NAME);
        MPI_Finalize();
        exit(EXIT_FAILURE);
    }
    const int rows = atoi(argv[MATRIX_ROWS_ARGV_INDEX]);
    if (rows < 1) {
        fprintf(stderr, "%s must be greater than one!\n", MATRIX_ROWS_ARGV_NAME);
        MPI_Finalize();
        exit(EXIT_FAILURE);
    }
    return rows;
}
```

```c
int get_number_of_comm_grid_rows_command_line(char **argv, const unsigned int total_processes) {
    const static int GRID_COMM_ROWS_ARGV_INDEX = 2;
    const static char *GRID_COMM_ROWS_ARGV_NAME = "<grid_comm_rows>";
    if (!is_number(argv[GRID_COMM_ROWS_ARGV_INDEX])) {
        fprintf(stderr, "%s must be a positive number!\n", GRID_COMM_ROWS_ARGV_NAME);
        exit(EXIT_FAILURE);
    }
    int grid_comm_rows = atoi(argv[GRID_COMM_ROWS_ARGV_INDEX]);
    if (grid_comm_rows < 1) {
        fprintf(stderr, "%s must be greater than one!\n", GRID_COMM_ROWS_ARGV_NAME);
        exit(EXIT_FAILURE);
    }
    if (grid_comm_rows > total_processes) {
        fprintf(
            stderr,
            "%s can't be greater than the number of processes %d!\n",
            GRID_COMM_ROWS_ARGV_NAME, total_processes
        );
        MPI_Finalize();
        exit(EXIT_FAILURE);
    }
    return grid_comm_rows;
}

bool is_number(const char *s) {
    if (s == NULL) return false;
    for (int i = 0; s[i] != '\0'; i++) {
        if (i == 0 && s[i] == '-' && isdigit(s[i + 1])) continue;
        if (!isdigit(s[i])) {
            return false;
        }
    }
    return true;
}
```

3.3 Lo Script `start_pbs.sh`

Corretto utilizzo dello script `start_pbs.sh` è il seguente:

```
$ ./start_pbs <processes> <grid_comm_rows> <matrix_rows> <matrix_columns>
```

dove `grid_comm_rows` sono le righe della matrice di processi, mentre `matrix_rows` e `matrix_columns` sono rispettivamente il numero di righe e colonne della matrice di numeri (la matrice è generata automaticamente inserendo numeri in ordine crescente). Implementazione:

```bash
#!/bin/bash

if [ "$#" -ne 4 ]; then
  echo "Correct usage: $0 <processes> <grid_comm_rows> <matrix_rows> <matrix_columns>"
  exit 1
fi

N_CPU=$1
GRID_COMM_ROWS=$2
MATRIX_ROWS=$3
MATRIX_COLUMNS=$4

qsub -v N_CPU="$N_CPU",GRID_COMM_ROWS="$GRID_COMM_ROWS",MATRIX_ROWS="$MATRIX_ROWS",MATRIX_COLUMNS="$MAT
```

3.4 Il File PBS

```
#!/bin/bash

#PBS -q studenti
#PBS -l nodes=8:ppn=8
#PBS -N matrix_partitioning_blocks
#PBS -o matrix_partitioning_blocks.out
#PBS -e matrix_partitioning_blocks.err

sort -u $PBS_NODEFILE > hostlist
NCPU=$(wc -l hostlist)

echo ----------------------------
echo 'This job is allocated on '${NCPU}' cpu(s)'
echo 'Job is running on node(s):'
cat hostlist
PBS_O_WORKDIR=$PBS_O_HOME/mpi_cartesian_topology_exercises/exercise3
echo -------------------------------------------
echo PBS: qsub is running on $PBS_O_HOST
echo PBS: originating queue is $PBS_O_QUEUE
echo PBS: executing queue is $PBS_QUEUE
echo PBS: working directory is $PBS_O_WORKDIR
echo PBS: execution mode is $PBS_ENVIRONMENT
echo PBS: job identifier is $PBS_JOBID
echo PBS: job name is $PBS_JOBNAME
echo PBS: node file is $PBS_NODEFILE
echo PBS: current home directory is $PBS_O_HOME
echo PBS: PATH = $PBS_O_PATH
echo ------------------------------------------

/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/matrix_partitioning_blocks $PBS_O_WORKDIR/matrix
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -n $N_CPU $PBS_O_WORKDIR/matrix_partitioni
```

3.5 Esempio di output

Se avviamo lo script `start_pbs.sh` con un numero totale di processi uguale a 8, righe della matrice di processi uguale a 2, e una matrice di numeri 12×8:

```
$ ./start_pbs.sh 8 2 12 8
```

otteniamo il seguente output:

```
Global matrix:
  0   1   2   3   4   5   6   7
  8   9  10  11  12  13  14  15
 16  17  18  19  20  21  22  23
 24  25  26  27  28  29  30  31
 32  33  34  35  36  37  38  39
 40  41  42  43  44  45  46  47
 48  49  50  51  52  53  54  55
 56  57  58  59  60  61  62  63
 64  65  66  67  68  69  70  71
 72  73  74  75  76  77  78  79
 80  81  82  83  84  85  86  87
 88  89  90  91  92  93  94  95

[LOCAL MATRIX] Rank = 0
 0   1
 8   9
 16  17
 24  25
 32  33
```

```
 40  41

[LOCAL MATRIX] Rank = 6
 52  53
 60  61
 68  69
 76  77
 84  85
 92  93

[LOCAL MATRIX] Rank = 1
  2   3
 10  11
 18  19
 26  27
 34  35
 42  43

[LOCAL MATRIX] Rank = 2
  4   5
 12  13
 20  21
 28  29
 36  37
 44  45

[LOCAL MATRIX] Rank = 4
 48  49
 56  57
 64  65
 72  73
 80  81
 88  89

[LOCAL MATRIX] Rank = 5
 50  51
 58  59
 66  67
 74  75
 82  83
 90  91

[LOCAL MATRIX] Rank = 3
  6   7
 14  15
 22  23
 30  31
 38  39
 46  47

[LOCAL MATRIX] Rank = 7
 54  55
 62  63
 70  71
 78  79
 86  87
 94  95
```