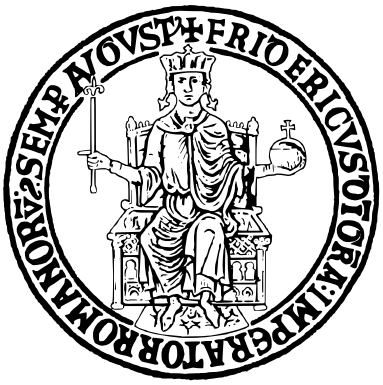


Università degli studi di Napoli Federico II

Scuola Politecnica e delle Scienze di base

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

Corso di laurea magistrale in Informatica

Parallel and Distributed Computing

Documentazione relativa al calcolo del prodotto matrice-matrice su architettura MIMD a memoria distribuita

Studente
Francesco Scarfato

Matricola
N86003769

2023-2024

Indice

Capitolo 1

Introduzione

Nel campo dell'informatica, viene definito *parallelismo* la decomposizioni di un problema in sotto problemi risolti contemporaneamente da più processori. L'introduzione del calcolo parallelo ha permesso di superare molti limiti tecnologici e di ottenere prestazioni di alto livello. L'architettura MIMD (Multiple Instruction Multiple Data) è un paradigma che consente di eseguire diverse operazioni su diversi processori simultaneamente. Nel caso di studio, ogni unità di elaborazione possiede una memoria indipendente, quindi la comunicazione tra processori è una parte fondamentale per la risoluzione di problemi.

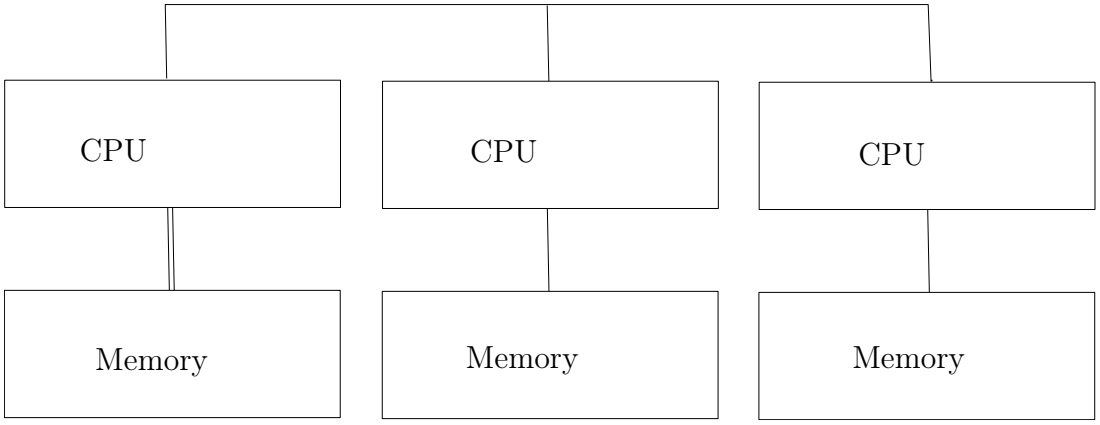


Figura 1.1: Architettura MIMD a memoria distribuita

Questo documento si propone di descrivere la problematica del prodotto tra due matrici utilizzando una strategia particolare e analizzarne, tramite lo studio di dati, le performance.

Capitolo 2

Descrizione del problema

Il calcolo del prodotto matrice-matrice è un'operazione che può risultare molto onerosa per dimensioni grandi.

Sia $A \in M_{d,d}$ matrice con d righe, d colonne e $B \in M_{d,d}$ matrice con d righe e d colonne, allora è possibile eseguire il prodotto matrice-matrice e il risultato è una matrice $R \in M_{d,d}$

$$A \in M_{d,d} \times B \in M_{d,d} \rightarrow R \in M_{d,d}$$

Nello specifico il prodotto viene calcolato moltiplicando ogni riga della matrice A per ogni colonna della matrice B e poi sommando.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,d} \\ a_{2,1} & a_{2,2} & \dots & a_{2,d} \\ \dots & \dots & \dots & \dots \\ a_{d,1} & a_{d,2} & \dots & a_{d,d} \end{bmatrix} \cdot \begin{bmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,d} \\ b_{2,1} & b_{2,2} & \dots & b_{2,d} \\ \dots & \dots & \dots & \dots \\ b_{d,1} & b_{d,2} & \dots & b_{d,d} \end{bmatrix} = \begin{bmatrix} a_{1,1} \cdot b_{1,1} + a_{1,2} \cdot b_{2,1} + \dots & a_{1,1} \cdot b_{1,2} + a_{1,2} \cdot b_{2,2} + \dots \\ a_{2,1} \cdot b_{1,1} + a_{2,2} \cdot b_{2,1} + \dots & a_{2,1} \cdot b_{1,2} + a_{2,2} \cdot b_{2,2} + \dots \\ \dots & \dots \\ a_{d,1} \cdot b_{1,1} + a_{d,2} \cdot b_{2,1} + \dots & a_{d,1} \cdot b_{1,2} + a_{d,2} \cdot b_{2,2} + \dots \end{bmatrix}$$

In un'architettura parallela, è possibile risolvere questo problema in modo molto veloce però sfruttare questo tipo di ambiente comporta la gestione di nuovi problemi:

- la distribuzione dei dati
- la comunicazione tra i processori

La distribuzione dei dati è il principale problema poiché la comunicazione e la conseguente sincronizzazione vengono gestiti tramite l'ambiente MPI con la sua libreria (capitolo 4).

Capitolo 3

Descrizione dell'algoritmo

L'idea è dividere il problema in sotto problemi risolvibili nello stesso momento da diversi processori. La risoluzione del problema del prodotto tra matrici può essere diviso in diversi step: dividere l'input, distribuirlo ai vari processori, calcolare localmente (in ogni processore) una sotto matrice con i numeri ricevuti e, eventualmente, ricostruire la matrice risultante.

Nota bene. La strategia usata sfrutta il posizionamento virtuale dei processori a forma di griglia.

3.1 Divisione e distribuzione dei dati

I problemi di divisione e distribuzione dei dati vengono risolti cercando di dividere entrambe le matrici in blocchi omogenei e distribuire ogni blocco delle due matrici a uno specifico processore. Nel caso di studio la matrice è quadrata e la dimensione deve essere multiplo della radice del numero di processori in modo da estrarre sotto blocchi di dimensione uguale.

Come mostrato nella figura sotto, i processori formano una griglia quadrata e, in base alla sua posizione, gli vengono assegnati un sotto blocco di A e un sotto blocco di B.

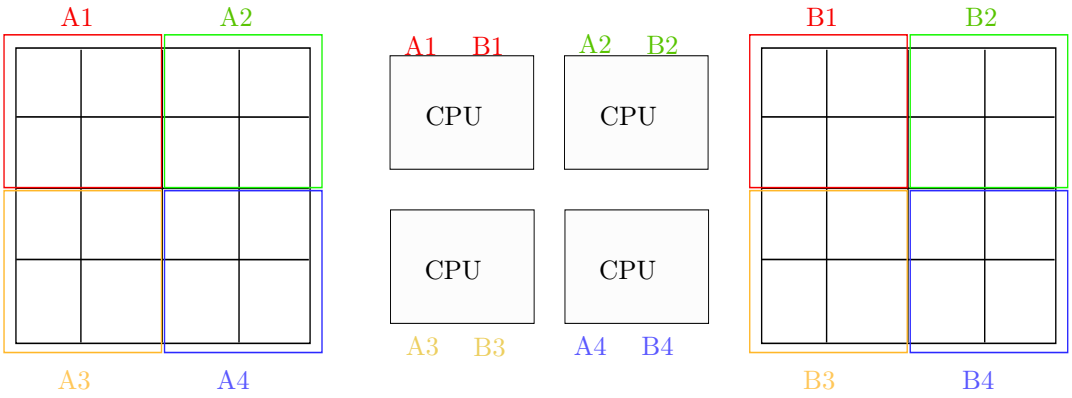


Figura 3.1: Divisione delle sotto matrici

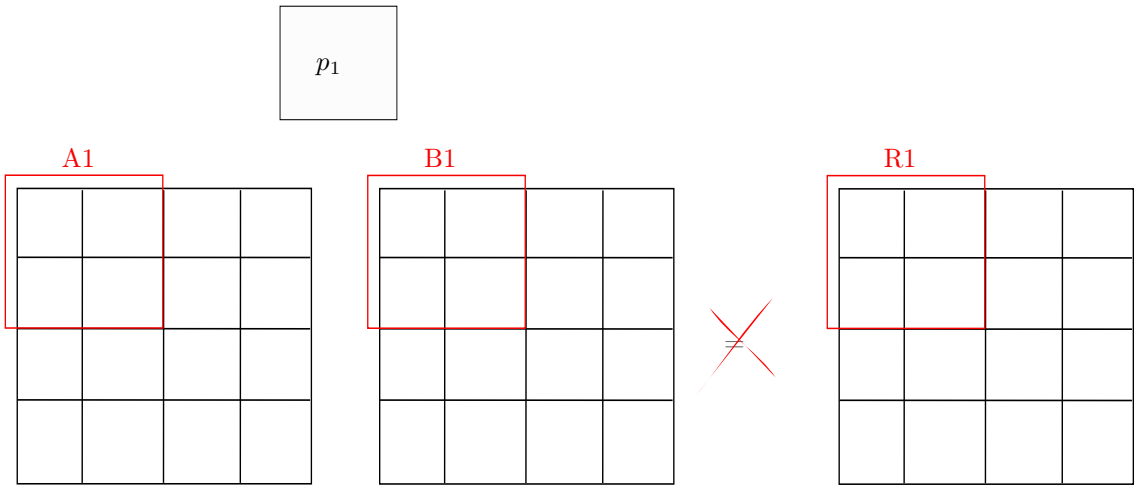
3.2 Broadcast Multiply Rolling

Una volta distribuiti i dati e creata la griglia di processori, parte la vera e propria strategia di risoluzione. Questa strategia è detta BMR (Broadcast Multiply Rolling) ed è divisa in 3 step: broadcast, multiply e rolling.

L'obbiettivo è che ogni processore calcoli un sotto blocco della matrice risultante. Dopo la distribuzione dei dati però questo è ancora impossibile perché a ogni processore mancano dei dati per calcolare il proprio sotto blocco.

Esempio. Siano A, B matrici 4×4, sia $p = 4$ numero di processori e sia già avvenuta la distribuzione dei dati, allora p_1 contiene i blocchi A_1 e B_1 (come visto precedentemente) quindi

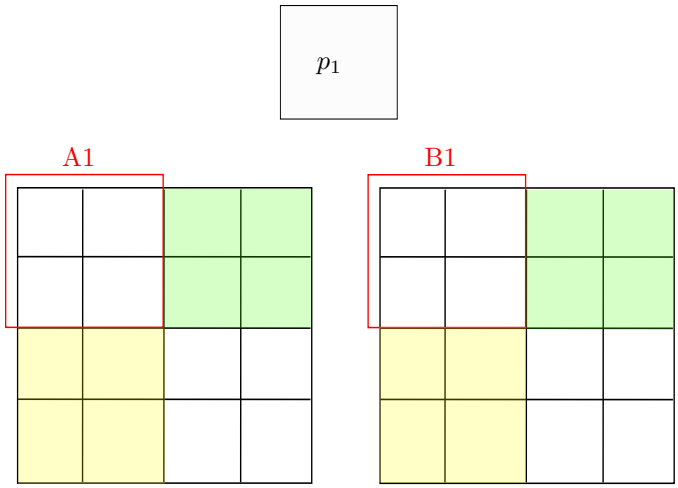
$$D_{p_1} = \{A_{0,0}, A_{0,1}, A_{1,0}, A_{1,1}, B_{0,0}, B_{0,1}, B_{1,0}, B_{1,1}\}$$



Con questi dati il processore non può ancora calcolare $R1$ poiché $R1$ ha 4 celle $R_{0,0}, R_{0,1}, R_{1,0}, R_{1,1}$ in cui:

- $R_{0,0} = A_{0,0} \cdot B_{0,0} + A_{0,1} \cdot B_{1,0} + A_{0,2} \cdot B_{2,0} + A_{0,3} \cdot B_{3,0}$ quindi mancano $A_{0,2}, A_{0,3}, B_{2,0}, B_{3,0}$
- $R_{0,1} = A_{0,0} \cdot B_{0,1} + A_{0,1} \cdot B_{1,1} + A_{0,2} \cdot B_{2,1} + A_{0,3} \cdot B_{3,1}$ quindi mancano $A_{0,2}, A_{0,3}, B_{2,1}, B_{3,1}$
- $R_{1,0} = A_{1,0} \cdot B_{0,0} + A_{1,1} \cdot B_{1,0} + A_{1,2} \cdot B_{2,0} + A_{1,3} \cdot B_{3,0}$ quindi mancano $A_{1,2}, A_{1,3}, B_{2,0}, B_{3,0}$
- $R_{1,1} = A_{1,0} \cdot B_{0,1} + A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} + A_{1,3} \cdot B_{3,1}$ quindi mancano $A_{1,2}, A_{1,3}, B_{2,1}, B_{3,1}$

Riassumendo al processore p_1 mancano questi dati che sono in possesso di p_2 e p_3



Applicando la strategia, la prima operazione che viene effettuata è un broadcast sulla riga (della griglia di processori) di appartenenza eseguito dai processori sulla diagonale principale. Con questo broadcast, p_1 e p_4 comunicano a tutti i processori sulla loro riga il loro sotto blocco di A. Nel caso dell'esempio p_1 trasmette $A1$ a p_2 e p_4 trasmette $A4$ a p_3.

A questo punto viene effettuata la moltiplicazione ma il sotto blocco ottenuto è una soluzione parziale. La situazione al momento è:

- a p_1 mancano $A2$ e $B2$
- a p_2 manca solo $B4$ in quanto $A1$ è stato trasmesso da p_1
- a p_3 manca solo $B1$ in quanto $A4$ è stato trasmesso da p_4
- a p_4 mancano $A3$ e $B3$

Inoltre non si possono più effettuare moltiplicazioni in quanto nessun processore ha i blocchi di A e B necessari.

Per risolvere questo problema si effettua un altro broadcast sulle righe dai processori situati sulla diagonale successiva alla principale quindi da p_2 e p_3 in modo che p_1 e p_4 abbiano i sotto blocchi di A necessari. Dopo questo broadcast, per risolvere completamente il problema servono i blocchi B quindi si effettua un rolling ovvero ogni processore trasmette il suo blocco B al processore che lo precede nella colonna. Dopo quest'operazione è possibile calcolare il prodotto tra i sotto blocchi di A e B trasmessi.

Nel caso di esempio, alla fine di questo step, il problema è stato completamente risolto. Nel caso ci siano p processori, il numero di BMR è proprio p.

Capitolo 4

Descrizione delle routine

Prima di descrivere le routine, vengono presentate le librerie utilizzate:

- `mpi.h`, libreria standard per l'implementazione di software in ambienti paralleli che segue il modello *Message Passing*. Contiene tutte le funzioni, le variabili e le costanti per l'inizializzazione e la gestione della comunicazione tra processori
- `stdio.h`, libreria standard di C per la gestione dell'input/output
- `stdlib.h`, libreria standard di C che comprende alcune utility per allocare memoria (`malloc()`), per la conversione (`atoi()`) e per la generazione di numeri pseudocasuali
- `string.h`, libreria standard di C per la gestione e la manipolazione delle stringhe (usata per funziona di copia e inizializzazione delle matrici)
- `math.h`, libreria standard che contiene utility per la gestione delle operazioni matematiche (`log2()`)
- `getopt.h`, libreria che contiene variabili e funzioni per gestire i parametri passati tramite linea di comando
- `ctype.h`, libreria che dichiara funzioni per la gestione dei singoli caratteri (usata per controllare se un carattere è effettivamente un numero con `isdigit()`)
- `time.h`, libreria standard di C per gestire tipi di dato temporali (usata per la generazione di numeri casuali dandogli come seed l'orario attuale con `time()`)

Listing 4.1: Librerie incluse nel codice

```
1  #include <mpi.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/time.h>
5  #include <time.h>
6  #include <getopt.h>
7  #include <ctype.h>
8  #include <math.h>
9  #include <string.h>
```

Viene definita una struttura dati che memorizza le informazioni essenziali per ogni processo come i diversi ID nei vari communicator.

```
1  struct coords{
2      int world;
3      int gridRank;
4      int gridCoords[2];
5      int row;
6      int column;
7  };
```

in cui `world` indica l'id del processo nel communicator `MPI_COMM_WORLD`, `gridRank` indica un singolo id associato al processore nel communicator a forma di griglia, `gridCoords[2]` indica le coordinate associate a ciascun processore nella griglia, `row` indica l'id associato al processore nel communicator di riga e `column` indica l'id associato al processore nel communicator di colonna.

Il primo step è la configurazione del problema tramite la lettura dei parametri da linea di comando. Il programma, per essere chiamato, necessita di alcune opzioni. Per chiarezza, la sinossi del comando è

```
mpiexec -np <n-proc> matrix-matrix -d <intero> [-s]
```

in cui `-d` indica la dimensione di un lato della matrice e `-s` è una flag che può essere attivata per vedere l'output completo del programma.

La configurazione inizia con il processore principale che chiama la funzione `interpretCommandLine()` che legge le opzioni elencate e le memorizza con i dovuti controlli. Segue una versione molto semplificata di questa funzione.

```
void interpretCommandLine(int argc, char* argv[], int* dims, int* nProc){
    int opt, tmp;
    //Per controllare se queste opzioni necessarie sono state passate a riga di comando
    int checkD = -1;

    //Lettura delle opzioni e degli argomenti passati a riga di comando
    while ((opt = getopt(argc, argv, ":d:s")) != -1) {
        switch (opt) {
            //Opzione per la dimensione delle matrici
            case 'd':
                checkD = 0;
                tmp = atoi(optarg);
                *dims = tmp;
                break;
            case 's':
                SHOW_OUTPUT = 0;
                break;
            case '?':
                fprintf(stderr, "One option is unknown.\n");
                exit(-1);
        }

    }
    //Controlla se tutte le opzioni sono state date a riga di comando
    if (checkD != 0) {
        fprintf(stderr, "Options missed. Insert:\n\t-d <matrix dimension>\n");
        exit(-1);
    }
}
```

Dopo il parsing dei parametri, vengono effettuati i controlli necessari in quanto il numero di processori deve essere un quadrato perfetto per costruire la griglia quadrata e la dimensione della matrice deve essere multiplo del radice del numero di processori.

Successivamente viene allocata la memoria necessaria per le varie strutture dati con la funzione `mallocMatrix()` e vengono inizializzate le matrici con numeri casuali tramite la funzione `generateRandomNumbers()`

Listing 4.2: Funzione per allocare matrice

```
void mallocMatrix(int ***matrix, int r, int c) {
    int i;

    int *cell = (int *)malloc(r*c*sizeof(int));
    if (!cell){
        exit(-1);
    };

    (*matrix) = (int **)malloc(r*sizeof(int*));
    if (!(*matrix)) {
        free(cell);
        exit(-1);
    }

    for (i=0; i<r; i++){
        (*matrix)[i] = &(cell[i*c]);
    }
}
```

Listing 4.3: Funzione per inizializzare matrice

```
void generateRandomNumbers(int **matrix, int r, int c){
    int i, j;

    for (i=0; i<r; i++){
        for (j=0; j<c; j++){
            matrix[i][j] = rand()%10;
        }
    }
}
```

A questo punto viene effettuata la distribuzione dei dati tramite la creazione di un nuovo `MPI_Datatype` che rappresenta il sotto blocco da distribuire. Successivamente vengono calcolati gli offset per memorizzare il punto di invio di ogni sotto blocco e, infine, viene usata la funzione `MPI_Scatterv()` per distribuirli.

```
MPI_Datatype distributeData(int **a, int **b, int **subBlockA, int **subBlockB,
                            struct coords coord, int dim, int dimSub, int gridSize,
                            int nProc, int *countDataToSend, int *offset) {
    int dimsMatrix[2] = {dim, dim};
    int dimsSubmatrix[2] = {dimSub, dimSub};
    int start[2] = {0,0};
    int i, j;
    int* ptrToMatrixA = NULL;
    int* ptrToMatrixB = NULL;
    MPI_Datatype type, block;

    /* Creazione del tipo block per distribuire blocchi di matrice*/
    MPI_Type_create_subarray(2, dimsMatrix, dimsSubmatrix, start, MPI_ORDER_C, MPI_INT, &type);
    MPI_Type_create_resized(type, 0, dimSub*sizeof(int), &block);
    MPI_Type_commit(&block);


    /* Definizione degli offset */
    if (coord.world == 0) {
```

```
20          /* ptr da cui partire per mandare le sotto matrici*/
21          ptrToMatrixA = &(a[0][0]);
22          ptrToMatrixB = &(b[0][0]);
23
24          /* Setting del numero di blocchi da mandare per ogni processore */
25          for (i=0; i<nProc; i++) countDataToSend[i] = 1;
26
27          /* Calcolo degli offset in base alla dim del sotto blocco */
28          int disp = 0;
29          for (i=0; i<gridSize; i++) {
30              for (j=0; j<gridSize; j++) {
31                  offset[i*gridSize+j] = disp;
32                  disp += 1;
33              }
34              disp += (dimSub-1)*gridSize;
35          }
36      }
37
38      MPI_Scatterv(ptrToMatrixA, countDataToSend,
39                   offset, block, &(subBlockA[0][0]), dimSub*dimSub,
40                   MPI_INT, 0, MPI_COMM_WORLD);
41      MPI_Scatterv(ptrToMatrixB, countDataToSend,
42                   offset, block, &(subBlockB[0][0]), dimSub*dimSub,
43                   MPI_INT, 0, MPI_COMM_WORLD);
44
45      for(i=0; i<nProc && SHOW_OUTPUT == 0; i++) {
46          if(coord.world == i) {
47              printf("Local block of A on rank %d is:\n", coord.world);
48              printMatrix(subBlockA, dimSub, dimSub);
49              printf("Local block of B on rank %d is:\n", coord.world);
50              printMatrix(subBlockB, dimSub, dimSub);
51              MPI_Barrier(MPI_COMM_WORLD);
52          }
53      }
54
55      return block;
56  }
```

Dopo la distribuzione dei dati, viene costruita la griglia e i communicator di riga e colonna tramite la funzione `createGrid()`. Subito dopo vengono assegnate le coordinate ad ogni processore tramite la funzione `MPI_Cart_coords()`

Nota bene. La griglia deve essere periodica per applicare la BMR altrimenti gli indici andrebbero "out of bound".

```
1   void createGrid(MPI_Comm *grid, MPI_Comm *rowC, MPI_Comm *columnC, int gridSize) {
2       int periods[] = {1,1}, dimsGrid[] = {gridSize, gridSize}, reorder = 0;
3
4       /* Creazione griglia*/
5       MPI_Cart_create(MPI_COMM_WORLD, 2, dimsGrid, periods, reorder, grid);
6
7       int remain1[] = {0,1};
8       MPI_Cart_sub(*grid, remain1, rowC);
9
10      int remain2[] = {1,0};
11      MPI_Cart_sub(*grid, remain2, columnC);
12  }
```

A questo punto viene avviato il timer e viene eseguita la BMR tramite l'apposita funzione. Prima di partire effettivamente con la risoluzione viene eseguita una configurazione in cui vengono salvati gli id e le coordinate dei processori nelle apposite strutture dati.

```
void BMR(int **subBlockA, int **subBlockB, int **resultSubBlock, struct coords coord,
         MPI_Comm rowC, MPI_Comm columnC, MPI_Comm grid, int dimSub, int gridSize) {
    int **temp;
    /* Processori sulla diagonale principale */
    int senderCoords[2] = {coord.gridCoords[0], coord.gridCoords[0]};

    /* Identifica l'id dei processi che devono fare broadcast */
    int senderGridRank, sender, i;

    /* Processori comunicanti sul communicator di colonna */
    int previousProcessorCoords[2] = {coord.gridCoords[0]-1, coord.gridCoords[1]};
    int nextProcessorCoords[2] = {coord.gridCoords[0]+1, coord.gridCoords[1]};
    int previousProcessorRank, nextProcessorRank;

    MPI_Request request;
    MPI_Status status;

    /* Viene salvato per ogni processore un rank associato alla sua coordinata */
    MPI_Cart_rank(grid, coord.gridCoords, &coord.gridRank);
    MPI_Comm_rank(columnC, &coord.column);

    /* Calcolo dell'id per i processori sulla diagonale principale */
    MPI_Cart_rank(grid, senderCoords, &senderGridRank);

    /* Id nel communicator di riga dei processi che mandano */
    MPI_Cart_rank(rowC, &senderGridRank, &sender);

    /* Calcolo dell'id dei processori che comunicano sul communicator di colonna */
    MPI_Cart_rank(columnC, previousProcessorCoords, &previousProcessorRank);
    MPI_Cart_rank(columnC, nextProcessorCoords, &nextProcessorRank);

    /* Allocazione e inizializzazione matrice di appoggio */
    mallocMatrix(&temp, dimSub, dimSub);
    memset(*temp, 0, dimSub*dimSub*sizeof(int));
```

Dopo la configurazione inizia il vero e proprio algoritmo di risoluzione con il primo step di base ovvero il broadcast effettuato dai processori sulla diagonale principale e la conseguente moltiplicazione

```
    /* Processori sulla diagonale principale */
    if(coord.gridCoords[0] == coord.gridCoords[1]) {
        /* Matrice di appoggio per non sovrascrivere il sotto blocco di A locale */
        memcpy(*temp, *subBlockA, dimSub*dimSub*sizeof(int));
    }

    /* BROADCAST */
    /* Processori sulla diagonale principale fanno un broadcast sul communicator di riga */
    MPI_Bcast(&(temp[0][0]), dimSub*dimSub, MPI_INT, sender, rowC);

    /* MULTIPLY */
    computeMatrixProduct(*temp, *subBlockB, *resultSubBlock, dimSub);
```

A seguire l'algoritmo iterativo per la risoluzione completa

```
    for (i=1; i<gridSize; i++) {
        /* Diagonale successiva */
        senderCoords[1] += 1;
        MPI_Cart_rank(grid, senderCoords, &senderGridRank);
        MPI_Cart_rank(rowC, &senderGridRank, &sender);

        if(senderGridRank == coord.gridRank) {
            /* Matrice di appoggio per non sovrascrivere il sotto blocco di A locale */
            memcpy(*temp, *subBlockA, dimSub*dimSub*sizeof(int));
        }

        /* BROADCAST */
        /* Processori sulla diagonale successiva fanno un broadcast sul communicator di riga */
        MPI_Bcast(&(temp[0][0]), dimSub*dimSub, MPI_INT, sender, rowC);

        /* ROLLING */
        MPI_Isend(*subBlockB, dimSub*dimSub, MPI_INT, previousProcessorRank, 0,
                  columnC, &request); //Send non bloccante
        MPI_Recv(&(subBlockB[0][0]), dimSub*dimSub, MPI_INT, nextProcessorRank,
                  MPI_ANY_TAG, columnC, &status);

        /* MULTIPLY */
        computeMatrixProduct(*temp, *subBlockB, *resultSubBlock, dimSub);
    }
    freeMatrix(&temp);
}
```

Nota bene. Viene usata la funzione `MPI_Isend()` e non `MPI_Send()` poiché la prima è non bloccante e, in questo caso, è necessario che gli invii siano non bloccanti altrimenti ci sarebbe un'attesa infinita. Questo avviene poiché tutti i processori sul communicator di colonna inviano dati e, se venisse usata la send bloccante, tutti aspetterebbero una risposta.

L'ultima routine importante da illustrare è `computeMatrixProduct()` ovvero la funzione che effettua la vera e propria moltiplicazione fra due matrici.

```
void computeMatrixProduct(int *A, int *B, int *C, int dim) {
    int i, j, z;

    for(i = 0; i < dim; i++){
       for(j = 0; j < dim; j++){
           for(z = 0; z < dim; z++){
               C[i*dim+j] += A[i*dim+z] * B[dim*z+j];
           }
       }
   }

}
```

Dopo la BMR è possibile ricostruire la matrice risultante tramite `MPI_Gatherv()`.

È possibile menzionare anche alcune routine accessorie come `printMatrix()` e `freeMatrix()` che rispettivamente stampano una matrice e deallocano una matrice.

```
void printMatrix(int **matrix, int r, int c){
    int i, j;

    for (i=0; i<r; i++){
        for (j=0; j<c; j++){
            printf("|%d|", matrix[i][j]);
        }
        printf("\n");
    }
}
```

```
void freeMatrix(int ***array) {
    free(&((*array)[0][0]));
    free(*array);
}
```

Capitolo 5

Descrizione dei test

I test sono stati effettuati sullo *SCoPE Datacenter*, infrastruttura con risorse di calcolo e storage accessibili mediante paradigmi di calcolo distribuito. Il datacenter ha messo a disposizione 8 nodi DELL blade M600, ognuno dotato di 2 processori Intel Xeon quadcore. Le varie esecuzioni sono state lanciate in batch tramite *Portable Batch System* (PBS). Tra i test rientrano anche gli esempi d'uso, nel quale è stato utilizzato un PBS più semplice e dei log più verbosi per rendere chiaro il più possibile il funzionamento del programma. Il PBS tipo utilizzato per gli esempi d'uso è questo:

Listing 5.1: Script PBS

```
1  #!/bin/bash
2
3  #PBS -q studenti
4  #PBS -l nodes=8:ppn=8
5  #PBS -o $PROJECT.out
6  #PBS -e $PROJECT.err
7
8  sort -u $PBS_NODEFILE > hostlist
9
10 cat $PBS_NODEFILE
11 PBS_O_WORKDIR=$PBS_O_HOME/$PROJECT
12
13 #Controlli per la configurazione
14
15 mpicc -o $PBS_O_WORKDIR/$PROJECT $PBS_O_WORKDIR/$PROJECT.c
16
17 mpiexec -machinefile hostlist -np $p $PBS_O_WORKDIR/$PROJECT -d $nELEM -s
```

e viene chiamato da questo comando

```
1  qsub -v PROJECT="<name>",p=<processors>,nELEM=<intero> script.pbs
```

n cui `PROJECT` è una variabile che prende il nome del progetto, `p` è una variabile che memorizza il numero di processori da usare e `nELEM` è la dimensione di un lato della matrice.

5.1 Esempi di uso

Sia n dimensione di un lato della matrice e p numero di processori, di seguito sono forniti gli esempi d'uso per:

1. sia $n = 8$ e $p = 5$
2. sia $n = 7$ e $p = 4$
3. sia $n = 4$ e $p = 4$

4. sia $n = 9$ e $p = 9$

Il primo caso d'uso restituisce un errore in quanto il numero di processori non è un quadrato perfetto quindi non si riesce a costruire una griglia.

```
Impossible to create pxp grid
: Success
Impossible to create pxp grid
: Success
--------------------------------------------------------------------------
MPI_ABORT was invoked on rank 0 in communicator MPI_COMM_WORLD
with errorcode 1.

NOTE: invoking MPI_ABORT causes Open MPI to kill all MPI processes.
You may or may not see output from other processes, depending on
exactly when Open MPI kills them.
--------------------------------------------------------------------------
--------------------------------------------------------------------------
mpiexec has exited due to process rank 0 with PID 28369 on
node wn273.scope.unina.it exiting without calling "finalize". This may
have caused other processes in the application to be
terminated by signals sent by mpiexec (as reported here).
--------------------------------------------------------------------------
[wn280.scope.unina.it:17874] 1 more process has sent help message help-mpi-api.txt / mpi-abort
[wn280.scope.unina.it:17874] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages
```

Anche nel secondo caso restituisce un errore in quanto il numero di elementi di un lato della matrice non è multiplo della radice del numero di processori.

```
Matrix dimension isn't multiple of processors number
: Success
--------------------------------------------------------------------------
MPI_ABORT was invoked on rank 2 in communicator MPI_COMM_WORLD
with errorcode 1.

NOTE: invoking MPI_ABORT causes Open MPI to kill all MPI processes.
You may or may not see output from other processes, depending on
exactly when Open MPI kills them.
--------------------------------------------------------------------------
Matrix dimension isn't multiple of processors number
: Success
--------------------------------------------------------------------------
mpiexec has exited due to process rank 2 with PID 13601 on
node wn275.scope.unina.it exiting without calling "finalize". This may
have caused other processes in the application to be
terminated by signals sent by mpiexec (as reported here).
--------------------------------------------------------------------------
[wn280.scope.unina.it:23747] 1 more process has sent help message help-mpi-api.txt / mpi-abort
[wn280.scope.unina.it:23747] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages
```

Nel terzo caso viene restituito il risultato corretto

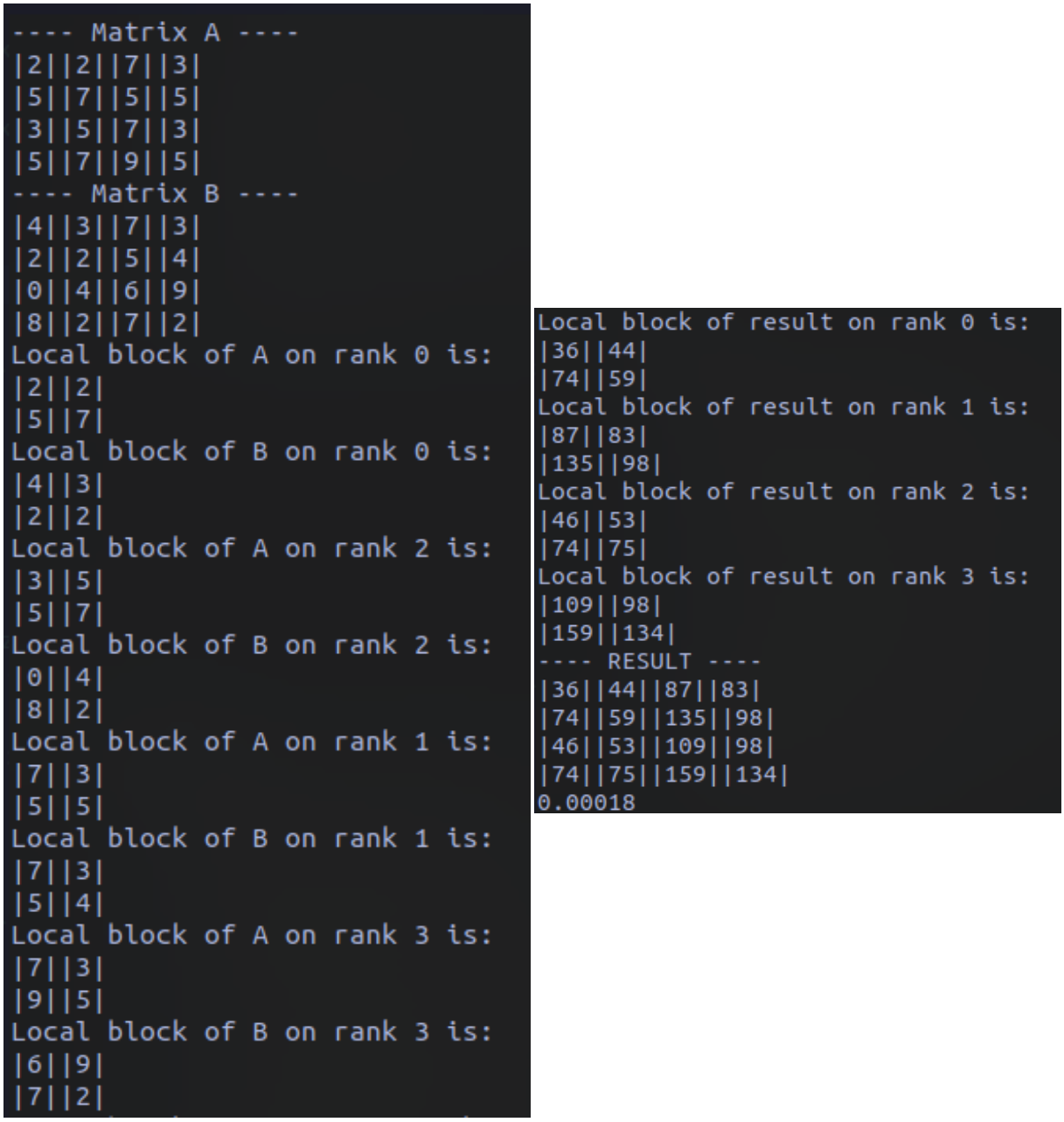
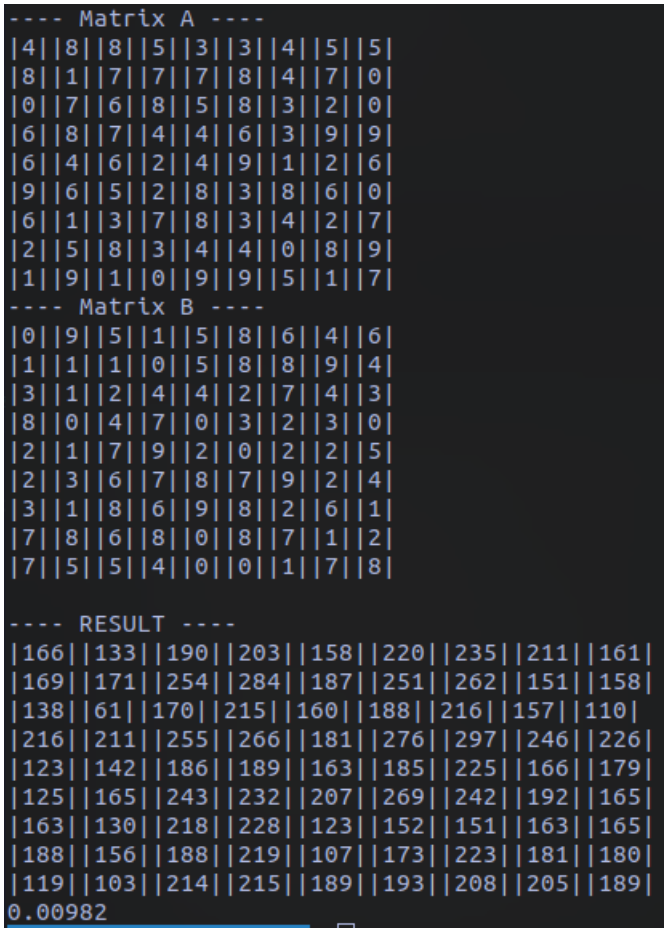


Figura 5.1: Terzo esempio d'uso

Nel quarto caso viene restituisce il risultato corretto ma bisogna sottolineare che il nono processore non è reale ma virtuale poiché, come descritto precedentemente, il datacenter utilizzato mette a disposizione solamente 8 nodi.


---- Matrix A ----
4		8		8		5		3		3		4		5		5
8		1		7		7		7		8		4		7		0
0		7		6		8		5		8		3		2		0
6		8		7		4		4		6		3		9		9
6		4		6		2		4		9		1		2		6
9		6		5		2		8		3		8		6		0
6		1		3		7		8		3		4		2		7
2		5		8		3		4		4		0		8		9
1		9		1		0		9		9		5		1		7
---- Matrix B ----
0		9		5		1		5		8		6		4		6
1		1		1		0		5		8		8		9		4
3		1		2		4		4		2		7		4		3
8		0		4		7		0		3		2		3		0
2		1		7		9		2		0		2		2		5
2		3		6		7		8		7		9		2		4
3		1		8		6		9		8		2		6		1
7		8		6		8		0		8		7		1		2
7		5		5		4		0		0		1		7		8
---- RESULT ----
166		133		190		203		158		220		235		211		161
169		171		254		284		187		251		262		151		158
138		61		170		215		160		188		216		157		110
216		211		255		266		181		276		297		246		226
123		142		186		189		163		185		225		166		179
125		165		243		232		207		269		242		192		165
163		130		218		228		123		152		151		163		165
188		156		188		219		107		173		223		181		180
119		103		214		215		189		193		208		205		189
0.00982

Capitolo 6

Analisi delle performance

Per eseguire l'analisi, sono stati presi dieci tempi d'esecuzione dell'algoritmo di risoluzione e ne è stata fatta la media, in modo da avere un risultato con un margine d'errore più basso. Questo non significa che i dati presi sono corretti al 100%. I test sono stati eseguiti per studiare il tempo impiegato dall'algoritmo al variare dei processori con in aggiunta il calcolo e l'analisi dello speed-up e dell'efficienza. Per rendere più robusto lo studio, l'algoritmo è stato eseguito con diverse dimensioni. Per rendere il tutto automatizzato, è stato implementato uno script PBS che, oltre a definire le direttive PBS come quello precedentemente presentato, calcola automaticamente tutti i dati necessari e li fornisce in output tramite una tabella.

```
#Direttive PBS e compilazione uguali

#Tempo impiegato dall'algoritmo con 1 processore
t1=0
efficiency=0
printf '%-20s%-20s%-15s%-15s\n' 'Processors number' 'Time (average)'
                              'Speed-up' 'Efficiency'

#Incrementa il numero processori
for p in 1 4
do
    sum=0 #Somma di 10 tempi
    for (( i=0; i<10; i++ ))
    do
        # Prende il valore del tempo totale di un esecuzione
        tmp=$(/usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np $p
              $PBS_O_WORKDIR/$PROJECT -d $nELEM)
        sum=$(echo "$sum $tmp" | awk '{printf("%.5f", $1+$2)}')
    done

    #echo Somma: $sum
    average=$(echo "$sum" | awk '{printf("%.6f", $1/10)}')
    #echo Media: $average

    if [ $p -eq 1 ]; then
        #T(1) settato quando ottengo il tempo di 1 processore
        t1=$average
        speedup=1
        efficiency=1
    else
        tp=$average
        #Calcolo speed-up T(1)/T(P)
        speedup=$(echo "$t1 $tp" | awk '{printf("%.6f", $1/$2)}')
        #Calcolo efficienza S(P)/P
```

```
35          efficiency=$(echo "$speedup $p" | awk '{printf("%.6f", $1/$2)}')
36      fi
37      printf "%-20d%-20.5f%-15.5f%-15.5f\n" "$p" "$average" "$speedup" "$efficiency"
38  done
```

Lo speed-up misura la riduzione del tempo di esecuzione di un algoritmo parallelo rispetto all'esecuzione dello stesso algoritmo su un solo processore (sequenziale) ed è calcolabile come

$$S(p) = \frac{T(1)}{T(p)}$$

Mentre l'efficienza misura quanto l'algoritmo sfrutta il parallelismo del calcolatore e si calcola con

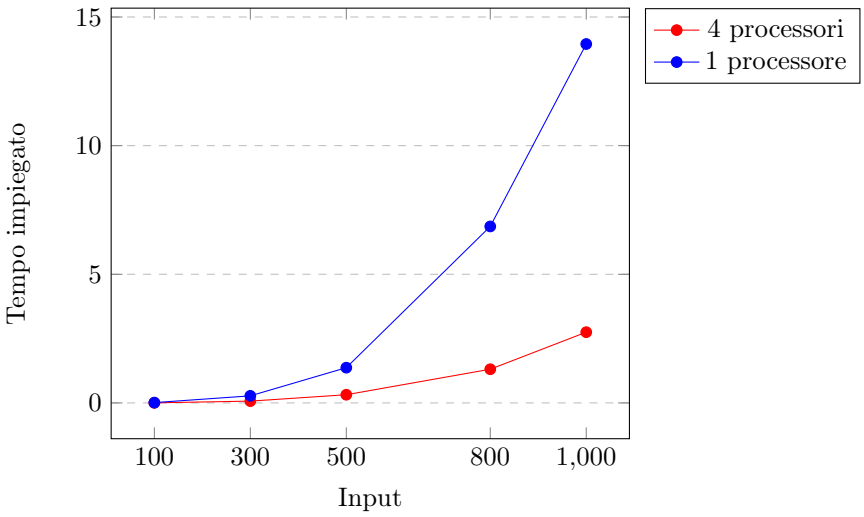
$$E(p) = \frac{S(p)}{p}$$

Questa è la tabella da cui sono stati generati i grafici. I test sono stati effettuati con matrici $100 \times 100, 300 \times 300, 500 \times 500, 800 \times 800, 1000 \times 1000$. Inoltre l'algoritmo è stato eseguito per ogni input con un solo processore e con 4 poiché l'unica griglia quadrata di processori realmente testabile è quella 2×2 (il massimo numero di processori disponibili è 8).

Input	p	T(p)	S(p)	E(p)
100x100	1	0.01169	1.00000	1.00000
100x100	4	0.00304	3.84891	0.96223
300x300	1	0.27465	1.00000	1.00000
300x300	4	0.06948	3.95299	0.98825
500x500	1	1.36929	1.00000	1.00000
500x500	4	0.31782	4.30841	1.07710
800x800	1	6.85955	1.00000	1.00000
800x800	4	1.30927	5.23920	1.30980
1000x1000	1	13.94949	1.00000	1.00000
1000x1000	4	2.74890	5.07457	1.26864

Nel grafico sottostante, è possibile notare come l'algoritmo parallelo sia molto più veloce rispetto all'algoritmo sequenziale. Quindi l'aspettativa è ampiamente rispettata.

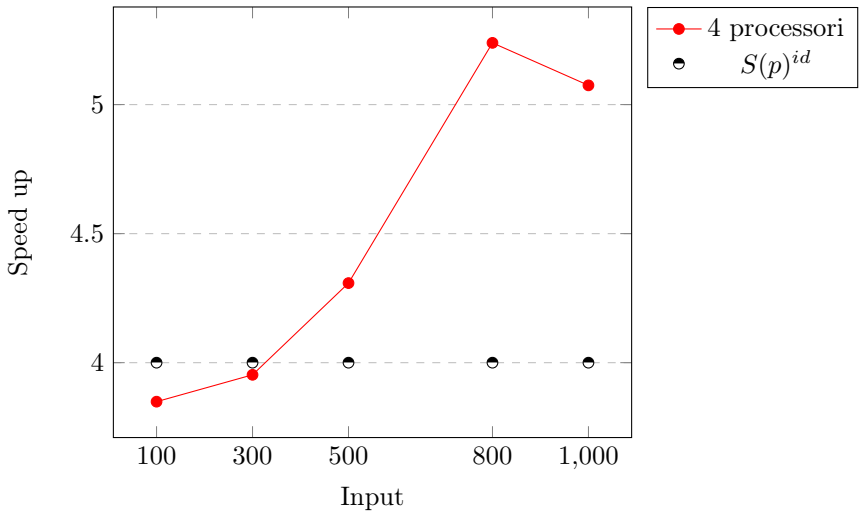
Tempo impiegato al crescere dell'input



Invece il grafico sottostante riguardante lo speed-up (conseguentemente anche l'efficienza) non rispetta le aspettative in quanto per input maggiori di 300, l'algoritmo con 4 processori supera lo speed-up ideale. Con matrici 800x800 si ha uno speed-up di 5.23920 mentre lo speed-up ideale dovrebbe essere 4. Probabilmente, oltre all'errore nelle misurazioni, bisogna riconsiderare l'esecuzione dell'algoritmo sequenziale. L'algoritmo è stato sviluppato per essere parallelo quindi ci sono molte operazione nella

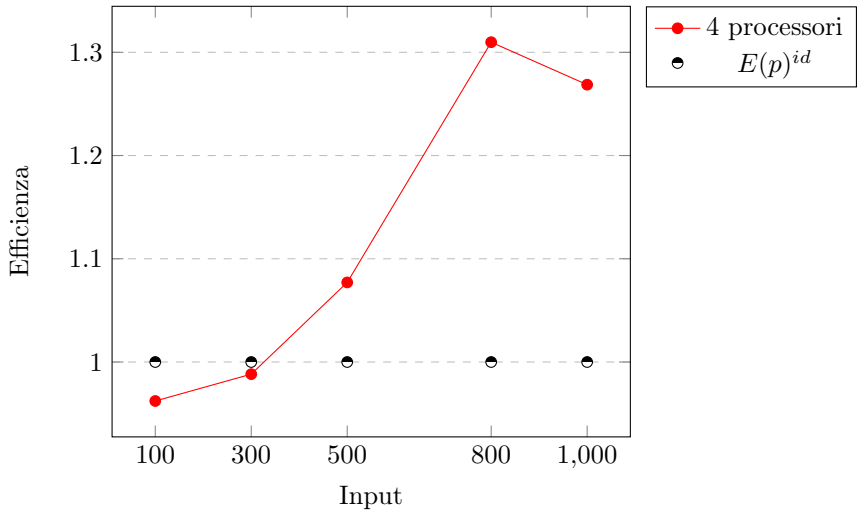
BMR che non interessano all'algoritmo sequenziale. Per avere una prova più concreta, si dovrebbe sviluppare un'algoritmo sequenziale e calcolare i tempi.

Speed-up al crescere dell'input



Naturalmente, l'anomalia riscontrata nel grafico precedente, si rispecchia anche nel grafico sottostante in quanto anche l'efficienza con input maggiori di 500 è maggiore a quella ideale.

Efficienza al crescere dell'input



Provando con un algoritmo specifico sequenziale come questo sottostante con delle matrici 1000x1000

```
1  int main(int argc, char** argv) {
2      int **a, **b;
3      int dim=1000;
4      int **r, i, j, k;
5      struct timeval time1;
6      double t0, t1;
7
8      mallocMatrix(&a, dim, dim);
9      mallocMatrix(&b, dim, dim);
10     mallocMatrix(&r, dim, dim);
11
12     srand(time(0));
13     generateRandomNumbers(a, dim, dim);
14     generateRandomNumbers(b, dim, dim);
```

```
15
16      //Calcolo tempo iniziale
17      gettimeofday(&time1, NULL);
18      t0 = time1.tv_sec + (time1.tv_usec/1000000.0);
19
20      for (i = 0; i < dim; i++) {
21          for (j = 0; j < dim; j++) {
22              int sum = 0;
23              for (k = 0; k < dim; k++) {
24                  sum += a[i][k] * b[k][j];
25              }
26              r[i][j] = sum;
27          }
28      }
29
30      //Calcolo tempo finale
31      gettimeofday(&time1, NULL);
32      t1 = time1.tv_sec + (time1.tv_usec/1000000.0);
33      printf("%f\n", t1-t0);
34  }
```

Si ottiene un tempo molto minore di 13.94949, infatti l'algoritmo sequenziale risolve il problema in 9.776775.

$$S(4) = \frac{T(1)}{T(4)} = 3,556613$$

che è un valore assolutamente possibile in quanto non è superiore allo speed-up ideale (4). Anche per l'efficienza si ottiene un valore al di sotto dell'efficienza ideale ma molto vicino infatti

$$E(4) = \frac{S(4)}{4} = 0,88915$$

Questo sottolinea significativamente che l'algoritmo sequenziale da confrontare dovrebbe essere ottimale e costruito ad-hoc per essere sequenziale in quanto, con algoritmi adattati, i dati potrebbero essere falsati e l'analisi sbagliata.