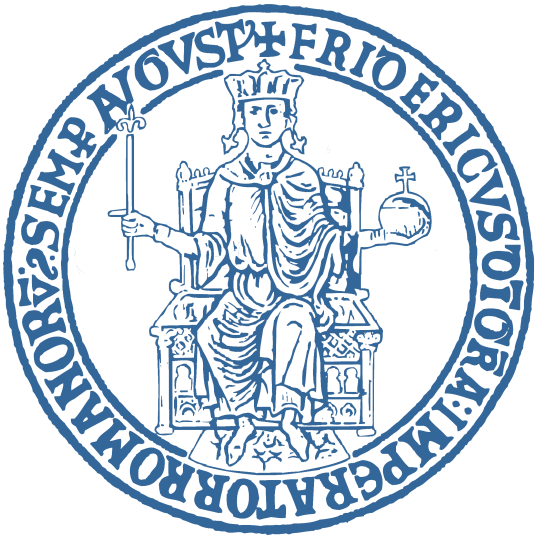

Università degli Studi di Napoli Federico II

Parallel and Distributed Computing

2022/2023

Professori	*Studente*
Giuliano Laccetti	Vincenzo Tramo
Anna Mele	

Indice

1 Prodotto matrice vettore

1.1 Obiettivo

- Dati P processi, una matrice $A \in \mathbb{R}^{M \times N}$, un vettore $x \in \mathbb{R}^N$ e il numero totale di righe p della topologia cartesiana di processi, l'algoritmo crea una griglia bidimensionale di processi di dimensione $p \times (\frac{P}{p})$ quindi partiziona e distribuisce la matrice A a questi processi e calcola il prodotto $A * x = b$ in parallelo. **Precondizioni**:
  - Il numero totale di elementi della matrice *deve essere maggiore o uguale* del numero totale di processi, sarebbe a dire $M * N \geq P$
  - Il numero totale di processi *deve essere divisibile* per il numero totale di righe della griglia di processi, sarebbe a dire $P\%p == 0$
  - Il numero totale di righe della matrice *deve essere divisibile* per il numero totale di righe della griglia di processi, sarebbe a dire $M\%p == 0$
  - Il numero totale di colonne *deve essere divisibile* per il numero totale di colonne della griglia di processi, sarebbe a dire $N\%(P/p) == 0$

In input al programma va fornito:

- Il numero totale di processi P
- Il numero totale di righe della griglia di processi p
- Il numero di righe della matrice M
- Il numero di colonne della matrice N

Nota bene: la matrice A (così come il vettore x) per semplicità vengono generati durante l'esecuzione del programma.

1.2 La funzione `create_bidimensional_grid`

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

    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);
}
```

Tale funzione crea una griglia bidimensionale di processi di dimensione `rows` $\times$ `columns`.

1.3 La funzione `partition_and_distribute_matrix_blocks`

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

MatrixBlock* partition_and_distribute_matrix_blocks(
```

```
    const MPI_Comm *comm_grid,
    const int matrix_rows,
    const int matrix_columns,
    int *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_total_rows = dims[0];
    const int comm_grid_total_columns = dims[1];
    const int total_processes = comm_grid_total_rows * comm_grid_total_columns;

    const int block_rows = matrix_rows / comm_grid_total_rows;
    const int block_columns = matrix_columns / comm_grid_total_columns;

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

    MPI_Datatype blocktype_not_resized;
    MPI_Datatype blocktype_resized;

    MPI_Type_vector(
        block_rows,
        block_columns,
        matrix_columns,
        MPI_INT,
        &blocktype_not_resized
    );
    MPI_Type_create_resized( blocktype_not_resized, 0, sizeof(int), &blocktype_resized);
    MPI_Type_commit(&blocktype_resized);

    int displs[total_processes];
    int send_counts[total_processes];
    for (int i = 0; i < comm_grid_total_rows; i++) {
        for (int j = 0; j < comm_grid_total_columns; j++) {
            displs[i * comm_grid_total_columns + j] =
                i * matrix_columns * block_rows + j * block_columns;
            send_counts[i * comm_grid_total_columns + j] = 1;
        }
    }

    MPI_Scatterv(
        global_matrix,
        send_counts,
        displs,
        blocktype_resized,
        local_matrix,
        block_rows * block_columns,
        MPI_INT,
        0,
        MPI_COMM_WORLD
    );

    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;
}
```

La seguente funzione restituisce la giusta sottomatrice partizionata di A per il processo chiamante appartenente alla griglia bidimensionale di processi. Per semplificare l'implementazione e permettere l'utilizzo di funzioni di MPI come `MPI_Scatterv` è stato scelto di rappresentare la matrice e le sottomatrici come un vettore unidimensionale. Il tipo strutturato `MatrixBlock` contiene la sottomatrice di un processo (nel campo `data`) e il numero totale di righe (`rows`) e colonne (`columns`).

1.4 La funzione `calculate_partial_solution`

```
int *calculate_partial_solution(
    MatrixBlock *matrix_block,
    const int *x,
    const int x_size,
    const int *comm_grid_coords
) {
    int *partial_solution = (int*)calloc(x_size, sizeof(int));
    for (int i = 0; i < matrix_block->rows; i++) {
        for (int j = 0; j < matrix_block->columns; j++) {
            partial_solution[i + (matrix_block->rows * comm_grid_coords[0])] +=
                matrix_block->data[i * matrix_block->columns + j]  *
                x[j + (matrix_block->columns * comm_grid_coords[1])];
        }
    }
    return partial_solution;
}
```

La seguente funzione restituisce per il processo chiamante con coordinate `comm_grid_coords` all'interno della griglia `comm_grid` (griglia bidimensionale di processi creata precedentemente) il vettore risultato parziale che dovrà poi essere sommato a tutti gli altri vettori soluzione parziali (notare che un vettore soluzione parziale ha la stessa dimensione del vettore soluzione intero b). Tutti i processi hanno il vettore `x` per intero (`x_size` è la dimensione del vettore `x`.

1.4.1 Perché si è scelto di non utilizzare la funzione `MPI_Cart_sub` per partizionare il communicator in sottogruppi

La funzione `calculate_partial_solution` restituisce il vettore soluzione parziale che ha esattamente la stessa dimensione del vettore soluzione reale b. Se la sottomatrice fornita a tale funzione non ha abbastanza colonne per calcolare il resto delle componenti parziali del vettore soluzione parziale `partial_solution`, le componenti non calcolate avranno valore zero (questo è fatto dalla funzione `calloc`). Quindi tutti i vettori soluzione parziali restituiti da `calculate_partial_solution` avranno tutti la stessa dimensione e in particolare avranno tutti la stessa dimensione del vettore soluzione reale b. Non ci resta allora che sommare tutti i vettori parziali utilizzando una `MPI_Allreduce`:

```
    /* Calculate partial solution */
    int *coords = (int*)malloc(sizeof(int) * 2);
    MPI_Cart_coords(*comm_grid, pid_comm_grid, 2, coords);
    int *partial_solution = calculate_partial_solution(local_matrix_block, x, ROWS, coords);

    /* Calculate total solution */
    int *total_solution = (int*)calloc(ROWS, sizeof(int));
    MPI_Allreduce(
        partial_solution,
        total_solution,
        ROWS,
        MPI_INT,
        MPI_SUM,
        *comm_grid
    );
```

così facendo tutti i processi avranno la soluzione intera. Per il corretto funzionamento dell'algoritmo è fondamentale fornire le coordinate del processo chiamante alla funzione `calculate_partial_solution` in maniera tale da essere identificato all'interno della griglia bidimensionale e capire di conseguenza in quale posizione calcolare le coordinate parziali all'interno del vettore soluzione parziale per ogni passo (i, j).

Facciamo un esempio per capire il funzionamento. Poniamo M il numero di righe della matrice uguale a 6 e il numero di colonne N della matrice uguale a 4 creando così una matrice di dimensione 6×4. Poniamo il numero di processi P uguale a 6 e il numero di righe della griglia bidimensionale uguale a 3 (e di conseguenza il numero di colonne uguale a $\frac{6}{3} = 2$) creando così una griglia bidimensione di 3×2 processi. Dovranno quindi essere create 6 sottomatrici e 6 invocazioni alla funzione `calculate_partial_solution` dovranno essere fatte. Se eseguiamo il programma con questi parametri e inseriamo la seguente `printf` all'interno della funzione `calculate_partial_solution`:

```
int *calculate_partial_solution(
    MatrixBlock *matrix_block,
    const int *x,
    const int x_size,
    const int *comm_grid_coords
) {
    int *partial_solution = (int*)calloc(x_size, sizeof(int));
    for (int i = 0; i < matrix_block->rows; i++) {
        for (int j = 0; j < matrix_block->columns; j++) {
            partial_solution[i + (matrix_block->rows * comm_grid_coords[0])] +=
                matrix_block->data[i * matrix_block->columns + j]  *
                x[j + (matrix_block->columns * comm_grid_coords[1])];
        }
    }

    printf(
        "Coordinate (%d, %d)\n"
        "\tRighe sottomatrice %d\n"
        "\tColonne sottomatrice %d\n"
        "\tSoluzione parziale calcolata: ",
        comm_grid_coords[0], comm_grid_coords[1], matrix_block->rows, matrix_block->columns
    );
    print_vector(partial_solution, x_size);

    return partial_solution;
}
```

otteniamo il seguente output:

```
Coordinate (0, 0)
    Righe sottomatrice 2
    Colonne sottomatrice 2
    Soluzione parziale calcolata: [1 5 0 0 0 0]
Coordinate (0, 1)
    Righe sottomatrice 2
    Colonne sottomatrice 2
    Soluzione parziale calcolata: [13 33 0 0 0 0]
Coordinate (1, 0)
    Righe sottomatrice 2
    Colonne sottomatrice 2
    Soluzione parziale calcolata: [0 0 9 13 0 0]
Coordinate (1, 1)
    Righe sottomatrice 2
    Colonne sottomatrice 2
    Soluzione parziale calcolata: [0 0 53 73 0 0]
Coordinate (2, 0)
    Righe sottomatrice 2
    Colonne sottomatrice 2
    Soluzione parziale calcolata: [0 0 0 0 17 21]
Coordinate (2, 1)
    Righe sottomatrice 2
    Colonne sottomatrice 2
    Soluzione parziale calcolata: [0 0 0 0 93 113]
```

```
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

Vector x: [0 1 2 3]

Result: [14 38 62 86 110 134]

[LOCAL MATRIX] Rank = 0
  0   1
  4   5

[LOCAL MATRIX] Rank = 1
  2   3
  6   7

[LOCAL MATRIX] Rank = 2
  8   9
 12  13

[LOCAL MATRIX] Rank = 3
 10  11
 14  15

[LOCAL MATRIX] Rank = 4
 16  17
 20  21

[LOCAL MATRIX] Rank = 5
 18  19
 22  23
```

1.5 Intero codice sorgente C

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

/*
 *  Given P processes, a matrix A of dimensions M x N, a vector x of dimension N and
 *  the total number of rows p of the cartesian topology of processes, the algorithm
 *  creates a bidimensional grid of processes of size p x (P/p), partitions, distributes the matrix
 *  to these processes and computes the product A * x = b in parallel.
 *
 *  Preconditions:
 *  - M * N >= P
 *  - P % p == 0
 *  - M % p == 0
 *  - N % (P/p) == 0
 */

typedef struct {
    int rows;
    int columns;
```

```
    int *data;
} MatrixBlock;

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

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

int *calculate_partial_solution(
    MatrixBlock *matrix_block,
    const int *x,
    int x_size,
    const int* comm_grid_coords
);

void print_matrix(int rows, int columns, const int *matrix);
void print_vector(int *v, int size);

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);

bool is_number(const char *s);

void check_arguments(int argc, char **argv);
void check_validity_cartesian_topology_processes(
    int total_rows_matrix,
    int total_columns_matrix,
    int total_processes,
    int comm_grid_total_rows,
    int comm_grid_total_columns,
    int pid_comm_world
);

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

    check_arguments(argc, argv);
    const int COLS = get_matrix_columns_command_line(argv);
    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_total_rows = (pid_comm_world == 0)
   ? get_number_of_comm_grid_rows_command_line(argv, total_processes)
   : 0;

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

const int comm_grid_total_columns = total_processes / comm_grid_total_rows;

check_validity_cartesian_topology_processes(
    ROWS,
    COLS,
    total_processes,
    comm_grid_total_rows,
    comm_grid_total_columns,
    pid_comm_world
);

MPI_Comm *comm_grid = (MPI_Comm*)malloc(sizeof(MPI_Comm));
create_bidimensional_grid(
    comm_grid,
    comm_grid_total_rows,
    comm_grid_total_columns
);

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

/* Create and fill the global matrix */
int global_matrix[ROWS * COLS];
if (pid_comm_grid == 0) {
    for (int i = 0; i < ROWS * COLS; i++) {
        global_matrix[i] = i;
    }
}

/* Get matrix block */
MatrixBlock *local_matrix_block =
    partition_and_distribute_matrix_blocks(
        comm_grid,
        ROWS,
        COLS,
        global_matrix
    );

/* Create and fill vector x (A * x = b) */
int *x = (int*)malloc(sizeof(int) * COLS);
for (int i = 0; i < COLS; i++) x[i] = i;

MPI_Barrier(*comm_grid);
const double t0 = MPI_Wtime();

/* Calculate partial solution */
int *coords = (int*)malloc(sizeof(int) * 2);
MPI_Cart_coords(*comm_grid, pid_comm_grid, 2, coords);
int *partial_solution = calculate_partial_solution(local_matrix_block, x, ROWS, coords);

/* Calculate total solution */
int *total_solution = (int*)calloc(ROWS, sizeof(int));
MPI_Allreduce(
    partial_solution,
    total_solution,
```

```
        ROWS,
        MPI_INT,
        MPI_SUM,
        *comm_grid
    );

    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, *comm_grid);

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

        printf("Vector x: ");
        print_vector(x, COLS);

        printf("Result: ");
        print_vector(total_solution, ROWS);

        printf("[TOTAL TIME] %e seconds.\n", max_total_time);
    }

    MPI_Barrier(*comm_grid);

    printf("\n[LOCAL MATRIX] Rank = %d \n", pid_comm_grid);
    print_matrix(local_matrix_block->rows, local_matrix_block->columns, local_matrix_block->data);

    free(total_solution);
    free(partial_solution);
    free(coords);
    free(local_matrix_block);
    free(x);
    free(comm_grid);

    return EXIT_SUCCESS;
}

void check_validity_cartesian_topology_processes(
    const int total_rows_matrix,
    const int total_columns_matrix,
    int total_processes,
    int comm_grid_total_rows,
    const int comm_grid_total_columns,
    int pid_comm_world
) {
    if (total_processes % comm_grid_total_rows != 0) {
        if (pid_comm_world == 0) {
            fprintf(stderr,
                "The total number of processes must be divisible by the number "
                "of rows of the cartesian topology of processes (argument <grid_comm_rows>)!\n"
            );
        }
        MPI_Finalize();
        exit(EXIT_FAILURE);
    }

    if (total_rows_matrix % comm_grid_total_rows != 0) {
```

```
        if (pid_comm_world == 0) {
            fprintf(stderr,
                "The total number of rows of the matrix must be divisible by the total number "
                "of rows of the cartesian topology of processes (argument <grid_comm_rows>)!\n"
            );
        }
        MPI_Finalize();
        exit(EXIT_FAILURE);
    }

    if (total_columns_matrix % comm_grid_total_columns != 0) {
        if (pid_comm_world == 0) {
            fprintf(stderr,
                "The total number of columns of the matrix must be divisible by the total number "
                "of columns of the cartesian topology of processes!\n"
            );
        }
        MPI_Finalize();
        exit(EXIT_FAILURE);
    }
}

void print_vector(int *v, int size) {
    for (int i = 0; i < size; i++)
        printf("%s%d%s", i == 0 ? "[" : "", v[i], i == size - 1 ? "]\n" : " ");
}

void print_matrix(int rows, int columns, const int *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 != 4) {
        fprintf(stderr, "Correct usage: %s <grid_comm_rows> <matrix_rows> <matrix_columns>\n", argv[0])
        exit(EXIT_FAILURE);
    }
}

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

    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);
}
```

```
MatrixBlock* partition_and_distribute_matrix_blocks(
    const MPI_Comm *comm_grid,
    const int matrix_rows,
    const int matrix_columns,
    int *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_total_rows = dims[0];
    const int comm_grid_total_columns = dims[1];
    const int total_processes = comm_grid_total_rows * comm_grid_total_columns;

    const int block_rows = matrix_rows / comm_grid_total_rows;
    const int block_columns = matrix_columns / comm_grid_total_columns;

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

    MPI_Datatype blocktype_not_resized;
    MPI_Datatype blocktype_resized;

    MPI_Type_vector(
        block_rows,
        block_columns,
        matrix_columns,
        MPI_INT,
        &blocktype_not_resized
    );
    MPI_Type_create_resized( blocktype_not_resized, 0, sizeof(int), &blocktype_resized);
    MPI_Type_commit(&blocktype_resized);

    int displs[total_processes];
    int send_counts[total_processes];
    for (int i = 0; i < comm_grid_total_rows; i++) {
        for (int j = 0; j < comm_grid_total_columns; j++) {
            displs[i * comm_grid_total_columns + j] =
                i * matrix_columns * block_rows + j * block_columns;
            send_counts[i * comm_grid_total_columns + j] = 1;
        }
    }

    MPI_Scatterv(
        global_matrix,
        send_counts,
        displs,
        blocktype_resized,
        local_matrix,
        block_rows * block_columns,
        MPI_INT,
        0,
        MPI_COMM_WORLD
    );

    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 *calculate_partial_solution(
    MatrixBlock *matrix_block,
    const int *x,
    const int x_size,
    const int *comm_grid_coords
) {
    int *partial_solution = (int*)calloc(x_size, sizeof(int));
    for (int i = 0; i < matrix_block->rows; i++) {
        for (int j = 0; j < matrix_block->columns; j++) {
            partial_solution[i + (matrix_block->rows * comm_grid_coords[0])] +=
                matrix_block->data[i * matrix_block->columns + j]  *
                x[j + (matrix_block->columns * comm_grid_coords[1])];
        }
    }
    return partial_solution;
}

int get_matrix_columns_command_line(char **argv) {
    const static int MATRIX_COLUMNS_ARGV_INDEX = 3;
    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 = 2;
    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;
}

int get_number_of_comm_grid_rows_command_line(char **argv, const unsigned int total_processes) {
    const static int GRID_COMM_ROWS_ARGV_INDEX = 1;
    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;
}
```

1.6 Lo script `start_pbs.sh`

Lo script `start_pbs.sh` avvia in maniera opportuna il file pbs. Per eseguire correttamente lo script vanno forniti i seguenti parametri di input:

```
./start_pbs <total_processes> <comm_grid_rows> <matrix_rows> <matrix_columns>
```

ricordando di rispettare le precondizioni (opportunamente controllate durante l'esecuzione del programma) elencate in Sezione 1.1.

```
#!/bin/bash

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

N_CPU=$1
COMM_GRID_ROWS=$2
ROWS=$3
COLUMNS=$4

qsub -v N_CPU="$N_CPU",COMM_GRID_ROWS="$COMM_GRID_ROWS",ROWS="$ROWS",COLUMNS="$COLUMNS" \
        ./parallel_vector_matrix_product.pbs
```

1.7 Il file PBS

```
#!/bin/bash

#PBS -q studenti
#PBS -l nodes=8:ppn=8
#PBS -N parallel_vector_matrix_product
#PBS -o parallel_vector_matrix_product.out
#PBS -e parallel_vector_matrix_product.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/parallel-vector-matrix-product
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/parallel_vector_matrix_product \
    $PBS_O_WORKDIR/parallel_vector_matrix_product.c -std=c99
/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -n $N_CPU \
    $PBS_O_WORKDIR/parallel_vector_matrix_product $COMM_GRID_ROWS $ROWS $COLUMNS
```

1.8 Esempio di output

Avviando lo script `start_pbs.sh` con i seguenti parametri:

```
./start_pbs.sh 8 4 8 8
```

sarebbe a dire 8 processi, una griglia bidimensionale di processi di dimensione 4×2 e una matrice di dimensione 8×8, 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

Vector x: [0 1 2 3 4 5 6 7]

Result: [140 364 588 812 1036 1260 1484 1708]

[TOTAL TIME] 8.296967e-05 seconds.

[LOCAL MATRIX] Rank = 0
 0   1   2   3
 8   9  10  11

[LOCAL MATRIX] Rank = 3
 20  21  22  23
 28  29  30  31

[LOCAL MATRIX] Rank = 4
 32  33  34  35
 40  41  42  43

[LOCAL MATRIX] Rank = 5
 36  37  38  39
```

```
 44  45  46  47

[LOCAL MATRIX] Rank = 6
 48  49  50  51
 56  57  58  59

[LOCAL MATRIX] Rank = 2
 16  17  18  19
 24  25  26  27

[LOCAL MATRIX] Rank = 1
  4   5   6   7
 12  13  14  15

[LOCAL MATRIX] Rank = 7
 52  53  54  55
 60  61  62  63
```