UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

CORSO DI LAUREA IN INFORMATICA

Anno Accademico 2021/2022

Algoritmo per il prodotto Matrice-Matrice in MPI

Autore

Carmine Grimaldi

Matr. N97000394

Sommario

1. Definizione, analisi e descrizione del problema

Si vuole progettare un algoritmo su architettura $\mathcal{MIMD}$ a memoria distribuita, che utilizzi la libreria $\mathcal{MPI}$, per risolvere il prodotto scalare tra due matrici quadrate in parallelo su un numero p di processi. In particolare si utilizza l'infrastruttura $\mathcal{S.C.o.P.E.}$ per permettere l'esecuzione del software in un ambiente di calcolo parallelo.

Più nel dettaglio, l'algoritmo implementato nel file elaborato3.c allegato a questa documentazione (e illustrato nel capitolo 4 di questa documentazione), implementa la strategia di comunicazione $\mathcal{BMR}$ (Broadcast, Multiply, Rolling), la quale prevede la decomposizione delle matrici di input in blocchi quadrati, ciascuno di tali blocchi verrà poi assegnato ai processori disposti lungo una griglia bidimensionale periodica capace di distribuire una matrice $A \in \Re^{mxm}$ e una matrice $\mathrm{B} \in \Re^{mxm}$ scorporandola in pxp processi (per valori di m multipli di p).

La strategia è costituita da p passi. Si parte dalla diagonale principale della griglia di processori; ad ogni passo k, si considera la k-ma diagonale situata al di sopra di quella principale. I processori situati lungo la diagonale effettuano una comunicazione collettiva del blocco della matrice A in loro possesso a tutti i processori della medesima riga. Inoltre, ad ogni passo, ciascun processore effettua una comunicazione del proprio blocco della matrice B al processore situato nella stessa colonna e nella riga precedente.

Si è scelto di misurare i tempi di esecuzione nel processo 0 usando la primitiva `MPI_Wtime()` prima e dopo il calcolo parallelo.

2. Guida alla compilazione e manuale d'uso

Di seguito è illustrato il modo con cui è possibile invocare l'eseguibile allegato a questa relazione.

```
qsub libreria3.pbs -v M=[matrix_size]
```

All'interno dello script "`libreria3`.pbs" è presente l'invocazione vera e propria dell'algoritmo (scritto in linguaggio C).

Passaggio dei parametri

`M`: indica il numero di righe, nonché di colonne, della matrice quadrata che si desidera utilizzare. Tale numero deve essere necessariamente positivo.

In output sarà fornito un messaggio che stamperà le due matrici quadrate (nel codice fornito in linguaggio C tali matrici vengono inizializzate automaticamente) di cui calcolare il prodotto scalare e la matrice risultato di tale prodotto (la stampa avviene soltanto se le matrici hanno una dimensione non superiore a 10x10); inoltre viene riportato quanto tempo è stato impiegato per ottenere il risultato mostrato a schermo, la dimensione delle matrici e il numero di processi che si è deciso di utilizzare.

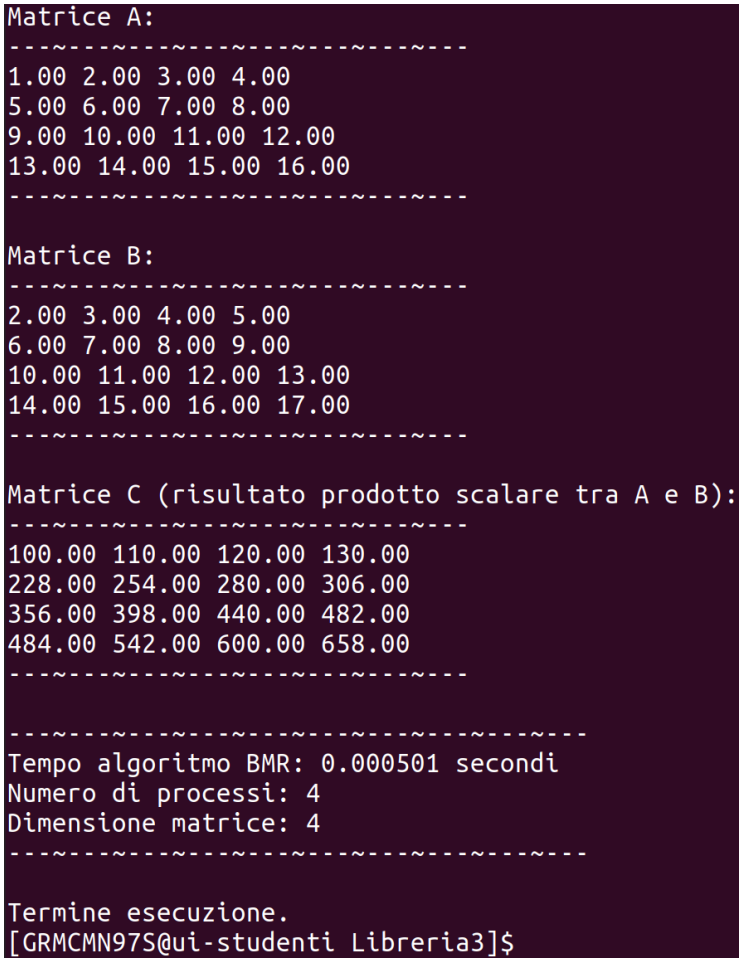
In allegato a questa documentazione è presente la cartella "Output", in cui sono contenuti alcuni file di output di esempio.

Input, output e condizioni di errore

- **Input**: le due matrici di cui effettuare il prodotto scalare e la loro dimensione.
- **Output**: la matrice risultato del prodotto scalare tra le due matrici date in input.
- **Condizioni di errore**: la dimensione delle matrici deve essere un numero intero positivo; inoltre, tale dimensione deve essere divisibile per la dimensione della griglia dei processi, in accordo con il numero di processi che si è scelto di utilizzare. Infine, il numero di processori deve essere tale da generare una griglia quadrata.

Esempio di funzionamento

Nell'immagine seguente vi è un esempio di funzionamento, con 4 processi e una dimensione delle matrici pari a 4x4.



Esempi di errore

Nelle successivi immagini, invece, sono mostrati i messaggi di errore al verificarsi delle condizioni precedentemente citate.

```
Eseguo: /usr/lib64/openmpi/1.4-gcc/bin/mpicc -o /homes/DMA/PDC/2021/
GRMCMN97S/Libreria3/libreria3 /homes/DMA/PDC/2021/GRMCMN97S/Libreria
3/libreria3.c
Eseguo: /usr/lib64/openmpi/1.4-gcc/bin/-machinefile hostlist -np 4 /
homes/DMA/PDC/2021/GRMCMN97S/Libreria3/libreria3 0

Errore: matrix_size (=0) deve essere un numero positivo!

Termine esecuzione.
[GRMCMN97S@ui-studenti Libreria3]$
```

Errore: dimensione matrici non positiva

```
Eseguo: /usr/lib64/openmpi/1.4-gcc/bin/mpicc -o /homes/DMA/PDC/2021/
GRMCMN97S/Libreria3/libreria3 /homes/DMA/PDC/2021/GRMCMN97S/Libreria
3/libreria3.c
Eseguo: /usr/lib64/openmpi/1.4-gcc/bin/-machinefile hostlist -np 4 /
homes/DMA/PDC/2021/GRMCMN97S/Libreria3/libreria3 1

Errore: la dimensione della matrice (=1) non e' divisibile
        per la dimensione della griglia dei processi (=4)!

Termine esecuzione.
[GRMCMN97S@ui-studenti Libreria3]$
```

Errore: dimensione matrice non divisibile per la dimensione della griglia dei processi

```
Eseguo: /usr/lib64/openmpi/1.4-gcc/bin/mpicc -o /homes/DMA/PDC/2021/
GRMCMN97S/Libreria3/libreria3 /homes/DMA/PDC/2021/GRMCMN97S/Libreria
3/libreria3.c
Eseguo: /usr/lib64/openmpi/1.4-gcc/bin/-machinefile
hostlist -np 3 /homes/DMA/PDC/2021/GRMCMN97S/Libreria3/libreria3 100

Errore: '-np 3' non e' un quadrato perfetto!
Termine esecuzione.
[GRMCMN97S@ui-studenti Libreria3]$
```

Errore: il numero dei processi non consente la creazione di una griglia di processi quadrata

Funzioni

void grid_init (GridInfo[1] *grid) :
funzione che, data in input una struttura contenente tutte le informazioni utili a una griglia di processi, inizializza tale griglia e, tra le varie inizializzazioni, crea e memorizza in tale struttura un communicator di righe e un communicator di colonne.

void matrix_creation (double **pA, double **pB, double **pC, int size) :
funzione che si occupa di allocare uno spazio di memoria quadratico rispetto a "size" (moltiplicato per la dimensione del tipo double), per ogni puntatore a matrice dato in input.

void matrix_init (double *A, double *B, int size) :
funzione che facilita il test del prodotto scalare inizializzando in maniera predefinita le matrici A e B di cui andare a calcolare il prodotto scalare.

void matrix_dot (double *A, double *B, double *C, int n) :
funzione che, date in ingresso due matrici A e B, ne calcola il prodotto scalare e lo memorizza nella matrice C.

void matrix_print (double *A, int n) :
funzione che stampa la matrice "A" di valori reali di dimensione "n" che è stata data in input.

void matrix_free (double **pA, double **pB, double **pC) :
funzione che libera la memoria per tutte le matrici date in input.

void BMRAlgorithm (double *A, double *B, double *C, int size, GridInfo *grid) :
funzione che implementa il prodotto scalare di due matrici "A" e "B" in parallelo, utilizzando la libreria **MPI**, basandosi sulla strategia **BMR** (Broadcast, Multiply, Rolling) descritta nel capitolo 1 di questa documentazione.
In input prende le due matrici "A" e "B" di cui calcolare il prodotto scalare, una matrice "C" in cui andare a memorizzare il risultato finale, la dimensione "size" di tutte le matrici quadrate, e infine un puntatore a struttura "grid" che contiene tutte le informazioni utili sulla griglia dei processi che è stata opportunamente inizializzata.

[1] Per maggiori informazioni sulla struttura GridInfo si rimanda al codice illustrato nel capitolo 4 di questa documentazione.

3. Analisi delle prestazioni

Si è scelto di misurare i tempi di esecuzione utilizzando la primitiva `MPI_Wtime()` prima e dopo il calcolo parallelo.

Siccome attraverso il cluster *S.C.o.P.E.* abbiamo a disposizione al massimo 8 nodi, e vogliamo che ogni processo sia su un nodo, ricordando che le matrici in input sono quadrate, abbiamo bisogno di un numero quadrato di processi, ma l'unico quadrato possibile per il cluster sarebbe 2^2, poiché già a partire da 3^2 si arriva a 9, e non avendo a disposizione 9 processi, ma 8, l'unico quadrato che possiamo utilizzare è 2^2 ossia 4.

Per tale ragione, i test che sono illustrati di seguito, fanno riferimento solo ed esclusivamente a una strategia di parallelizzazione in cui varia la dimensione delle matrici ma non dei processi, i quali sono fissati a 4, e diventano 1 soltanto nel caso in cui viene calcolato il tempo T(1) con un solo processore ai fini del calcolo dello speedUp.

Si noti che, essendo nel caso di matrici quadrate, si è scelto di limitare l'aumento della dimensione delle matrici a 2000x2000, data l'impossibilità di eseguire sul cluster (rientrando nei limiti di tempo di esecuzione consentiti) il programma con dimensioni superiori.

I grafici e le tabelle illustrati nelle pagine seguenti riassumono i risultati ottenuti. Successivamente sono presenti delle conclusioni per la valutazione dei risultati e il commento dei grafici di seguito illustrati.

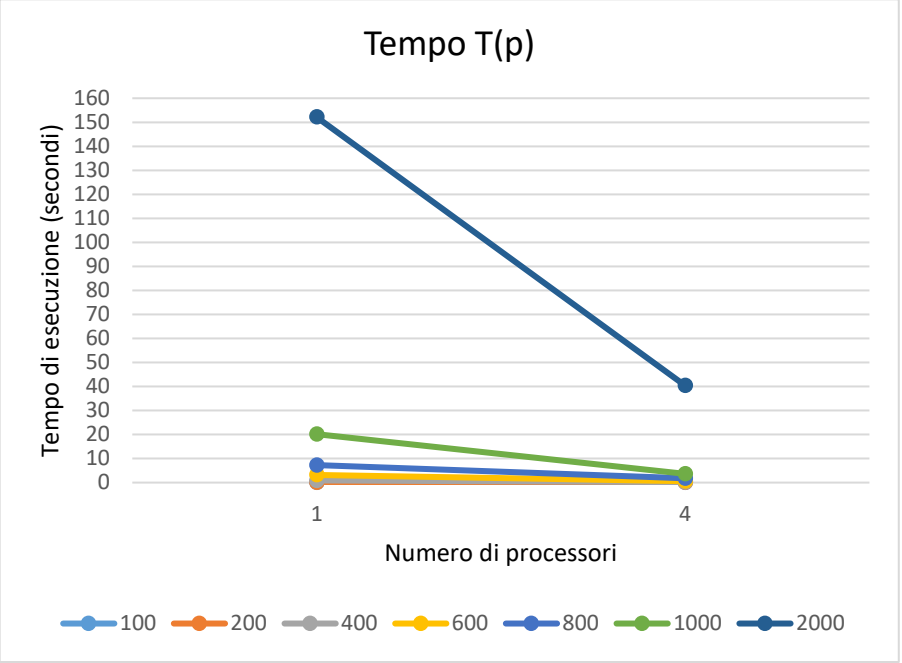
Una media di 5 esecuzioni è stata eseguita ai fini del calcolo dei tempi medi illustrati.

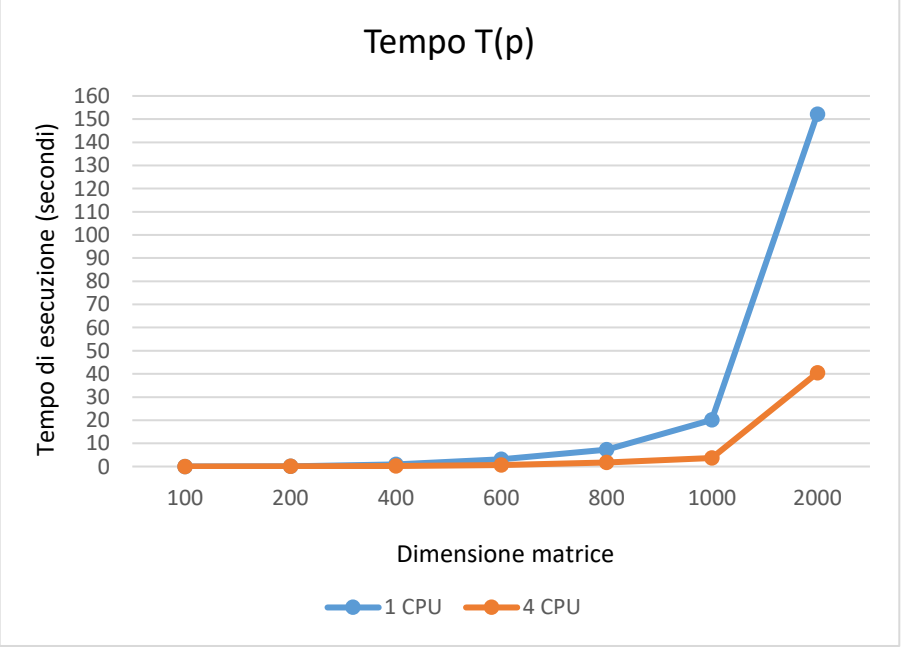
Inoltre, la precisione numerica utilizzata per i valori delle matrici è quella del tipo double.

Tempi di esecuzione T(p)

Tabella dei tempi di esecuzione:

Numero CPU	Dimensione input						
	100	200	400	600	800	1.000	2.000
1	0,012104	0,085372	0,893013	3,176393	7,284349	20,152027	152,155364
4	0,004053	0,023208	0,169839	0,575580	1,792697	3,633251	40,379482





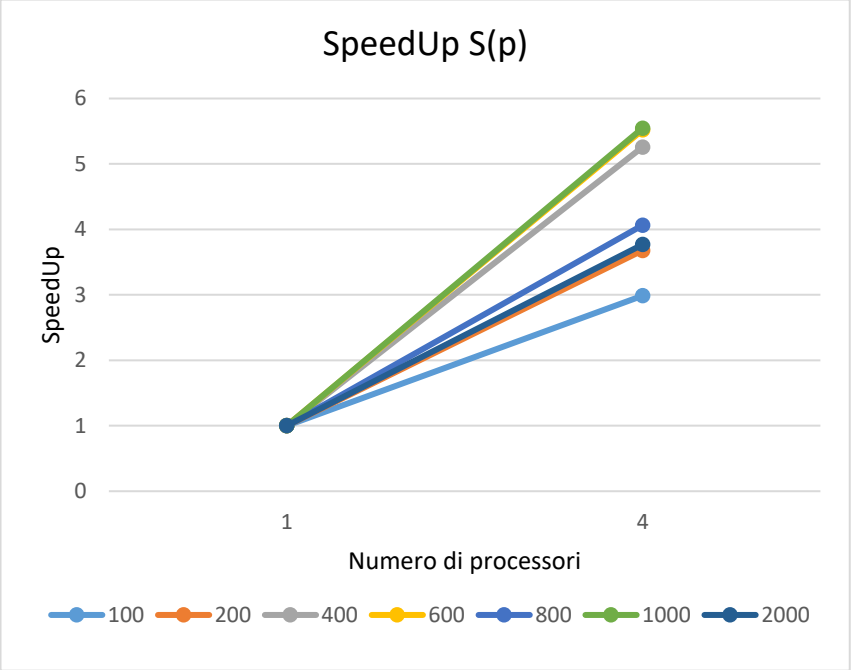
Per ulteriori considerazioni su questi risultati si rimanda alla sezione *"Conclusioni"* di questo capitolo.

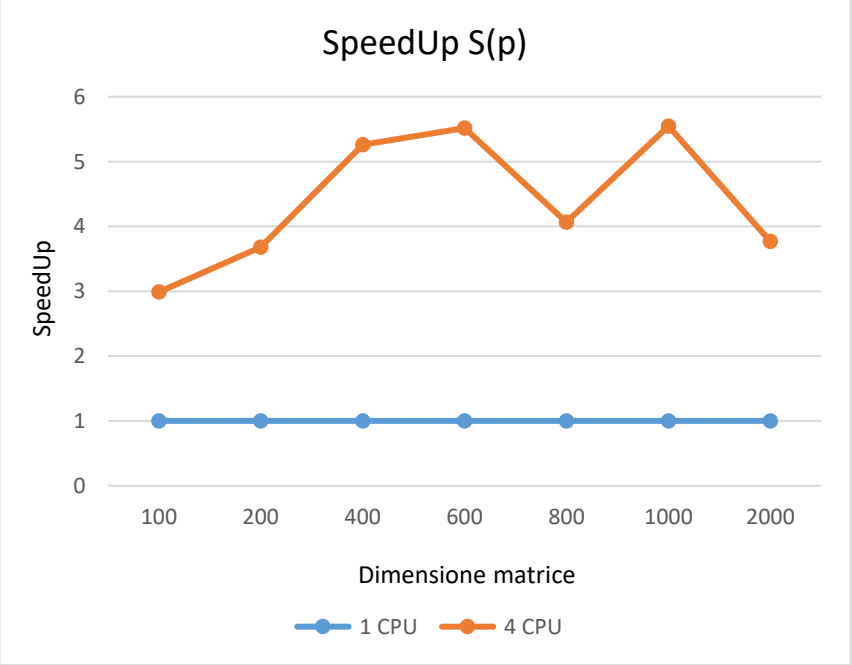
SpeedUp S(p)

A partire dai tempi misurati nella sezione precedente è stato calcolato lo speed-up al variare di M.

Tabella SpeedUp:

Numero CPU	Dimensione input						
	100	200	400	600	800	1.000	2.000
1	1,000000	1,000000	1,000000	1,000000	1,000000	1,000000	1,000000
4	2,986430	3,678559	5,257997	5,518595	4,063346	5,546555	3,768136



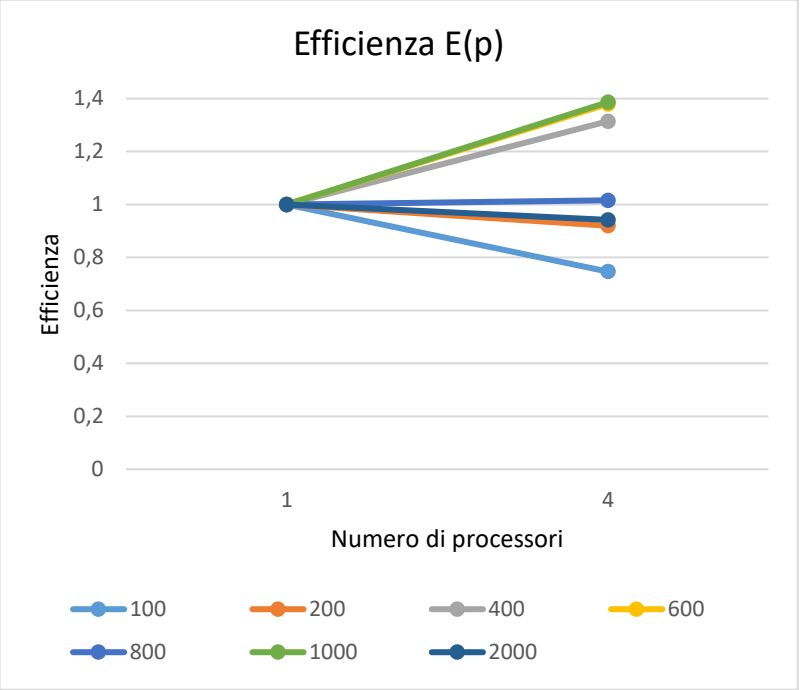


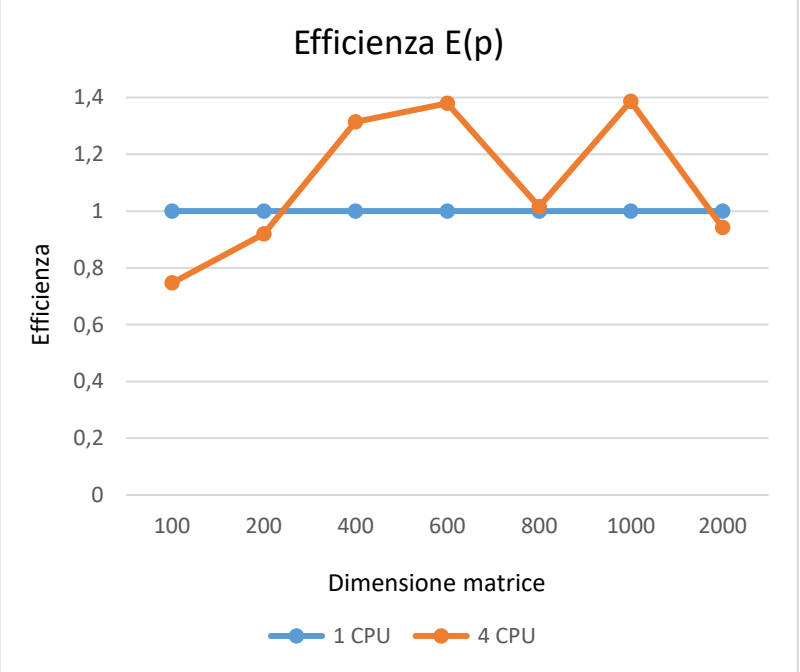
Efficienza E(p)

A partire dallo SpeedUp misurato nella sezione precedente è stata calcolata l'efficienza al variare di M.

Tabella efficienza:

Numero CPU	Dimensione input						
	100	200	400	600	800	1.000	2.000
1	1,000000	1,000000	1,000000	1,000000	1,000000	1,000000	1,000000
4	0,746607	0,919640	1,314499	1,379649	1,015837	1,386639	0,942034





Conclusioni

Dai grafici e dalle tabelle appena presentate si possono trarre le seguenti considerazioni:

- in tutte le dimensioni del problema analizzate, l'efficienza e lo SpeedUp risultano complessivamente maggiori nella configurazione in cui M = 1000;
- l'efficienza e lo speedUp iniziano a degradare nettamente in particolare quando la dimensione M = 2000.

Da come è possibile notare dai grafici, è evidente che sono presenti delle anomalie nei valori di efficienza e speedUp, poiché ricordiamo che il valore dello speedUp S(p) non può superare p (il numero di processi utilizzati, nel nostro caso 4), mentre l'efficienza non può superare il valore di 1. A seguito di alcuni esperimenti, testando il tempo di esecuzione sequenziale con un algoritmo analogo puramente iterativo, è emerso che i tempi di esecuzione con un singolo processore sono veritieri e coerenti con l'algoritmo parallelo illustrato al capitolo 4, per cui le anomalie sono dovute a un'eccessiva riduzione del tempo di esecuzione nel caso parallelo in cui si utilizzano 4 processori contemporaneamente.

A volte capita che lo speedup S(p) > p, questo è impossibile se l'algoritmo sequenziale e quello parallelo eseguono le stesse identiche istruzioni nelle stesse condizioni, in particolare sulla stessa macchina o su due macchine identiche.
In questi casi quindi si parla di "Speedup superlineare".

In sostanza, è il caso in cui una macchina con un numero n di processori ottiene prestazioni migliori più di n volte rispetto a una macchina con un singolo processore di velocità analoga; tale situazione può essere ottenuta da una macchina non solo n volte più potente, ma anche con più memoria cache e di sistema, appiattendo la gerarchia cache-memoria-disco, con una migliore gestione della memoria da parte di ogni processore e con altri accorgimenti. Tuttavia, la qualità delle prestazioni dipende soprattutto dal compito da svolgere e dalla possibilità di suddividerlo tra le unità di calcolo.

4. Codice sorgente

Il codice sorgente di seguito illustrato implementa l'algoritmo descritto nel capitolo 2, implementato attraverso l'ausilio del linguaggio di programmazione C.

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

/*******************************Strutture*******************************/

typedef struct {
    MPI_Comm grid_comm; /* communicator di griglia globale  */
    MPI_Comm row_comm;  /* communicator di righe della griglia */
    MPI_Comm col_comm;  /* communicator di colonne della griglia */
    int n_proc;         /* numero di processori */
    int grid_dim;       /* dimensione della griglia, = sqrt(n_proc) */
    int my_row;         /* posizione di riga di un processo in una griglia */
    int my_col;         /* posizione di colonna di un processo in una griglia */
    int my_rank;        /* il rank nella griglia */
} GridInfo;

/*************************Prototipi funzioni*************************/

/* Inizializzazione per la griglia dei processi */
void grid_init(GridInfo *grid);

/* Funzioni per operazioni su matrici */
void matrix_creation(double **pA, double **pB, double **pC, int size);
void matrix_init(double *A, double *B, int size);
void matrix_dot(double *A, double *B, double *C, int n);
void matrix_print(double *A, int n);
void matrix_free(double **pA, double **pB, double **pC);

/* Algoritmo BMR: prodotto scalare parallelo tra matrici */
void BMRAlgorithm(double *A, double *B, double *C, int size, GridInfo *grid);

/*********************************************************************/


int main(int argc, char *argv[]) {
    int i, j;
    double *pA, *pB, *pC;
    double *local_pA, *local_pB, *local_pC;
    int matrix_size;

    MPI_Init(&argc, &argv);

    GridInfo grid;
    grid_init(&grid);

    if(argc != 2) {
        if(grid.my_rank == 0) {
            printf("Errore: occorre passare in input il numero \"M\"
                "di righe e di colonne della matrice.\n");
            printf("Sintassi di invocazione PBS:
                "qsub nome_file.pbs -v M=[matrix_size]\n");
        }

        MPI_Finalize();
        exit(1);
    }

    matrix_size = atoi(argv[1]);
```

```c
    /* Controlli di errore */
    if(matrix_size < 1) {
        if(grid.my_rank == 0)
            printf("Errore: matrix_size (=%d) deve essere un "
                "numero positivo!\n\n", matrix_size, grid.n_proc);
        MPI_Finalize();
        exit(1);
    }
    if (matrix_size % grid.grid_dim != 0) {
        if(grid.my_rank == 0)
            printf("Errore: la dimensione della matrice (=%d) non e' divisibile\n"
                   "per la dimensione della griglia"
                   " dei processi (=%d)!\n\n", matrix_size, grid.n_proc);
        MPI_Finalize();
        exit(1);
    }

    if (grid.my_rank == 0) {
        matrix_creation(&pA, &pB, &pC, matrix_size);
        matrix_init(pA, pB, matrix_size);

        if(matrix_size <= 10) {
            printf("Matrice A:\n");
            matrix_print(pA, matrix_size);
            printf("Matrice B:\n");
            matrix_print(pB, matrix_size);
        }
    }

    int local_matrix_size = matrix_size / grid.grid_dim;
    matrix_creation(&local_pA, &local_pB, &local_pC, local_matrix_size);

    MPI_Datatype blocktype, type;
    int array_size[2] = {matrix_size, matrix_size};
    int subarray_sizes[2] = {local_matrix_size, local_matrix_size};
    int array_start[2] = {0, 0};

    MPI_Type_create_subarray(2, array_size, subarray_sizes, array_start,
                             MPI_ORDER_C, MPI_DOUBLE, &blocktype);
    MPI_Type_create_resized(blocktype, 0, local_matrix_size * sizeof(double), &type);
    MPI_Type_commit(&type);

    int displs[grid.n_proc];
    int sendcounts[grid.n_proc];

    if (grid.my_rank == 0) {
        for (i = 0; i < grid.n_proc; ++i) {
            sendcounts[i] = 1;
        }
        int disp = 0;
        for (i = 0; i < grid.grid_dim; ++i) {
            for (j = 0; j < grid.grid_dim; ++j) {
                displs[i * grid.grid_dim + j] = disp;
                disp += 1;
            }
            disp += (local_matrix_size - 1) * grid.grid_dim;
        }
    }

    MPI_Scatterv(pA, sendcounts, displs, type, local_pA,
                 local_matrix_size * local_matrix_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Scatterv(pB, sendcounts, displs, type, local_pB,
                 local_matrix_size * local_matrix_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    double start_time, end_time;
    MPI_Barrier(grid.grid_comm);

    if (grid.my_rank == 0) {
        start_time = MPI_Wtime();
    }
```

```
133     BMRAlgorithm(local_pA, local_pB, local_pC, local_matrix_size, &grid);
134
135     /* Raccolgo le sottomatrici da tutti i processi */
136     MPI_Gatherv(local_pC, local_matrix_size*local_matrix_size,
137                 MPI_DOUBLE, pC, sendcounts, displs, type, 0, MPI_COMM_WORLD);
138
139     if (grid.my_rank == 0) {
140         end_time = MPI_Wtime() - start_time;
141
142         if(matrix_size <= 10) {
143             printf("Matrice C (risultato prodotto scalare tra A e B):\n");
144             matrix_print(pC, matrix_size);
145         }
146
147         printf("---~---~---~---~---~---~---~---~---~---\n");
148         printf("Tempo algoritmo BMR: %.6lf secondi\nNumero di processi: "
149             "%d\nDimensione matrice: %d\n", end_time, grid.n_proc, matrix_size);
150         printf("---~---~---~---~---~---~---~---~---~---\n\n");
151
152         matrix_free(&pA, &pB, &pC);
153     }
154
155     matrix_free(&local_pA, &local_pB, &local_pC);
156     MPI_Finalize();
157     return 0;
158 }
159
160
161 /*********************************************************************/
162
163 /* Inizializza la griglia dei processi */
164 void grid_init(GridInfo *grid) {
165     int old_rank;
166     int dimensions[2];
167     int periods[2];
168     int coordinates[2];
169     int free_coords[2];
170
171     /* Raccolgo le informazioni generali prima di procedere */
172     MPI_Comm_size(MPI_COMM_WORLD, &(grid->n_proc));
173     MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);
174
175     grid->grid_dim = (int)sqrt(grid->n_proc);
176
177     /* Controllo di robustezza */
178     if (grid->grid_dim * grid->grid_dim != grid->n_proc) {
179         if (old_rank == 0)
180             printf("Errore: \'-np %d\' non e' un quadrato perfetto!\n", grid->n_proc);
181
182         MPI_Finalize();
183         exit(1);
184     }
185
186     /* Inizializzo le dimensioni */
187     dimensions[0] = dimensions[1] = grid->grid_dim;
188     periods[0] = periods[1] = 1; /* Caso griglia periodica */
189
190     MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions, periods, 1, &(grid->grid_comm));
191
192     /* Siccome abbiamo impostato il parametro reorder a true,
193        cio' potrebbe aver cambiato i ranks */
194     MPI_Comm_rank(grid->grid_comm, &(grid->my_rank));
195
196     /* Raccolgo le coordinate cartesiane per il processo corrente */
197     MPI_Cart_coords(grid->grid_comm, grid->my_rank, 2, coordinates);
198
199     /* Memorizzo i valori delle coordinate per il processo corrente */
200     grid->my_row = coordinates[0];
201     grid->my_col = coordinates[1];
```

```c
    /* Creo communicator di righe */
    free_coords[0] = 0;
    free_coords[1] = 1;
    MPI_Cart_sub(grid->grid_comm, free_coords, &(grid->row_comm));

    /* Creo communicator di colonne */
    free_coords[0] = 1;
    free_coords[1] = 0;
    MPI_Cart_sub(grid->grid_comm, free_coords, &(grid->col_comm));
}


/* Alloca uno spazio di memoria quadratico rispetto
   a "size", per ogni matrice data in input */
void matrix_creation(double **pA, double **pB, double **pC, int size) {
    *pA = (double *)malloc(size * size * sizeof(double));
    *pB = (double *)malloc(size * size * sizeof(double));
    *pC = (double *)calloc(size * size, sizeof(double));
}


/* Inizializza la matrice con valori predefiniti */
void matrix_init(double *A, double *B, int size) {
    int i;
    for (i = 0; i < size * size; ++i) {
        *(A + i) = (double)i + 1.;
        *(B + i) = (double)i + 2.;
    }
}


/* Effettua il prodotto scalare tra le matrici A e B, e lo memorizza in C */
void matrix_dot(double *A, double *B, double *C, int size) {
    int i, j, k;
    for (i = 0; i < size; ++i) {
        for (j = 0; j < size; ++j) {
            for (k = 0; k < size; ++k) {
                C[i * size + j] += A[i * size + k] * B[k * size + j];
            }
        }
    }
}


/* Stampa la matrice di valori reali data in input */
void matrix_print(double *A, int size) {
    int i;

    if(size > 5)
        printf("---~---~---~---~---~---~---~---~---~---~---~---\n");
    else
        printf("---~---~---~---~---~---~---\n");


    for (i = 0; i < size * size; ++i) {
        printf("%.2lf ", *(A + i));
        if ((i + 1) % size == 0){
            printf("\n");
        }
    }

    if(size > 5)
        printf("---~---~---~---~---~---~---~---~---~---~---~---\n\n");
    else
        printf("---~---~---~---~---~---\n\n");
}
```

```c
/* Libera la memoria per tutte le matrici date in input */
void matrix_free(double **pA, double **pB, double **pC) {
    free(*pA);
    free(*pB);
    free(*pC);
}


/* ------------------------------ Algoritmo BMR ------------------------------ */
void BMRAlgorithm(double *A, double *B, double *C, int size, GridInfo *grid) {
    MPI_Status status;
    int root;

    /* Alloco lo spazio di memoria per il blocco di A da trasmettere (broadcast) */
    double *buff_A = (double*)calloc(size * size, sizeof(double));

    // Calcolo gli indirizzi per lo spostamento circolare di B
    int src = (grid->my_row + 1) % grid->grid_dim;
    int dst = (grid->my_row - 1 + grid->grid_dim) % grid->grid_dim;

    /**
     * Ad ogni iterazione:
     *   1. trova i blocchi che si trovano sulla diagonale della griglia dei processi
     *   2. condividi quel blocco con la riga della griglia a cui appartiene quel processo
     *   3. moltiplica l'A aggiornato (o buff_A) con B e salvalo in C
     *   4. sposta i blocchi di B verso la riga precedente della stessa colonna
     */
    int stage;
    for (stage = 0; stage < grid->grid_dim; ++stage) {
        root = (grid->my_row + stage) % grid->grid_dim;
        if (root == grid->my_col) {
            MPI_Bcast(A, size * size, MPI_DOUBLE, root, grid->row_comm);
            matrix_dot(A, B, C, size);
        } else {
            MPI_Bcast(buff_A, size * size, MPI_DOUBLE, root, grid->row_comm);
            matrix_dot(buff_A, B, C, size);
        }
        MPI_Sendrecv_replace
           (B, size * size, MPI_DOUBLE, dst, 0, src, 0, grid->col_comm, &status);
    }
}

/* --------------------------------------------------------------------------- */
```