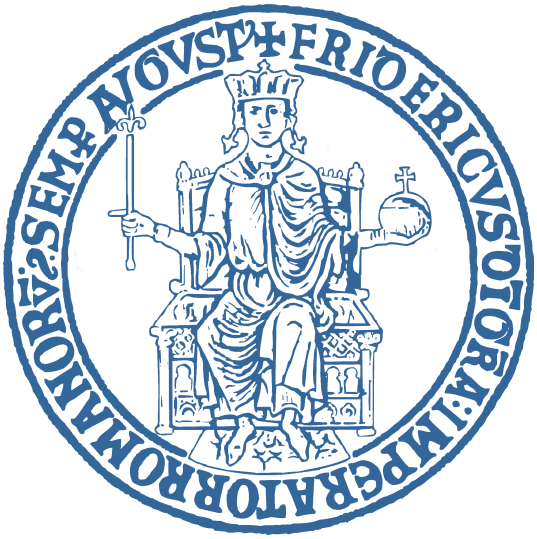


Università degli Studi di Napoli Federico II

Parallel and Distributed Computing

Broadcast Multiply Rolling con OpenMPI

2022/2023

Professori	*Studente*
Anna Mele	N86002592
Giuliano Laccetti	Vincenzo Tramo

Indice

Descrizione del problema

1 Descrizione del problema

1.1 Obiettivo

- Calcolo del prodotto matrice-matrice, in ambiente di calcolo parallelo su architettura MIMD a memoria distribuita utilizzando la libreria MPI.

  Date due matrici $\mathbf{A} \in \mathbb{R}^{m \times m}$ e $\mathbf{B} \in \mathbb{R}^{m \times m}$, e dati $p \times p$ processi, l'algoritmo parallelo costruisce una griglia quadrata di processi di dimensione $p \times p$ e, successivamente, partiziona e distribuisce equamente le matrici $\mathbf{A}$ e $\mathbf{B}$ a tutti i $p \times p$ processi. Dopodichè esegue il calcolo del prodotto matrice-matrice:

  $$\mathbf{A} \cdot \mathbf{B} = \mathbf{C} \in \mathbb{R}^{m \times m}$$

  in maniera parallela (algoritmo **Broadcast Multiply Rolling**).

  Precondizioni:

  – Il valore m deve essere un multiplo di p

1.1.1 Confronto con l'algoritmo sequenziale

L'obiettivo è anche quello di confrontare l'algoritmo parallelo con quello sequenziale dal punto di vista del tempo di esecuzione totale necessario per risolvere il problema.

L'algoritmo sequenziale del prodotto tra due matrici $\mathbf{A} \in \mathbb{R}^{m \times m}$ e $\mathbf{B} \in \mathbb{R}^{m \times m}$ che esegue il prodotto $\mathbf{A} \cdot \mathbf{B} = \mathbf{C} \in \mathbb{R}^{m \times m}$ è il seguente:

```
MatrixBlock *multiply_matrices(MatrixBlock *A, MatrixBlock *B) {
    if (A->columns != B->rows) return NULL;
    int *result = (int*) malloc(sizeof(int) * A->rows * B->columns);
    for (int i = 0; i < A->rows; i++) {
        for (int j = 0; j < B->columns; j++) {
            result[i * B->columns + j] = 0;
            for (int k = 0; k < B->rows; k++) {
                result[i * B->columns + j] +=
                    A->data[i * A->columns + k] * B->data[k * B->columns + j];
            }
        }
    }
    MatrixBlock *C = (MatrixBlock*) malloc(sizeof(MatrixBlock));
    C->data = result;
    C->rows = A->rows;
    C->columns = B->columns;
    return C;
}
```

Il prodotto tra due matrici $\mathbf{A}$ e $\mathbf{B}$ può essere eseguito soltanto se il numero di colonne della matrice $\mathbf{A}$ è uguale al numero di righe della matrice $\mathbf{B}$. La funzione `multiply_matrices` quindi controlla questa precondizione, esegue il prodotto matrice-matrice e ritorna la matrice $\mathbf{C}$ risultato.

1.1.2 Il tipo di dato `MatrixBlock`

Il tipo di dato `MatrixBlock` rappresenta fondamentalmente una matrice. I suoi campi sono:

- `int rows`: il numero di righe della matrice
- `int columns`: il numero di colonne della matrice
- `int *data`: i valori contenuti nella matrice

E' possibile passare dal tipo di dato `int` a quello `float` o `double` in qualunque momento.

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

Il tipo di dato `MatrixBlock` viene utilizzato da molte funzioni del programma che implementano l'algoritmo Broadcast Multiply Rolling come ad esempio la funzione `multiply_matricies` appena descritta che esegue il prodotto matrice-matrice in maniera sequenziale.

Tuttavia si è scelto di realizzare un file sorgente C indipendente per eseguire il calcolo del prodotto matrice per matrice su un singolo processore. Tale file sorgente si chiama `single_process_matrix_per_matrix_product.c` e viene utilizzato per misurare precisamente il tempo di esecuzione dell'algoritmo che risolve il problema del prodotto matrice per matrice su un singolo processore. Il file sorgente C che implementa l'algoritmo parallelo Broadcast Multiply Rolling è chiamato invece `broadcast_multiply_rolling.c`. Entrambi gli algoritmi (quello parallelo e sequenziale) utilizzano la funzione `multiply_matrices`. In particolare l'algoritmo parallelo utilizza tale funzione per eseguire i prodotti parziali di matrici parziali.

Un'altra funzione che utilizza il tipo di dato `MatrixBlock` è la funzione `sum_matrices` utilizzata in particolare dall'algorlitmo parallelo BMR per eseguire la somma tra due matrici. La funzione `sum_matrices` è la seguente:

```
MatrixBlock *sum_matrices(MatrixBlock *A, MatrixBlock *B) {
    if (A->rows != B->rows || A->columns != B->columns) return NULL;
    const int rows = A->rows; const int columns = A->columns;
    int *result = (int*) malloc(sizeof(int) * A->rows * A->columns);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            result[i * columns + j] = A->data[i * columns + j] + B->data[i * columns + j];
        }
    }
    MatrixBlock *C = (MatrixBlock*) malloc(sizeof(MatrixBlock));
    C->data = result;
    C->rows = A->rows;
    C->columns = A->columns;
    return C;
}
```

La somma tra due matrici può essere fatta solo se le due matrici **A** e **B** hanno lo stesso numero di righe e lo stesso numero di colonne. La funzione `sum_matrices` prima controlla se tale precondizione è soddisfatta, poi esegue la somma tra le due matrici e restituisce la matrice **C** come risultato.

Descrizione Algoritmo

2 Descrizione Algoritmo

2.1 Broadcast Multiply Rolling

La strategia utilizzata per eseguire il prodotto matrice-matrice è quella dell'algoritmo **Broadcast Multiply Rolling**. Il Broadcast multiply Rolling, o BMR, è un algoritmo utilizzato per effettuare il prodotto di matrici in parallelo su una architettura di calcolo distribuita. Tale sezione spiega come funziona in dettaglio l'algoritmo BMR e come è stato implementato con l'utilizzo della libreria MPI. Dopodichè verrà spiegato come eseguire correttamente il programma sul cluster SCoPE, quali sono gli input e verranno presentati degli esempi di output. L'implementazione di tale algoritmo è il file sorgente C `broadcast_multiply_rolling.c`.

2.1.1 Distribuzione delle matrici A e B

Il passo preliminare è quello di partizionare e suddividere le matrici quadrate **A** e **B**. Le matrici **A** e **B** vengono suddivise in $p \times p$ blocchi e le sottomatrici così ottenute vengono distribuite tra i $p \times p$ processi. In particolare, le sottomatrici $\mathbf{A}_{ij}$ e $\mathbf{B}_{ij}$ sono assegnate al corrispondente processo p_{ij} nella griglia quadrata di processi dove $i = 0, \ldots, p-1$ e $j = 0, \ldots, p-1$.

Prima di tutto è necessario acquisire in input le matrici **A** e **B**. Tali matrici, per semplicità, vengono generate automaticamente dal programma. Prima di procedere con l'algoritmo, vengono create e riempiete le matrici **A** e **B** in maniera deterministica. Questo è il pezzo di codice situato nel `main()` che svolge questo compito:

```
/* Create and fill the global matrix A and B */
int global_matrix_A[square_matrix_total_elements];
int global_matrix_B[square_matrix_total_elements];
if (pid_comm_grid == 0) {
    for (int i = 0; i < square_matrix_total_elements; i++) {
        global_matrix_A[i] = i;
        global_matrix_B[i] = i;
    }
}
```

La variabile `square_matrix_total_elements` può essere data in input al programma prima dell'esecuzione. In particolare, il parametro che può essere dato in input al programma è la dimensione della matrice quadrata. Questa variabile deve sottostare alla precondizione definita in Sezione 1.

Se ad esempio `square_matrix_total_elements` $= 16$, allora la dimensione delle matrici quadrate sarà 4×4 e quindi verranno generate due matrici **A** e **B** di dimensione 4×4 con i seguenti valori:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

Successivamente, prima di partizionare e distribuire le matrici **A** e **B** è necessario creare la griglia quadrata di processi di dimensione $p \times p$ (dove p è un parametro di input e rappresenta il numero di righe e colonne della griglia quadrata di processi). La creazione della griglia quadrata di processi di dimensione $p \times p$ viene fatta dalla seguente funzione:

```
void create_square_grid_processes(
    MPI_Comm *comm_grid,
    const unsigned int dim
) {
    static const int N_DIMS = 2;

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

    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);
}
```

La funzione riceve in input la dimensione `dim` della griglia quadrata di processi da creare (se ad esempio `dim` $= 2$ verrà creata una griglia di processi di dimensione 2×2). La precondizione di tale funzione è che `dim` deve essere uguale al numero di processi allocati $p \times p$, altrimenti la funzione `MPI_Cart_create` genererà un errore. L'output della funzione sarà `comm_grid` di tipo `MPI_Comm` opportunamente istanziato prima dell'invocazione di tale funzione. Nel `main()` la sua invocazione avviene in questo modo:

```
// nel main()
MPI_Comm *comm_grid = (MPI_Comm *) malloc(sizeof(MPI_Comm));
create_square_grid_processes(comm_grid, comm_grid_dim);
```

dove il parametro attuale `comm_grid_dim` è un parametro da dare in input al programma prima della sua esecuzione.

Una volta ottenute le due matrici quadrate $\mathbf{A}$ e $\mathbf{B}$ e la corrispondente griglia quadrata di processi $p \times p$, si procede con il partizionamento e la distribuzione di tali matrici. La responsabilità di tale compito è stata incapsulata nella funzione `partition_and_distribute_matrix_blocks`:

```
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;
}
```

Tale funzione riceve in input:

- `const MPI_Comm *comm_grid`: una griglia di processi di qualsiasi dimensione
- `const int matrix_rows`: il numero di righe della matrice da partizionare e da distribuire tra i processi
- `const int matrix_columns`: il numero di colonne della matrice da partizionare e da distribuire tra i processi
- `int *global_matrix`: la matrice da partizionare e da distribuire tra i processi nella griglia di processi `comm_grid`

Le precondizioni da rispettare per il corretto funzionamento di tale funzione sono:

- Il numero di righe della griglia di processi deve essere un multiplo del numero di righe della matrice da partizionare e distribuire
- Il numero di colonne della griglia di processi deve essere un multiplo del numero di colonne della matrice da partizionare e distribuire

La violazione di tali precondizioni genererà un errore.

La funzione restituirà una determinata sottomatrice di `global_matrix` in base a quale processo ha invocato tale funzione. La funzione restituisce la giusta sottomatrice per il processo chiamante che ha determinate coordinate all'interno della griglia di processi `comm_grid` passata in input alla funzione.

La funzione `partition_and_distribute_matrix_blocks` prima ottiene varie informazioni di servizio, come l'identificativo del processo chiamante all'interno della griglia di processi, le sue coordinate all'interno di essa, il numero totale di righe e colonne della griglia di processi, il numero totale dei processi e determina esattamente il numero di righe e colonne che tutte le sottomatrici della matrice principale devono avere. Dopodichè, con l'uso di tali informazioni, realizza un tipo di dato MPI personalizzato utilizzando le funzione `MPI_Type_vector`, `MPI_Type_create_resized` e `MPI_Type_commit` da dare in input successivamente poi alla funzione responsabile del partizionamento e della distribuzione della matrice principale, cioè `MPI_Scatterv`. Tali funzioni verranno discusse meglio successivamente ma si consiglia tuttavia di consultare la documentazione ufficiale di OpenMPI per maggiori dettagli. Una volta eseguita la funzione `MPI_Scatterv`, ogni processo avrà la propria sottomatrice che, all'interno della funzione, è chiamata `local_matrix`. A questo punto viene istanziato un oggetto di tipo `MatrixBlock` per incapsulare tale matrice e restituirla al processo chiamante.

La funzione `partition_and_distribute_matrix_blocks` all'interno del `main()` in questo modo:

```
    /* Get local matrix block A for this process */
    MatrixBlock *local_matrix_block_A =
        partition_and_distribute_matrix_blocks(
            comm_grid,
            square_matrix_dim,
            square_matrix_dim,
            global_matrix_A
        );

    /* Get local matrix block B for this process */
    MatrixBlock *local_matrix_block_B =
```

```
partition_and_distribute_matrix_blocks(
    comm_grid,
    square_matrix_dim,
    square_matrix_dim,
    global_matrix_B
);
```

Le matrici $\mathbf{A}$ e $\mathbf{B}$ verranno quindi partizionate e distribuite a tutti i processi nella griglia quadrata di processi `comm_grid`. Alla fine, quindi, tutti i processi avranno le proprie sottomatrici $\mathbf{A}_{ij}$ e $\mathbf{B}_{ij}$ dove la coppia (i, j) sono le coordinate del processo p_{ij} all'interno della griglia di processi.

Supponiamo ad esempio che $m = 4$. Dunque abbiamo due matrici quadrate $\mathbf{A} \in \mathbb{R}^{4\times4}$ e $\mathbf{B} \in \mathbb{R}^{4\times4}$. Supponiamo inoltre di avere una griglia bidimensionale di processi di dimensione 2×2 (ponendo $p = 2$). La Figura 1 mostra quali sottomatrici vengono assegnate a quali processi.

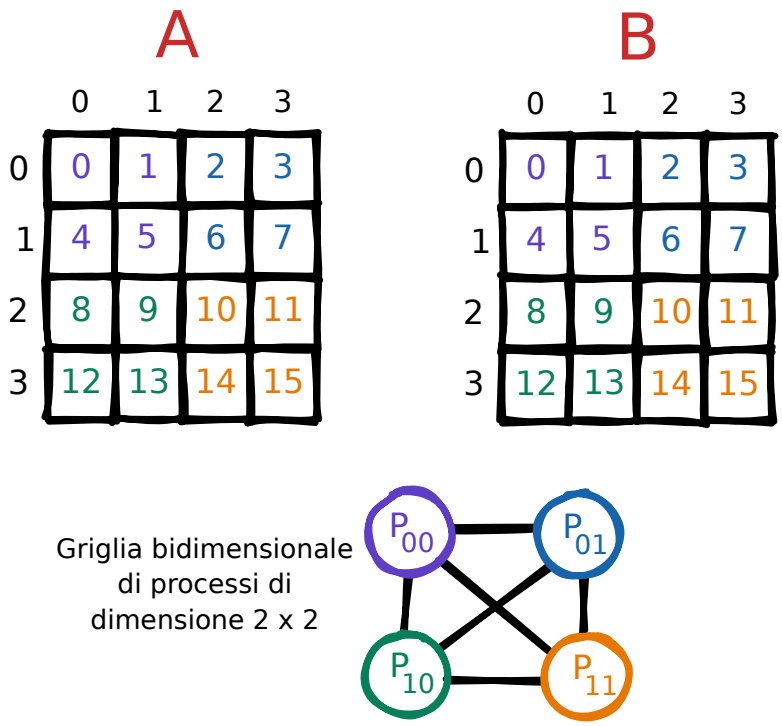


Figura 1: Esempio concreto di partizionamento e distribuzione delle matrici

2.1.2 Distribuzione delle sottomatrici di A presenti nella diagonale principale

Il calcolo del prodotto tra una matrice $\mathbf{A}$ e una matrice $\mathbf{B}$ può essere visto come la somma dei prodotti delle sottomatrici di $\mathbf{A}$ e $\mathbf{B}$. Ad esempio ponendo $m = 4$ e $p = 2$ e quindi una griglia di processi di dimensione 2×2, una volta partizionate e distribuite le matrici $\mathbf{A}$ e $\mathbf{B}$ ai processi come in Figura 1, la matrice risultante $\mathbf{C}$ del prodotto tra $\mathbf{A}$ e $\mathbf{B}$ può essere vista come:

$$\mathbf{C} = \begin{bmatrix} \mathbf{C}_{00} & \mathbf{C}_{01} \\ \mathbf{C}_{10} & \mathbf{C}_{11} \end{bmatrix}$$

dove:

$$\mathbf{C}_{00} = \mathbf{A}_{00} \cdot \mathbf{B}_{00} + \mathbf{A}_{01} \cdot \mathbf{B}_{10} \quad \mathbf{C}_{01} = \mathbf{A}_{00} \cdot \mathbf{B}_{01} + \mathbf{A}_{01} \cdot \mathbf{B}_{11}$$
$$\mathbf{C}_{10} = \mathbf{A}_{10} \cdot \mathbf{B}_{00} + \mathbf{A}_{11} \cdot \mathbf{B}_{10} \quad \mathbf{C}_{11} = \mathbf{A}_{10} \cdot \mathbf{B}_{10} + \mathbf{A}_{11} \cdot \mathbf{B}_{11}$$

Essenzialmente vogliamo che ogni processo p_{ij} calcoli il proprio $\mathbf{C}_{ij}$. Notiamo però che ogni processo p_{ij} ha inizialmente solo le sottomatrici $\mathbf{A}_{ij}$ e $\mathbf{B}_{ij}$. Possiamo subito notare che una componente delle sottomatrici $\mathbf{C}_{ij}$ tale per cui $i = j$ (quindi le sottomatrici sulla diagonale principale) può essere già calcolata. Ad esempio

il processo p_{00} può già calcolare $\mathbf{A}_{00} \cdot B_{00}$. Allo stesso modo il processo p_{11} può già calcolare $\mathbf{A}_{11} \cdot \mathbf{B}_{11}$. Un'osservazione più importante è che i processi che sono situati sulla stessa riga, hanno bisogno tutti della sottomatrice A_{kk} dove k è la $k-$esima riga. Ad esempio il processo p_{01} in riga $k = 0$ ha bisogno di $\mathbf{A}_{00}$ per eseguire il calcolo di una sua componente $\mathbf{A}_{00} \cdot \mathbf{B}_{01}$. Allo stesso modo il processo p_{10} in riga $k = 1$ ha bisogno della sottomatrice $\mathbf{A}_{11}$ per calcolare una componente della matrice $\mathbf{C}_{11}$.

Il primo passo dell'algoritmo BMR è quello di distribuire a tutti i processi in riga k, la sottomatrice $\mathbf{A}_{kk}$. In questo modo, dopo tale distribuzione, tutti i processi p_{ij} possono già calcolare una componente della propria soluzione parziale $\mathbf{C}_{ij}$.

La responsabilità di tale distribuzione è stata affidata alla funzione `distribute_main_diagonal_blocks` che è la seguente:

```
MatrixBlock *distribute_main_diagonal_blocks(
    const MPI_Comm *comm_grid,
    MatrixBlock *local_matrix_A
) {
    int dims[2], periods[2], coords[2];
    MPI_Cart_get(*comm_grid, 2, dims, periods, coords);

    int rank;
    MPI_Cart_rank(*comm_grid, coords, &rank);

    MPI_Comm *newcomm_rows = (MPI_Comm*) malloc(sizeof(MPI_Comm));
    const int remain_dims[2] = {false, true};
    MPI_Cart_sub(*comm_grid, remain_dims, newcomm_rows);
    int newcomm_rows_rank;
    MPI_Comm_rank(*newcomm_rows, &newcomm_rows_rank);

    int roots[dims[0] * dims[1]];
    int count = 0;
    for (int i = 0; i < dims[0]; i++) {
        for (int j = 0; j < dims[1]; j++) {
            roots[count] = i;
            count++;
        }
    }

    const int size_matrix = (local_matrix_A->rows) * (local_matrix_A->columns);
    int *matrix = (roots[rank] == newcomm_rows_rank)
        ? local_matrix_A->data
        : (int*) calloc(sizeof(int), size_matrix);

    MPI_Bcast(
        matrix,
        size_matrix,
        MPI_INT,
        roots[rank],
        *newcomm_rows
    );

    MatrixBlock *matrix_block = (MatrixBlock*)malloc(sizeof(MatrixBlock));
    matrix_block->columns = local_matrix_A->columns;
    matrix_block->rows = local_matrix_A->rows;
    matrix_block->data = matrix;
    return matrix_block;
}
```

La funzione riceve in input:

- `const MPI_Comm *comm_grid`: la griglia bidimensionale di processi
- `MatrixBlock *local_matrix_A`: la sottomatrice $\mathbf{A}_{ij}$ assegnata inizialmente al processo chiamante p_{ij}

La funzione restituisce la matrice $\mathbf{A}_{kk}$ per il processo chiamante situato nella riga k nella griglia bidimensionale di processi. Ad esempio, se p_{01} chiama tale funzione, esso riceverà in output la matrice $\mathbf{A}_{00}$. Per fare ciò, la funzione suddivide la griglia di processi in due communicator di riga indipendenti, come rappresentato di seguito:

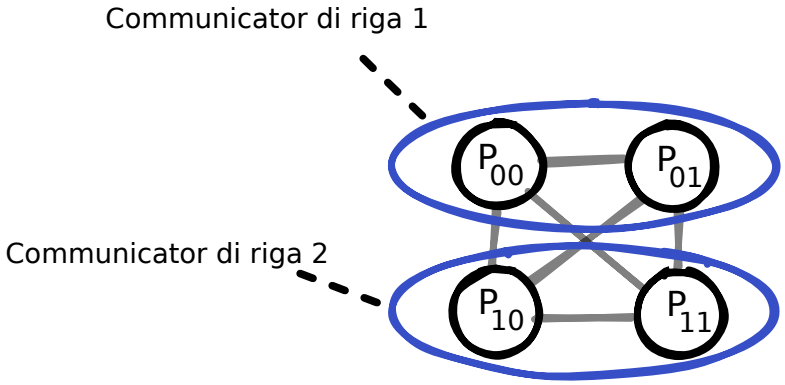


Figura 2: Divisione della griglia di processi in subcommunicators di righe

Dopodichè, ogni processo situato sulla diagonale principale effettua un `MPI_Broadcast` all'interno del proprio communicator di riga distribuendo di fatto la propria sottomatrice $\mathbf{A}_{kk}$ dove k è la riga in cui il processo sulla diagonale principale è situato. Tutti gli altri riceveranno tale matrice corrispondente alla riga in cui si trovano.

A questo punto, tutti i processi possono già calcolare una componente della propria soluzione parziale. Infatti, dopo aver chiamato la funzione `distribute_main_diagonal_blocks`, tutti i processi calcolano una propria componente:

```
// nel main...
// Start BROADCAST MULTIPLY ROLLING STEP 0
// Each block of matrix A that is on the main diagonal is distributed to
// all processes that are on the same row of the block.
MatrixBlock *local_main_diagonal_matrix_block_A =
    distribute_main_diagonal_blocks(
        comm_grid,
        local_matrix_block_A
    );

// Calculates a component of the local solution
MatrixBlock *local_result_matrix = multiply_matrices(
    local_main_diagonal_matrix_block_A,
    local_matrix_block_B
);
```

2.1.3 Distribuzione sottomatrici mancanti necessarie per calcolare la soluzione finale

Per permettere a tutti i processi p_{ij} di calcolare le restanti componenti della propria soluzione parziale $\mathbf{C}_{ij}$ vengono adesso fatte $m-1$ iterazioni (dove m è il numero di righe o colonne delle matrici quadrate). Ad ogni iterazione $k = 1, \ldots, m-1$ vengono fatte le seguenti operazioni:

- Viene ripetuta l'operazione di distribuzione di una sottomatrice di $\mathbf{A}$ esattamente come è stata fatta nella sezione precedente con l'unica differenza che la diagonale considerata in questo caso non è più quella principale, ma è la diagonale che dista k volte verso l'alto dalla diagonale principale. Ad esempio, se ci troviamo alla prima iterazione $k = 1$, la diagonale considerata sarà la seguente:

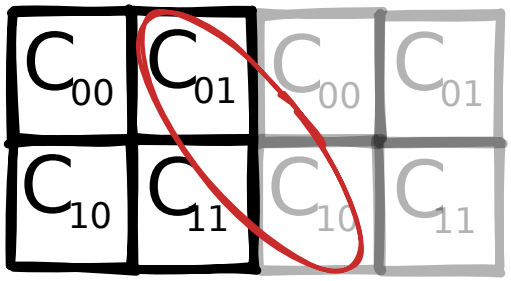


Figura 3: Iterazione $k = 1$ diagonale considerata

  Quindi, all'iterazione $k = 1$, non saranno più i processi p_{00} e p_{11} a distribuire le proprie sottomatrice $\mathbf{A}$ nei propri communicator di riga corrispondenti, ma saranno invece i processi p_{01} e p_{10} che distribuiranno la propria sottomatrice $\mathbf{A}$ (in questo caso $\mathbf{A}_{01}$ e $\mathbf{A}_{10}$ rispettivamente) nei corrispondenti communicator di riga. Essendo il numero di righe in questo caso uguale a 2 (numero di righe della griglia di processi), non può esserci un'altra iterazione.

- Tutti i processi p_{ij} inviano la propria corrente sottomatrice $\mathbf{B}_{ij}$ alla processo immediatamente sopra di esso (stessa colonna) nella griglia bidimensionale di processi. Se la riga è 0, viene considerato il processo nell'ultima riga e nella stessa colonna. Tutti i processi quindi ricevono una sottomatrice di $\mathbf{B}$ dal processo immediatamente sotto (stessa colonna) nella griglia quadrata di processi. Se la riga è l'ultima, il processo riceve dal processo nella prima riga e nella stessa colonna. Nelle prossime iterazioni, i processi invieranno i blocchi di $\mathbf{B}$ ricevuti nello step precedente.

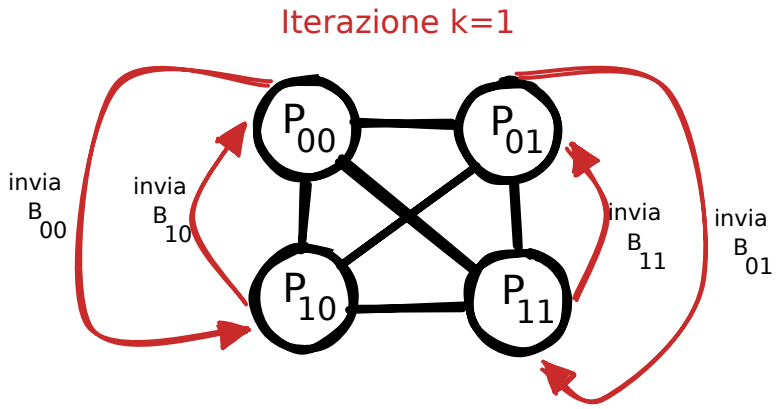


Figura 4: Iterazione $k = 1$ invio e ricezione sottoblocchi di $\mathbf{B}$

Queste due operazioni sono implementate dalla funzione `broadcast_rolling`:

```
MatrixBlock *broadcast_rolling(
    const int step,
    const MPI_Comm *comm_grid,
    MatrixBlock *local_matrix_A,
    MatrixBlock *current_matrix_B
) {
```

```
int dims[2], periods[2], coords[2];
MPI_Cart_get(*comm_grid, 2, dims, periods, coords);

int rank;
MPI_Cart_rank(*comm_grid, coords, &rank);

MPI_Comm *newcomm_rows = (MPI_Comm*) malloc(sizeof(MPI_Comm));
const int remain_dims[2] = {false, true};
MPI_Cart_sub(*comm_grid, remain_dims, newcomm_rows);
int newcomm_rows_rank;
MPI_Comm_rank(*newcomm_rows, &newcomm_rows_rank);

int roots[dims[0] * dims[1]];
int count = 0;
for (int i = 0; i < dims[0]; i++) {
    for (int j = 0; j < dims[1]; j++) {
        roots[count] = (i + step) % dims[0];
        count++;
    }
}

const int size_matrix = (local_matrix_A->rows) * (local_matrix_A->columns);
int *matrix_A = (roots[rank] == newcomm_rows_rank)
  ? local_matrix_A->data
  : (int*) malloc(sizeof(int) * size_matrix);

MPI_Bcast(
    matrix_A,
    size_matrix,
    MPI_INT,
    roots[rank],
    *newcomm_rows
);

MPI_Comm *newcomm_columns = (MPI_Comm*) malloc(sizeof(MPI_Comm));
const int columns_remain_dims[2] = {true, false};
MPI_Cart_sub(*comm_grid, columns_remain_dims, newcomm_columns);

const int size_matrix_B = current_matrix_B->rows * current_matrix_B->columns;

int newcomm_columns_rank;
MPI_Comm_rank(*newcomm_columns, &newcomm_columns_rank);

const static int TAG = 80;
MPI_Request *mpi_request = (MPI_Request*) malloc(sizeof(MPI_Request));

const int previous_process_rank = abs(newcomm_columns_rank - 1) % dims[0];

MPI_Isend(
    current_matrix_B->data,
    size_matrix_B,
    MPI_INT,
    previous_process_rank,
    TAG + newcomm_columns_rank,
    *newcomm_columns,
    mpi_request
);

const int next_process_rank = (newcomm_columns_rank + 1) % dims[0];
int *matrix_B = (int*) malloc(sizeof(int) * size_matrix_B);
MPI_Recv(
```

```
        matrix_B,
        size_matrix_B,
        MPI_INT,
        next_process_rank,
        TAG + next_process_rank,
        *newcomm_columns,
        MPI_STATUSES_IGNORE
    );

    MPI_Request_free(mpi_request);

    MatrixBlock *matrix_blocks = (MatrixBlock*) malloc(sizeof(MatrixBlock) * 2);
    matrix_blocks[0].data = matrix_A;
    matrix_blocks[0].rows = local_matrix_A->rows;
    matrix_blocks[0].columns = local_matrix_A->columns;
    matrix_blocks[1].data = matrix_B;
    matrix_blocks[1].rows = current_matrix_B->rows;
    matrix_blocks[1].columns = current_matrix_B->columns;
    return matrix_blocks;
}
```

La funzione riceve in input:

- `const int step`: lo step (iterazione) corrente. Deve essere minore del numero di righe o colonne delle matrici quadrate.
- `const MPI_Comm *comm_grid`: la griglia bidimensionale di processi
- `MatrixBlock *local_matrix_A`: la sottomatrice di **A** originariamente distribuita al processo chiamante
- `MatrixBlock *current_matrix_B`: la sottomatrice corrente di **B** distribuita al processo chiamante al precedente step (o quella distribuita originariamente se `step = 1`).

La funzione in output restuisce un array di `MatrixBlock`. In posizione 0 ci sarà la sottomatrice di **A** per il processo chiamante distribuita dal processo situato nella stessa riga (nella griglia bidimensionale di processi) che in quel momento si trovava nella diagonale considerata. In posizione 1 ci sarà la sottomatrice di **B** distribuita dal processo appena sotto al processo chiamante all'interno della griglia di processi.

Queste due componenti, ad ogni iterazione, permettono al processo chiamante di calcolare una nuova componente per la sua soluzione locale. Il processo iterativo appena descritto viene fatto all'interno del `main()`:

```
// At each new step, the diagonal that is #step times upward from the main diagonal is considered.
// Then the same operation as in step 0 is repeated.
// In addition, each process passes its current block of B to
// the process located just above it in the process grid.
MatrixBlock *current_matrix_B = local_matrix_block_B;
for (int step = 1; step < comm_grid_dim; step++) {
    MatrixBlock *matrix_blocks = broadcast_rolling(
        step,
        comm_grid,
        local_matrix_block_A,
        current_matrix_B
    );
    current_matrix_B = &matrix_blocks[1];
    // At each step a new component of the local solution is calculated,
    // and it is added to the total local solution
    local_result_matrix = sum_matrices(
        local_result_matrix,
        multiply_matrices(&matrix_blocks[0], current_matrix_B)
    );
}
```

Prima del ciclo `for`, `current_matrix_B` è la sottomatrice originariamente distribuita al processo chiamante. Ad ogni nuova iterazione, viene considerata invece la sottomatrice **B** ricevuta nello step precedente. Ad ogni iterazione, viene calcolata una nuova componente e sommata alla soluzione parziale finale. Alla fine ogni

processo avrà la propria soluzione finale. Adesso possiamo scegliere di distribuire tutte le sottosoluzioni ad un unico processo, o semplicemente fermarci qui e stampare sullo stdout o inserire in un file tutte le soluzioni parziali.

2.1.4 Distribuzione delle soluzioni parziali

Tutte le soluzioni parziali calcolate da ogni processo verranno inviate e raccolte da un unico processo. Poniamo tale processo `root` uguale a 0. Il programma definisce il processo radice come una variabile globale:

```
    const static int root = 0;
```

La funzione che si occupa di inviare tutte le soluzioni parziali al processo `root` è la seguente:

```
MatrixBlock *gather_partial_solutions(
    const int pid_root,
    const MPI_Comm *comm_grid,
    MatrixBlock *partial_solution,
    const int global_matrix_rows,
    const int global_matrix_columns
) {
    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;

    int displs[total_processes];
    int recv_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 * global_matrix_columns * partial_solution->rows +
                    j * partial_solution->columns;
            recv_counts[i * comm_grid_total_columns + j] = 1;
        }
    }

    MPI_Datatype blocktype_not_resized;
    MPI_Datatype blocktype_resized;
    MPI_Type_vector(
            partial_solution->rows,
            partial_solution->columns,
            global_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 *total_solution_data = NULL;
    if (pid_comm_grid == pid_root) {
        total_solution_data = (int*) malloc(sizeof(int) * global_matrix_columns * global_matrix_rows);
    }

    MPI_Gatherv(
        partial_solution->data,
        partial_solution->rows * partial_solution->columns,
        MPI_INT,
        total_solution_data,
```

```
        recv_counts,
        displs,
        blocktype_resized,
        pid_root,
        MPI_COMM_WORLD
    );

    MatrixBlock *total_solution = NULL;
    if (pid_comm_grid == pid_root) {
        total_solution = (MatrixBlock*) malloc(sizeof(MatrixBlock));
        total_solution->rows = global_matrix_rows;
        total_solution->columns = global_matrix_columns;
        total_solution->data = total_solution_data;
    }
    return total_solution;
}
```

La funzione `gather_partial_solutions` prende in input:

- `pid_root`: il rank del processo a cui inviare tutte le soluzioni parziali
- `comm_grid`: la griglia di processi
- `partial_solution`: la soluzione parziale del processo che invoca tale funzione
- `global_matrix_rows`: il numero di righe della matrice risultato $\mathbf{C}$
- `global_matrix_columns`: il numero di colonne della matrice risultato $\mathbf{C}$

La funzione in output restiuirà la matrice $\mathbf{C}$ per il processo con rank `pid_root`. Tutti gli altri processi riceveranno `NULL`.

Supponiamo di aver eseguito già la moltiplicazione tra due matrici quadrate $\mathbf{A}$ e $\mathbf{B}$ di dimensione 4×4 utilizzando una griglia di processi di dimensione 2×2. Ogni processo avrà quindi calcolato la propria soluzione parziale $\mathbf{C}_{ij}$ dove la coppia (i, j) (dove $i = 0, \ldots, 1$ e $j = 0, \ldots, 1$) rappresenta le coordinate di ogni processo all'interno della griglia dei processi. La funzione `gather_partial_solutions` verrà chiamata da ogni processo. Ogni processo darà in input alla funzione la propria soluzione parziale $\mathbf{C}_{ij}$ e tutte le soluzioni parziali verranno appropriatamente inviate al processo `root` nell'ordine giusto. Il seguente diagramma mostra quello che effettivamente succede:

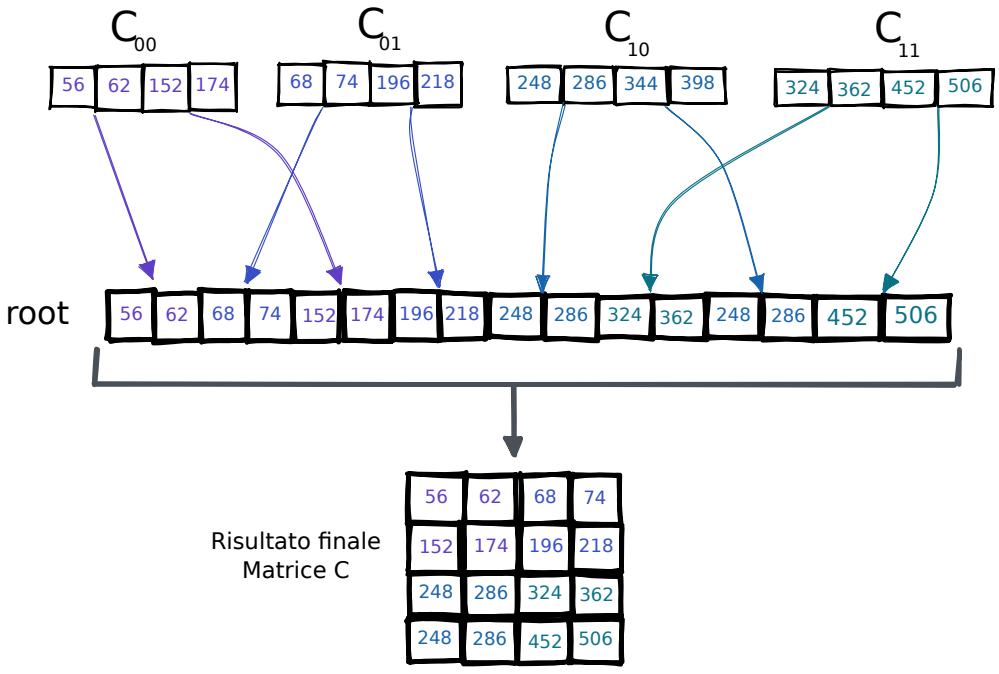


Figura 5: Aggregazione dei risultati parziali

Per facilitare l'aggregamento delle soluzioni parziali, la funzione `gather_partial_solutions` utilizza un tipo di dato vettore personalizzato utilizzando la funzione `MPI_Type_vector`. La seguente funzione prende in input:

- `count`: il numero di blocchi
- `blocklength`: il numero di elementi contenuti in ogni blocco
- `stride`: il numero di elementi tra l'inizio di ogni blocco (dopo quanti elementi inizia il prossimo blocco)

Il numero di blocchi sarà uguale al numero di righe della matrice quadrata che rappresenta la soluzione parziale. Il numero di elementi contenuti in ogni blocco sarà il numero di colonne della matrice quadrata che rappresenta la soluzione parziale. Lo stride (dopo quanti elementi inizia il prossimo blocco) è il numero di colonne della matrice totale (e non quella parziale). Quando un vettore viene creato e inserito nel vettore finale `total_solution_data` (che ha dimensione maggiore del vettore creato), il prossimo vettore verrà inserito a partire dalla fine del vettore precedente. Questo è chiaramente un problema perché oltre a non dare un risultato corretto, ci porterà anche fuori dallo spazio di memoria allocato per l'array. Quello che vogliamo invece è che il prossimo vettore venga inserito a partire dalla fine del primo blocco del vettore precedente. Per fare questo è possibile utilizzare la funzione `MPI_Type_create_resized`. Tale funzione prende in input:

- `oldtype`: il tipo di vettore che vogliamo ridimensionare
- `lb`: il nuovo limite inferiore del tipo di dato vettore
- `extent`: la nuova estensione del tipo di dato

Settando `lb = 0` e `extent = sizeof(int)` l'aggregazione delle soluzioni avverrà correttamente.

La funzione che effettivamente aggrega i dati in un unico processo è la funzione `MPI_Gatherv`. Con tale funzione, possiamo dire a tutti i processi di inviare dei dati a un particolare processo. La funzione prende in input:

- `sendbuf`: l'indirizzo iniziale di memoria del dato da inviare (ogni processo può settare il proprio indirizzio iniziale di memoria)
- `sendcount`: il numero di elementi all'interno del buffer di invio (quanti elementi bisogna inviare a partire dall'indirizzo iniziale fornito in `sendbuf`)
- `sendtype`: il tipo di dato da inviare. Può essere ad esempio `MPI_INT` o un tipo di dato personalizzato
- `recvcounts`: (significante solo per il processo `root`) il numero di elementi che il processo `root` si aspetta da ogni processo (tale parametro è un array di interi e deve avere la stessa dimensione del numero di processi)
- `recvtype`: il tipo di dato che il processo `root` utilizzerà per inserire i dati ricevuti all'interno di `sendbuf`. Tale tipo sarà proprio il tipo di dato personalizzato creato.
- `root`: il processo che riceverà tutti i dati
- `comm`: il communicator

2.2 Come avviare il programma sul cluster SCoPE

Per eseguire l'algoritmo BMR sul cluster SCoPE, bisogna recercasi nella cartella `broadcast-multiply-rolling` e avviare lo script `start_pbs.sh` che eseguirà il file pbs `broadcast_multiply_rolling.pbs`. Per eseguire lo script, digitare il seguente comando. Lo script vi chiederà anche se si vuole stampare il risultato finale in `broadcast_multiply_rolling.out` oppure stampare soltanto i tempi di esecuzione dell'algoritmo parallelo e di quello sequenziale:

```
$ ./start_pbs.sh
```

Lo script chiederà di inserire la dimensione delle matrici quadrate **A** e **B**. Ad esempio, se scegliamo 200, verranno create due matrici di dimensione 200×200:

```
The number of processes is set equal to 4.
The grid of processes has size 2x2.
You can change these settings by changing the env variables in the .env file.

Enter the dimension of the square matrix: 200
```

```
Do you want print the matrix A, B and all local results on the stdout? (Y/N): N

EXECUTING BROADCAST PROCESS MATRIX
4007354.torque02.scope.unina.it

Il pbs broadcast_multiply_rolling.pbs e' stato accodato.
I risultati saranno stampati in broadcast_multiply_rolling.pbs.out e \
    broadcast_multiply_rolling.pbs.err.
I risultati sulle performance verranno memorizzatin in results.txt.
```

I settaggi di default sono un numero di processi uguale a 4 e una griglia di processi uguale a 2×2. E' possibile cambiare questi settaggi cambiando le variabili d'ambiente contenute nel file `.env`:

```
PROCESSES=4
GRID_COMM_DIM=2
PRINT_MODE="false"
```

La variabile `PRINT_MODE` è un flag booleano che attiva o disattiva la stampa del risultato finale.

2.3 Esempi di output

2.3.1 Esempio di output con stampa risultato finale

Eseguendo lo script `start_pbs.sh` con una dimensione uguale a 4, verranno create due matrici quadrate di dimensione 4×4. Scegliamo inoltre di voler stampare anche il risultato finale digitando Y:

```
The number of processes is set equal to 4.
The grid of processes has size 2x2.
You can change these settings by changing the env variables in the .env file.

Enter the dimension of the square matrix: 4
Do you want print the matrix A, B and all local results on the stdout? (Y/N): N

EXECUTING BROADCAST PROCESS MATRIX
4007354.torque02.scope.unina.it

Il pbs broadcast_multiply_rolling.pbs e' stato accodato.
I risultati saranno stampati in broadcast_multiply_rolling.pbs.out e \
    broadcast_multiply_rolling.pbs.err.
I risultati sulle performance verranno memorizzatin in results.txt.
```

Il risultato finale in `broadcast_multiply_rolling.out` sarà il seguente:

```
Matrix A:
  0   1   2   3
  4   5   6   7
  8   9  10  11
 12  13  14  15

Matrix B:
  0   1   2   3
  4   5   6   7
  8   9  10  11
 12  13  14  15

Matrix C:
 56  62  68  74
152 174 196 218
248 286 324 362
344 398 452 506

[PID 0 Coords (0, 0)] Local result:
 56  62
152 174
```

```
[PID 1 Coords (0, 1)] Local result:
 68  74
196 218

[PID 3 Coords (1, 1)] Local result:
324 362
452 506

[PID 2 Coords (1, 0)] Local result:
248 286
344 398

[TOTAL TIME PARALLEL] 9.570122e-04 seconds.
[TOTAL TIME SINGLE PROCESS] 0.000000e+00 seconds.
```

Con una dimensione così piccola è chiaro che l'iterativo è più veloce.

2.3.2 Esempio di output senza stampa risultato finale

Eseguiamo lo script `start_pbs.sh` senza stampare il risultato finale con una dimensione uguale a 200:

```
The number of processes is set equal to 4.
The grid of processes has size 2x2.
You can change these settings by changing the env variables in the .env file.

Enter the dimension of the square matrix: 200
Do you want print the matrix A, B and all local results on the stdout? (Y/N): N

EXECUTING BROADCAST PROCESS MATRIX
4007357.torque02.scope.unina.it

Il pbs broadcast_multiply_rolling.pbs e' stato accodato.
I risultati saranno stampati in broadcast_multiply_rolling.pbs.out e
 broadcast_multiply_rolling.pbs.err.
I risultati sulle performance verranno memorizzatin in results.txt.
```

Il file `broadcast_multiply_rolling.out` conterrà:

```
------------------------------
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/broadcast-multiply-rolling
PBS: execution mode is PBS_BATCH
PBS: job identifier is 4007358.torque02.scope.unina.it
PBS: job name is broadcast_multiply_rolling
PBS: node file is /var/spool/pbs/aux//4007358.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
----------------------------------------
```

```
[TOTAL TIME PARALLEL] 2.789593e-02 seconds.
[TOTAL TIME SINGLE PROCESS] 1.000000e-01 seconds.
```

2.4 Codice sorgente

Il codice sorgente può essere trovato nella mia repository di GitHub:

- https://github.com/vtramo/parallel-and-distributed-computing-experiments

Ci sono due branch: `main` per l'esecuzione in locale e `cluster` per l'esecuzione sul cluster SCoPE.

Grafici

3 Grafici del tempo d'esecuzione

In questa sezione verranno presentati i risultati raccolti dal punto di vista del tempo di esecuzione e dello speed-up dell'algoritmo parallelo Broadcast Multiply Rolling rispetto a quello sequenziale. Il cluster SCoPE offre 8 nodi, quindi l'unica soluzione possibile è stata quella di allocare 4 nodi e utilizzare una griglia di processori di dimensione 2×2. Gli esperimenti sono stati effettuati con questi settaggi. I risultati ottenuti sono i seguenti (riportati prima in forma tabellare):

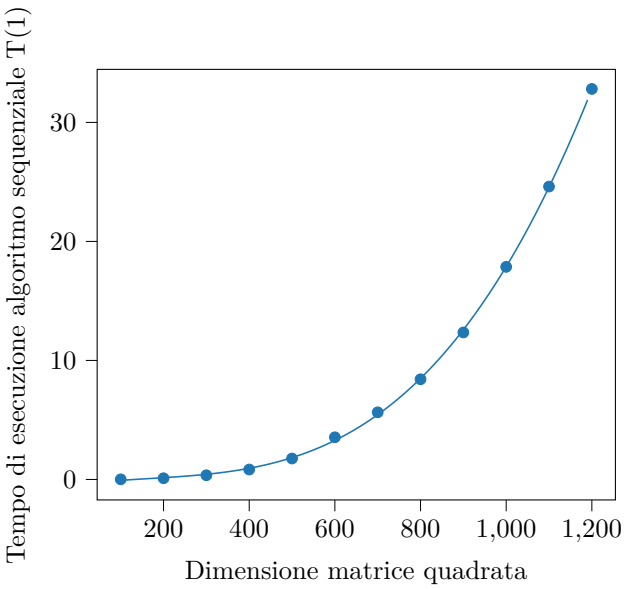
Table 1: Tabella dei risultati raccolti

Tempo di esecuzione algoritmo parallelo su 4 processori T(4) in secondi	**Tempo di esecuzione algoritmo sequenziale su un singolo processore T(1) in secondi**	**Speed-up T(1)/T(p)**	**Processori totali**	**Dimensione griglia dei processi**	**Dimensione matrice quadrata**	**Numero totale di iterazioni eseguite per ottenere i risultati (media dei risultati)**
0.004534	0.01	2.20556	4	2x2	100	10
0.0279425	0.101	3.61457	4	2x2	200	10
0.901941	0.351	3.89161	4	2x2	300	10
0.211356	0.834	3.94595	4	2x2	400	10
0.411168	1.759	4.27806	4	2x2	500	10
0.711377	3.544	4.98189	4	2x2	600	10
1.13656	5.641	4.96322	4	2x2	700	10
1.69032	8.418	4.98012	4	2x2	800	10
2.47896	12.352	4.98273	4	2x2	900	10
3.56554	17.863	5.0099	4	2x2	1000	10
5.27153	24.609	4.66828	4	2x2	1100	10
7.11314	32.812	4.61287	4	2x2	1200	10

La dimensione della matrice quadrata nella tabella rappresenta il numero di righe o colonne. Ad esempio 200 vuol dire che le matrici **A** e **B** hanno dimensione 200×200. Come già riportato in Sezione 2.1.1, tali matrici, per semplicità, vengono generate automaticamente dal programma. Prima di procedere con l'algoritmo, vengono create e riempite le matrici **A** e **B** in maniera deterministica. Ad esempio se la dimensione delle matrici quadrate è 4, allora verrano create due matrici di dimensione 4×4 esattamente in questa maniera:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

3.1 Grafico tempo d'esecuzione algoritmo sequenziale su un singolo processore

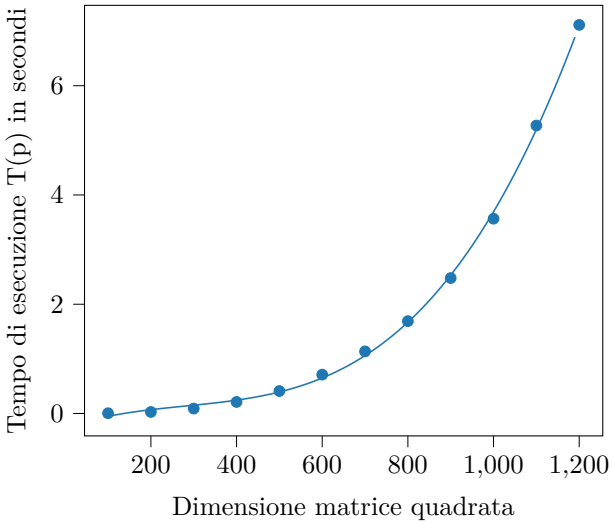


- **Asse delle ascisse**: dimensione matrice quadrata crescente

- **Asse delle ordinate**: tempo di esecuzione algoritmo sequenziale $T(1)$ su un singolo processore

All'aumentare della dimensione della matrice quadrata **A** e della matrice quadrata **B**, il tempo di esecuzione dell'algoritmo sequenziale aumenta drasticamente.

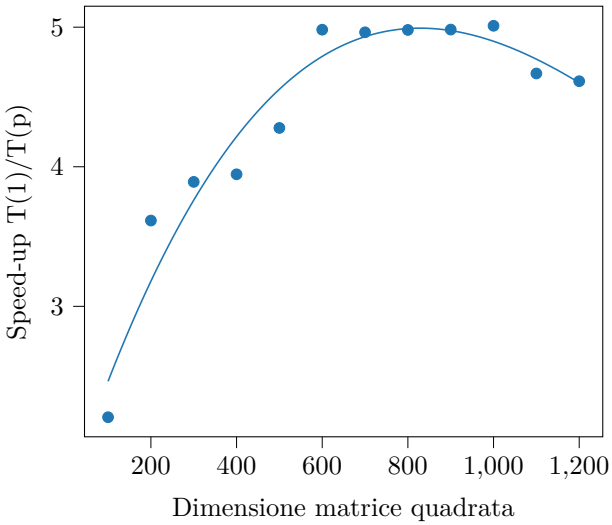
3.2 Grafico tempo d'esecuzione algoritmo parallelo



- **Asse delle ascisse**: dimensione matrice quadrata crescente
- **Asse delle ordinate**: tempo di esecuzione algoritmo parallelo $T(p)$ con $p = 4$ e griglia di processi di dimensione 2×2

All'aumentare delle dimensioni delle matrici quadrate **A** e **B**, abbiamo un aumento del tempo di esecuzione che è sicuramente più piccolo di quello sequenziale. Come riportato in Tabella 1, con una dimensione delle matrici uguale a 1200 (sarebbe a dire matrici quadrate di dimensioni 1200×1200), l'algoritmo parallelo impiega 7.11314 secondi mentre quello sequenziale impiega 32.812 secondi.

3.3 Grafico speed-up algoritmo parallelo rispetto all'algoritmo sequenziale



- **Asse delle ascisse**: dimensione matrice quadrata crescente
- **Asse delle ordinate**: rapporto $\frac{T(1)}{T(p)}$ con $p = 4$ (speed-up)

All'aumentare delle dimensioni delle matrici quadrate **A** e **B** abbiamo uno speed-up sempre crescente. Ad un certo punto possiamo notare che lo speed-up diventa quasi costante nel range di dimensioni $[600, 1000]$ per poi cominciare leggermente a decrescere.