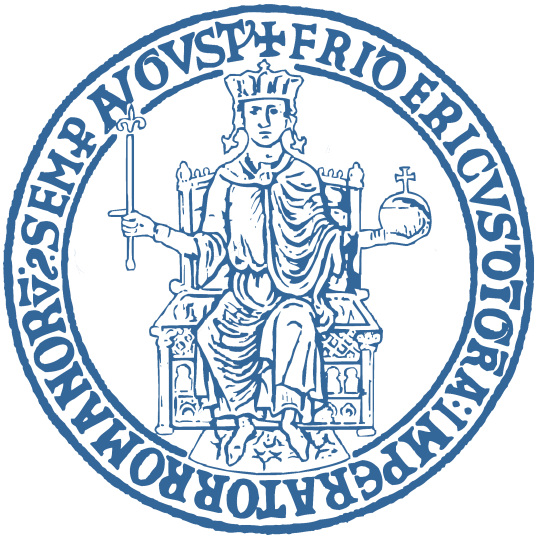


Università degli Studi di Napoli Federico II

Parallel and Distributed Computing

Parallel Matrix Vector Product con OpenMP

2022/2023

Professori	*Studente*
Anna Mele	N86002592
Giuliano Laccetti	Vincenzo Tramo

Indice

Descrizione del problema

1 Introduzione

Il seguente elaborato ha l'obiettivo di documentare l'implementazione dell'algoritmo per il calcolo del prodott matrice vettore:

$$\mathbf{A} * \mathbf{x} = \mathbf{b}$$

dove $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{x} \in \mathbb{R}^n$ e $\mathbf{b} \in \mathbb{R}^m$ in ambiente di calcolo parallelo su architettura MIMD (Multiple Instruction Multiple Data) a memoria distribuita utilizzando OpenMP API (The OpenMP API specification for parallel programming https://www.openmp.org/). L'obiettivo del seguente documento è anche quello di mostrare alcuni risultati relativi al comportamento dell'algoritmo in termini di tempo di esecuzione e speed up rispetto alla strategia utilizzata per implementarlo.

1.1 Scopo del software

Lo scopo principale del software è quello di eseguire in maniera parallela il prodotto matrice vettore utilizzando la libreria OpenMP. L'utilizzatore del software può scegliere la dimensione della matrice e il numero di threads da utilizzare. Il software alla fine fornirà il risultato corretto e il tempo impiegato (in secondi) per calcolarlo.

2 Descrizione dell'algoritmo

L'algoritmo è un semplice prodotto matrice vettore parallelo. La strategia utilizzata è la suddivisione in righe della matrice **A**. La principale funzione che esegue il calcolo del prodotto è la seguente:

```
double *matxvet(
    int rows,
    int columns,
    const double *restrict x,
    double **restrict A
) {
    double *b = (double*) calloc(rows, sizeof(double));
    int i, j;

    #pragma omp parallel for default(none) shared(rows, columns, A, x, b) private(i, j)
    for (i = 0; i < rows; i++) {
        for (j = 0; j < columns; j++) {
            b[i] += A[i][j] * x[j];
        }
    }

    return b;
}
```

La funzione `double *matxvet(rows, columns, x, A)` riceve in input:

- `rows`: il numero di righe della matrice di tipo `int`
- `columns`: il numero di colonne della matrice di tipo `int`
- `x`: il vettore **x** (di dimensione `columns`) di tipo `double*`
- `A`: la matrice **A** (di dimensione `rows` × `columns`) di tipo `double**`

e restituisce in output il vettore risultato `b` (di dimensione `rows`) di tipo `double*`.

La keyword `restrict` può essere aggiunta ad un puntatore ed è stata applicata ai due puntatori `x` e `A`. Tale keyword suggerisce al compilatore che per il ciclo di vita del puntatore, nessun altro puntatore accederà all'oggetto a cui punta. Questo permette al compilatore di fare delle ottimizzazioni (riducendo il numero di istruzioni linguaggio macchina) portando a performance migliori. E' cura del programmatore assicurare che un altro puntatore non accedi mai all'oggetto puntato dal puntatore dichiarato `restrict` e il compilatore non aiuterà in alcun modo il rivelamento di questa violazione. Se un puntatore indipendente accede alla memoria puntata dal puntatore dichiarato `restrict`, questo risulterà in un undefined behavior.

2.1 Come eseguire il programma

Il programma può essere eseguito in due modi. Il primo è quello di compilare ed eseguire il programma `parallel_vector_matrix_product.c` con il comando:

```
$ gcc -fopenmp -lgomp -o parallel_vector_matrix_product ./parallel_vector_matrix_product.c
$ ./parallel_vector_matrix_product <rows> <columns> <input_file_matrix> \
    <input_file_vector_x> <enable_printing_mode_boolean_value>
```

nella cartella in cui si trova il file `parallel_vector_matrix_product.c`. Al programma vanno forniti i seguenti input:

- `rows`: il numero di righe della matrice **A** (deve essere maggiore di 0)
- `columns`: il numero di colonne della matrice **A** (deve essere maggiore di 0)
- `input_file_matrix`: il percorso di un file di testo contenente i numeri per la matrice **A** (devono essere esattamente `rows*columns` in totale)
- `input_file_vector`: il percorso di un file di testo contenente i numeri per il vettore **x** (devono essere esattamente `columns` in totale)

- `enable_printing_mode_boolean_value`: il programma alla fine stampa l'intera matrice **A** insieme ai vettori **x** e **b**. Quando le dimensioni non sono banali, questo potrebbe essere un fastidio. E' possibile disattivare la stampa fornendo come input il valore booleano `false` (o `true` se si vuole stampare). Questa opzione non disattiva la stampa del tempo impiegato per calcolare il risultato.

Per avviare invece il programma su un cluster tramite il file pbs `parallel_vector_matrix_product.pbs` è necessario eseguire lo script `start_parallel_vector_matrix_product_pbs.sh`. Tale script, prima di avviare il file pbs, chiederà le seguenti informazioni:

- il numero di righe della matrice **A**
- il numero di colonne della matrice **A**
- il numero di threads da utilizzare
- se abilitare la stampa della matrice **A** e dei vettori **x** e **b** alla fine della computazione (true o false)

Se il numero di elementi della matrice è minore o uguale di 20, allora il programma chiederà di inserire manualmente gli elementi della matrice **A** e del vettore **x**, altrimenti li genera automaticamente in maniera random. Alla fine esegue il pbs `parallel_vector_matrix_product.pbs` effettivamente sul cluster. Se si sceglie di avviare il programma in questo modo, lo script eseguito (oltre a generare i file che normalmente vengono generati dopo l'esecuzione del pbs) genera un file `results.txt` contenente dei records i cui campi sono:

- `PARALLEL_TIME`: tempo di esecuzione in secondi dell'algoritmo parallelo
- `SINGLETH_TIME`: tempo di esecuzione in secondi del miglior algoritmo sequenziale (single thread)
- `SPEED_UP_P/ST`: lo speed up
- `TOTAL_THREADS`: il numero totale di threads
- `ROWS`: il numero totale di righe
- `COLUMNS`: il numero totale di colonne
- `TOT_NUMBERS`: il numero totale di numeri della matrice **A**
- `TOT_ITERATIONS`: il numero totale di iterazioni fatte per calcolare la media del tempo di esecuzione (sia dell'algoritmo parallelo sia dell'algoritmo sequenziale)

```
Enter the number of rows: 50
Enter the number of columns: 50

2500 random numbers have just been generated

50 random numbers have just been generated
Enter the number of threads: 8
Print matrix A, vector x and result b on stdout? (Y/N): N

EXECUTING PARALLEL MATRIX VECTOR PRODUCT

Il pbs parallel_vector_matrix_product.pbs e' stato accodato. I risultati saranno \
stampati in parallel_vector_matrix_product.out e parallel_vector_matrix_product.err.
```

2.2 La direttiva `pragma omp parallel for`

L'unica regione parallela del programma è quella presente nella funzione `double *matxvet(rows, columns, x, A)`:

```
...
#pragma omp parallel for default(none) shared(rows, columns, A, x, b) private(i, j)
for (i = 0; i < rows; i++) {
    for (j = 0; j < columns; j++) {
        b[i] += A[i][j] * x[j];
    }
}
...
```

In questa porzione di codice vengono distribuite le righe della matrice tra i threads. In particolare:

- `parallel`: forma un team di thread ed avvia un'esecuzione parallela
- `for`: specifica che le iterazioni del ciclo for (più esterno) devono essere distribuite tra i thread del team
- `default`: in genere tutti i dati sono condivisi dai threads. Questo costrutto permette di stabilire se nessuna o tutte le variabili devono essere condivise con `share` oppure `none`
- `shared`: la lista di variabili specificate in questo costrutto sono condivise tra i threads
- `private`: la lista di variabili specificate in questo costrutto sono private per ogni thread

Alla fine della direttiva è un'implicita barriera di sincronizzazione al di fuori della quale esisterà poi un solo thread, cioè quello master. Dunque, quando la funzione `double *matxvet(..)` ritorna, il risultato è stato già calcolato e ritornato correttamente dalla stessa funzione.

2.3 Il codice sorgente

```c
#include <omp.h>
#include <stdio.h>
#include <stdbool.h>
#include <ctype.h>
#include <stdlib.h>
#include <sys/time.h>
#include <string.h>

double *matxvet(int rows, int columns, const double *restrict x, double **restrict A);

double **read_matrix_from_file(int rows, int columns, char *file_name);
double *read_vector_from_file(char *file_name);

int get_number_of_rows_command_line(char **argv);
int get_number_of_columns_command_line(char **argv);
bool get_printing_mode(char **argv);

void check_arguments(int argc, char **argv);
bool is_number(const char *s);
void print_correct_usage(char **argv);

void print_vector(double *vector, int size);
void print_matrix(double **A, int rows, int columns);

void free_matrix(double **A, int rows);

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

    const bool is_active_printing_mode = get_printing_mode(argv);
    const int rows = get_number_of_rows_command_line(argv);
    const int columns = get_number_of_columns_command_line(argv);

    char *matrix_file_name = argv[3];
    char *vector_file_name = argv[4];

    double **A = read_matrix_from_file(rows, columns, matrix_file_name);
    double *x = read_vector_from_file(vector_file_name);

    struct timeval time;
    double t0, t1;

    gettimeofday(&time, NULL);
    t0 = time.tv_sec + (time.tv_usec / 100000.0);
```

```
    /* Compute solution */
    double *b = matxvet(rows, columns, x, A);

    gettimeofday(&time, NULL);
    t1 = time.tv_sec + (time.tv_usec / 100000.0);


    /* Print result */
    if (is_active_printing_mode) {
        printf("Matrix:\n\n");
        print_matrix(A, rows, columns);
        printf("\n\nVector x: ");
        print_vector(x, columns);
        printf("\nResult: ");
        print_vector(b, rows);
    }

    printf("[TOTAL TIME PARALLEL] %e seconds.\n", t1 - t0);

    free(x);
    free_matrix(A, rows);

    return EXIT_SUCCESS;
}

double *matxvet(int rows, int columns, const double *restrict x, double **restrict A) {
    double *b = (double*) calloc(rows, sizeof(double));
    int i, j;

    #pragma omp parallel for default(none) shared(rows, columns, A, x, b) private(i, j)
    for (i = 0; i < rows; i++) {
        for (j = 0; j < columns; j++) {
            b[i] += A[i][j] * x[j];
        }
    }

    return b;
}

double **read_matrix_from_file(const int rows, const int columns, char *file_name) {
    FILE *file = fopen(file_name, "r");
    double **A = (double**) malloc(rows * sizeof(double*));
    double n = 0;
    for (int i = 0; i < rows; i++) {
        double *row = (double*) malloc(columns * sizeof(double));
        for (int j = 0; j < columns; j++) {
            if (fscanf(file, "%lf", &n) <= 0) return NULL;
            row[j] = n;
        }
        A[i] = row;
    }
    fclose(file);
    return A;
}

double* read_vector_from_file(char *file_name) {
    FILE *file = fopen(file_name, "r");
    double n = 0;
    int i = 0;
    int dim = 1;
```

```
    double *vector = malloc(sizeof(double));
    while (fscanf(file, "%lf", &n) > 0) {
        vector[i++] = n;
        vector = realloc(vector, sizeof(double) * ++dim);
    }
    fclose(file);
    return vector;
}

void check_arguments(const int argc, char **argv) {
    if (argc != 6 || !is_number(argv[1]) || !is_number(argv[2])) {
        print_correct_usage(argv);
        exit(EXIT_FAILURE);
    }
}

int get_number_of_rows_command_line(char **argv) {
    const static int NO_ROWS_INDEX_COMM_LINE = 1;
    if (!is_number(argv[NO_ROWS_INDEX_COMM_LINE])) {
        print_correct_usage(argv);
        exit(EXIT_FAILURE);
    }
    return atoi(argv[NO_ROWS_INDEX_COMM_LINE]);
}

int get_number_of_columns_command_line(char **argv) {
    const static int NO_COLUMNS_INDEX_COMM_LINE = 2;
    if (!is_number(argv[NO_COLUMNS_INDEX_COMM_LINE])) {
        print_correct_usage(argv);
        exit(EXIT_FAILURE);
    }
    return atoi(argv[NO_COLUMNS_INDEX_COMM_LINE]);
}

bool get_printing_mode(char **argv) {
    const static int PRINTING_MODE_INDEX_COMM_LINE = 5;
    char *printing_mode = argv[PRINTING_MODE_INDEX_COMM_LINE];
    if (strcmp(printing_mode, "true") != 0 && strcmp(printing_mode, "false") != 0) {
        print_correct_usage(argv);
        exit(EXIT_FAILURE);
    }
    return strcmp(printing_mode, "true") == 0;
}

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

void print_correct_usage(char **argv) {
    printf("Correct usage: %s <rows> <columns> <file_matrix> <file_vector> <printing_mode_boolean_value
}

void print_matrix(double **A, int rows, int columns) {
    for (int i = 0; i < rows; i++) {
```

```
        for (int j = 0; j < columns; j++) {
            printf("%f%s", A[i][j], j + 1 == columns ? "\n" : " ");
        }
    }
}

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

void free_matrix(double **A, int rows) {
    for (int i = 0; i < rows; i++) {
        free(A[i]);
    }
    free(A);
}
```

2.4 Il file pbs parallel_vector_matrix_product.pbs

```
#!/bin/bash

#PBS -q studenti
#PBS -l nodes=1:ppn=8
#PBS -N parallel_vector_matrix_product
#PBS -o parallel_vector_matrix_product.out
#PBS -e parallel_vector_matrix_product.err

echo 'Job is running on node(s): '
cat $PBS_NODEFILE

PBS_O_WORKDIR=$PBS_O_HOME/parallel-vector-matrix-product-openmp
echo ------------------------------------------------------
echo PBS: qsub is running on $PBS_O_HOST
echo PBS: originating queue is $PBS_O_QUEUE
echo PBS: executing queue is $PBS_QUEUE
echo PBS: working directory is $PBS_O_WORKDIR
echo PBS: execution mode is $PBS_ENVIRONMENT
echo PBS: job identifier is $PBS_JOBID
echo PBS: job name is $PBS_JOBNAME
echo PBS: node file is $PBS_NODEFILE
echo PBS: current home directory is $PBS_O_HOME
echo PBS: PATH = $PBS_O_PATH
echo ------------------------------------------------------

# Questo PBS deve essere avviato tramite lo script "start_parallel_vector_matrix_product_pbs.sh"

FILENAME_VECTOR_VALUES="$PBS_O_WORKDIR/vector_values.txt"
FILENAME_MATRIX_VALUES="$PBS_O_WORKDIR/matrix_values.txt"
FILENAME_RESULTS="$PBS_O_WORKDIR/results.txt"
TOT_ITERATIONS=5

result_parallel_vector_matrix_product=""
result_single_thread_vector_matrix_product=""
time_parallel_vector_matrix_product=0
time_single_thread_vector_matrix_product=0

function execute_parallel_vector_matrix_product() {
    export OMP_NUM_THREADS=$total_threads
```

```
    export PSC_OMP_AFFINITY=TRUE
    gcc -fopenmp -lgomp -o $PBS_O_WORKDIR/parallel_vector_matrix_product \
        $PBS_O_WORKDIR/parallel_vector_matrix_product.c

    for ((i = 0; i < $TOT_ITERATIONS; i++)); do
        result_parallel_vector_matrix_product=$($PBS_O_WORKDIR/parallel_vector_matrix_product \
            $rows $columns $FILENAME_MATRIX_VALUES $FILENAME_VECTOR_VALUES $printing_mode)
        this_time=$(echo "$result_parallel_vector_matrix_product" | grep "TIME" | awk '{print $4}')
        time_parallel_vector_matrix_product=$(awk -v a="$this_time" \
            -v b="$time_parallel_vector_matrix_product" 'BEGIN{print (a + b)}')
    done

    time_parallel_vector_matrix_product=$(awk -v a="$time_parallel_vector_matrix_product" \
        -v b="$TOT_ITERATIONS" 'BEGIN{print (a / b)}')
    echo "$result_parallel_vector_matrix_product"
    rm $PBS_O_WORKDIR/parallel_vector_matrix_product
}

function execute_single_thread_vector_matrix_product() {
    gcc -o $PBS_O_WORKDIR/single_thread_vector_matrix_product \
        $PBS_O_WORKDIR/single_thread_vector_matrix_product.c

    for ((i = 0; i < $TOT_ITERATIONS; i++)); do
        result_single_thread_vector_matrix_product=$($PBS_O_WORKDIR/single_thread_vector_matrix_product
            $rows $columns $FILENAME_MATRIX_VALUES $FILENAME_VECTOR_VALUES)
        this_time=$(echo "$result_single_thread_vector_matrix_product" | grep "TIME" | awk '{print $5}'
        time_single_thread_vector_matrix_product=$(awk -v a="$this_time" \
            -v b="$time_single_thread_vector_matrix_product" 'BEGIN{print (a + b)}')
    done

    time_single_thread_vector_matrix_product=$(awk -v a="$time_single_thread_vector_matrix_product"
        -v b="$TOT_ITERATIONS" 'BEGIN{print (a / b)}')
    echo "$result_single_thread_vector_matrix_product"
    rm $PBS_O_WORKDIR/single_thread_vector_matrix_product
}

execute_single_thread_vector_matrix_product
execute_parallel_vector_matrix_product

rm $FILENAME_VECTOR_VALUES
rm $FILENAME_MATRIX_VALUES

if [[ ! -e "$FILENAME_RESULTS" ]]; then
    printf "PARALLEL_TIME\tSINGLETH_TIME\tSPEED_UP_P/ST\tTOTAL_THREADS\tROWS\tCOLUMNS\tTOT_NUMBERS\
        \tTOT_ITERATIONS" > $FILENAME_RESULTS
fi

speed_up=$(awk -v a="$time_single_thread_vector_matrix_product"
    -v b="$time_parallel_vector_matrix_product" 'BEGIN{print (a / b)}')
printf "\n$time_parallel_vector_matrix_product\t$time_single_thread_vector_matrix_product\t$speed_up \
    \t$OMP_NUM_THREADS\t$rows\t$columns\t$(($rows * $columns))\t$TOT_ITERATIONS" >> $FILENAME_RESULTS
```

2.5 Esempi di output

Adesso verranno mostrati esempi di output eseguendo il programma sul cluster SCoPE con lo script `start_parallel_vector_` (che a sua volta eseguira il pbs).

2.5.1 Output inserendo manualmente i dati

Avviando lo script `start_parallel_vector_matrix_product.sh` scegliendo un numero di righe e colonne tale per cui il numero totale degli elementi della matrice sia minore di 20 permetterà all'utente di inserire manual-

mente i numeri all'interno della matrice **A** e del vettore **b**. Utilizziamo 8 threads e vogliamo che l'output della matrice, del vettore **x** e del risultato **b** vengano scritti sul file `parallel_vector_matrix_product.out`.

```
Enter the number of rows: 3
Enter the number of columns: 3
Position A(0, 0): 5
Position A(0, 1): 4
Position A(0, 2): 3
Position A(1, 0): 2
Position A(1, 1): 1
Position A(1, 2): 2
Position A(2, 0): 3
Position A(2, 1): 4
Position A(2, 2): 5
Position x(0): 1
Position x(1): 1
Position x(2): 1
Enter the number of threads: 8
Print matrix A, vector x and result b on stdout? (Y/N): Y

EXECUTING PARALLEL MATRIX VECTOR PRODUCT
4006786.torque02.scope.unina.it

Il pbs parallel_vector_matrix_product.pbs e' stato accodato. \
I risultati saranno stampati in parallel_vector_matrix_product.out e \
parallel_vector_matrix_product.err.
```

Il risultato contenuto in `parallel_vector_matrix_product.out` sarà:

```
[TOTAL TIME SINGLE THREAD] 0.000000e+00 seconds.
Matrix:

5.000000 4.000000 3.000000
2.000000 1.000000 2.000000
3.000000 4.000000 5.000000


Vector x: [1.000000, 1.000000, 1.000000]

Result: [12.000000, 5.000000, 12.000000]
[TOTAL TIME PARALLEL] 1.760006e-03 seconds.
```

Chiaramente con un input così piccolo risulta essere decisamente migliore l'algoritmo iterativo single thread rispetto all'algoritmo parallelo con 8 threads. Lo scopo di inserire manualmente i dati all'interno della matrice **A** e il vettore **x** è quello di verificare il corretto funzionamento del programma dal punto di vista del risultato **b** che ritorna.

2.5.2 Output generando dati casualmente

Se si avvia lo script `start_parallel_vector_matrix_product.sh` con un numero di righe e colonne tale per cui il numero totale degli elementi della matrice è maggiore di 20, il programma genererà in maniera random i numeri per la matrice **A** e per il vettore **x**. Se eseguiamo il programma con 1000 righe e 1200 colonne con 8 threads, otteniamo:

```
Enter the number of rows: 1000
Enter the number of columns: 1200

1200000 random numbers have just been generated

1200 random numbers have just been generated
Enter the number of threads: 8
Print matrix A, vector x and result b on stdout? (Y/N): N
```

```
Il pbs parallel_vector_matrix_product.pbs e' stato accodato. \
I risultati saranno stampati in parallel_vector_matrix_product.out e \
parallel_vector_matrix_product.err.
```

Il risultato contenuto in `parallel_vector_matrix_product.out` sarà:

```
[TOTAL TIME SINGLE THREAD] 3.674006e-02 seconds.
[TOTAL TIME PARALLEL] 2.712011e-02 seconds.
```

Notiamo che l'algoritmo parallelo in questo caso risulta essere più veloce con uno speed up di:

$$S(8) = \frac{T(1)}{T(8)} = \frac{0.03674006}{0.02712011} = 1.43366$$

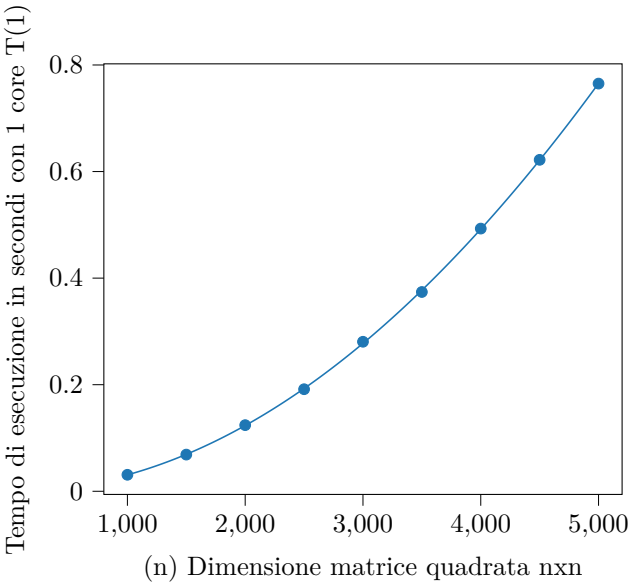
3 Grafici tempo di esecuzione

In questa sezione verranno mostrati alcuni risultati relativi al tempo di esecuzione dell'algoritmo parallelo del prodotto matrice vettore. L'obiettivo è quello di capire come le performance cambiano all'aumentare dell'input e dei cores. Vogliamo inoltre confrontare l'algoritmo parallelo con quello iterativo su un singolo core e capire quanto l'algoritmo parallelo "è migliore" di quello su un singolo thread facendo affidamento a metriche come lo speed up.

Si ricorda che la strategia adottata è quella di partizionare la matrice $\mathbf{A}$ in righe e distribuire tale righe agli n threads. Ogni punto disegnato sui grafici è la media dei risultati di *5 iterazioni*.

3.1 Grafico algoritmo su un singolo core

Sulle ascisse c'è la dimensione n della matrice quadrata. Sulle ordinate abbiamo il tempo di esecuzione del miglior algoritmo sequenziale su singolo core.



3.2 Grafici algoritmo parallelo su 8-cores CPU

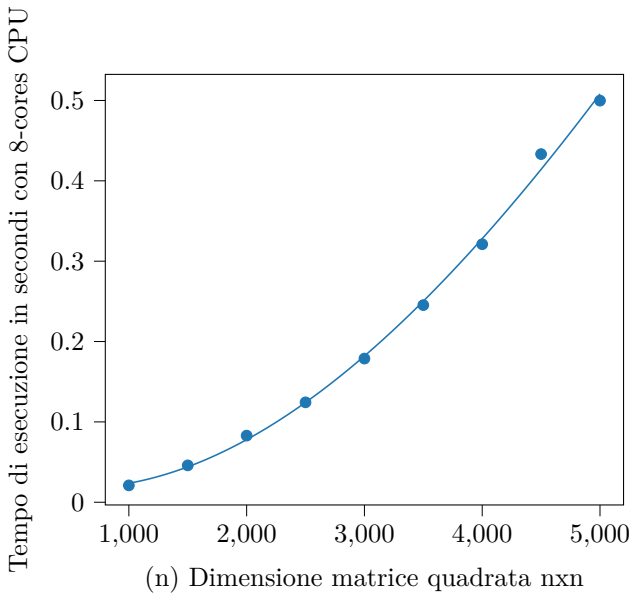
Il caso più interessante è sicuramente quello di sfruttare tutti i cores della macchina sul cluster SCoPE per eseguire l'algoritmo parallelo. In questa sezione verranno mostrati due grafici:

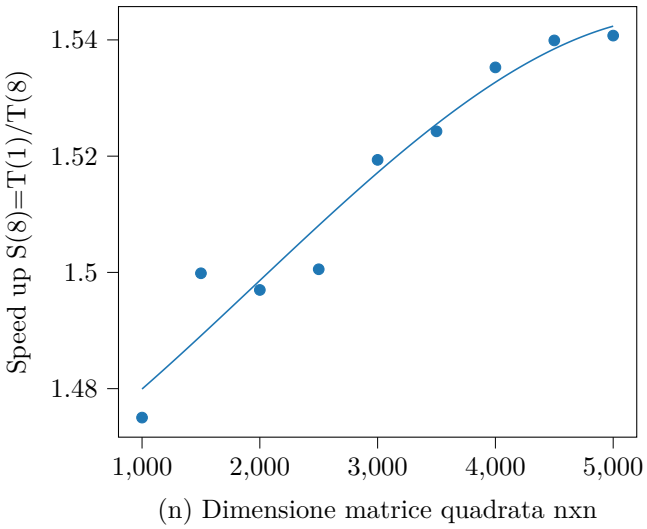
1. Grafico del tempo di esecuzione dell'algoritmo parallelo su 8-cores CPU all'aumentare della dimensione della matrice
2. Grafico del rapporto tra il tempo di esecuzione dell'algoritmo su singolo core e il tempo di esecuzione dell'algoritmo su 8-cores CPU all'aumentare della dimensione della matrice

La matrice in input è sempre una matrice quadrata, quindi sull'asse delle ascisse viene messo il valore di n cioè la dimensione della matrice quadrata $\mathbf{A} \in \mathbb{R}^n$.

Nel primo grafico abbiamo sulle ascisse la dimensione della matrice parte da 1000 e finisce a 5000 con un passo di 500. Sulle ordinate c'è il tempo di esecuzione dell'algoritmo parallelo in secondi.

Nel secondo grafico abbiamo sulle ascisse sempre la dimensione della matrice ma sulle ordinate in questo caso abbiamo lo speed up. Notiamo che all'aumentare dell'input l'algoritmo parallelo su 8-cores è sempre più veloce dell'algoritmo su singolo core.


Tempo di esecuzione in secondi con 8-cores CPU
0.5
0.4
0.3
0.2
0.1
0
1,000
2,000
3,000
4,000
5,000
(n) Dimensione matrice quadrata nxn

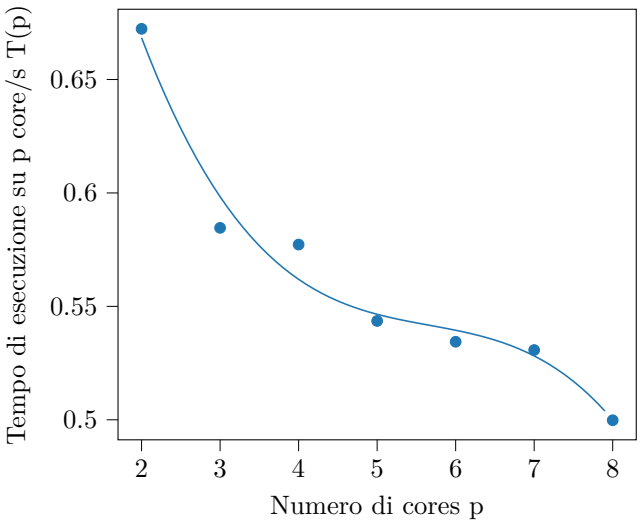

Speed up S(8)=T(1)/T(8)
1.54
1.52
1.5
1.48
1,000
2,000
3,000
4,000
5,000
(n) Dimensione matrice quadrata nxn

3.3 Grafici algoritmo parallelo all'aumentare dei threads

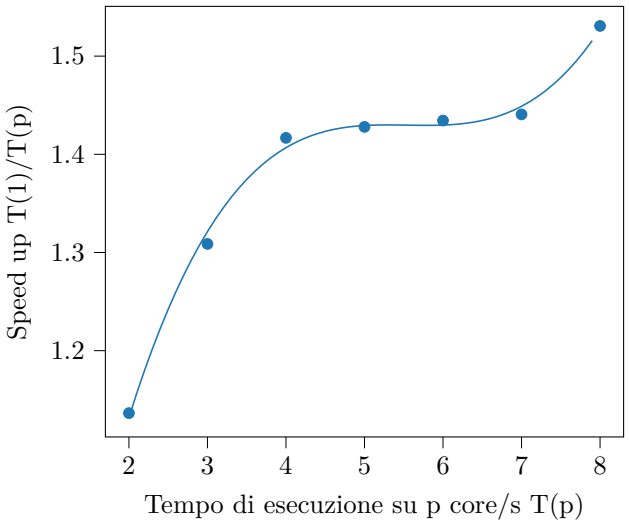
Due grafici:

1. Il tempo di esecuzione dell'algoritmo parallelo all'aumentare dei threads
2. Lo speedup all'aumentare dei threads

La dimensione della matrice quadrata è **fissata** a 5000 quindi $\mathbf{A} \in \mathbb{R}^{5000}$.



All'aumentare dei cores il tempo di esecuzione diminuisce



All'aumentare dei cores lo speedup aumenta.