Calcolo Parallelo e Distribuito - 2021/2022

Progetto 9

Michele Fiorentino
0124002085

Prof. Livia MARCELLINO

Indice

1 Definizione ed Analisi del problema

"Implementare un programma parallelo per l'ambiente multicore con np unità processanti che impieghi la libreria OpenMP. Il programma deve essere organizzato come segue: il core master deve generare una matrice di dimensione NxN, ogni core deve estrarre N/p righe e calcolare il prodotto puntuale tra i vettori corrispondenti alle righe estratte".

Sostanzialmente, ogni core deve estrarre N/p righe da una matrice generata A di dimensioni NxN e fare il prodotto puntuale fra queste. Quest'operazione consiste nell'effettuare il prodotto fra tutti gli elementi sulla stessa colonna, per tutte le colonne della matrice: il risultato sarà un nuovo vettore che avrà N colonne. Considerati p processori, avremo dunque p vettori risultato.

2 Descrizione dell'approccio parallelo

Ogni core estrarrà le N/p righe dalla matrice A, successivamente si occuperà di effettuare il prodotto puntuale tra questi vettori. Dato un core c, il prodotto di ogni elemento della colonna i-sima di questa sotto-matrice costituirà l'elemento i-simo del c-simo vettore risultato.
Si tratta di un algoritmo **completamente parallelizzabile**, infatti una volta che ogni core avrà completato la sua serie di prodotti non avremo bisogno di effettuare altre operazioni.
Il discorso sarebbe diverso se dovessimo effettuare anche il prodotto puntuale fra i vettori risultato (così da ottenere infine un unico vettore risultato), tuttavia quest'operazione non è richiesta dalla definizione del problema e dunque non ce ne preoccupiamo.

Per quanto riguarda la **valutazione dell'algoritmo parallelo**, siccome l'algoritmo descritto è completamente parallelizzabile, ci aspettiamo Speed Up ideale, Overhead nullo ed Efficienza massima.

2.1 Tempo di esecuzione dell'algoritmo T(n)

Un algoritmo per il calcolo del prodotto puntuale tra vettori di una matrice quadrata NxN, considerati p vettori e n/p righe, dà come risultato p vettori.
Ogni core, per ognuna delle n colonne, dovrà effettuare n/p prodotti, per un totale di n*(n/p) prodotti. Il prodotto fra gli elementi della colonna i-sima (di dimensione n/p) sarà il risultato dell'i-simo elemento del vettore risultato calcolato dal core.
Ovviamente, se consideriamo un algoritmo sequenziale p=1, il calcolo si riduce banalmente a: [NxN]molt.
Tuttavia il tempo di una moltiplicazione è circa quello di un'addizione, dunque possiamo considerare molt = add. Di conseguenza,

la **complessità computazionale dell'algoritmo sequenziale** sarà:

$$T_1(N \cdot N) = [N \cdot N]t_{calc}$$

la **complessità computazionale dell'algoritmo parallelo** sarà invece:

$$T_p(N \cdot N) = [N \cdot (N/p)]t_{calc}$$

2.2 Calcolo dello Speed Up

Siccome l'algoritmo è completamente parallelizzabile, avremo Speed Up ideale, ovvero il rapporto fra il tempo d'esecuzione dell'algoritmo sequenziale rispetto a quello parallelo è p.

$$S_p = \frac{T_1}{T_p}$$

Infatti:

$$S_p(N \cdot N) = \frac{T_1(N \cdot N)}{T_p(N \cdot N)} =$$

$$\frac{N \cdot N}{N \cdot (N/p)} =$$

$$p\frac{N \cdot N}{N \cdot N} = \text{p}$$

2.3 Calcolo dell'Overhead

Siccome l'algoritmo è completamente parallelizzabile, avremo Overhead nullo, ovvero la differenza fra lo speed up ottenuto e lo speed up ideale è 0.

$$O_h(pT_p - T_1)t_{calc}$$

Infatti:

$$O_h = pT_p(N \cdot N)t_{calc} - T_1(N \cdot N)t_{calc} =$$

$$p[N \cdot (N/p)]t_{calc} - [N \cdot N]t_{calc} =$$

$$(N \cdot N) - (N \cdot N) = 0$$

2.4 Calcolo dell'Efficienza

Siccome l'algoritmo è completamente parallelizzabile, avremo Efficienza massima, ovvero il rapporto fra lo Speed Up ed il numero di core è 1.

$$E_p(N \cdot N) = \frac{S_p(N \cdot N)}{p} =$$

$$\frac{p}{p} = 1$$

2.5 Ware-Amdhal

La legge di WA è un modo alternativo per scrivere lo Speed Up, che fa una distinzione fra parte puramente sequenziale e parte puramente parallela. Tuttavia in questa caso ho solo la parte puramente parallela.

Infatti:

$$S_p = \frac{1}{\alpha + (1-\alpha)/p}$$

diventa solo:

$$\frac{1}{p}$$

2.6 Calcolo dell'isoefficienza

Ricordiamo che lo scopo dell'isoefficienza è individuare il nuovo size n_1 tale che l'efficienza resti costante. Il tutto si riduce al calcolo degli Overhead e a fare il rapporto fra questi:

$$I = \frac{O_h(n_1/p_1)}{O_h(n_0/p_0)}$$

Tuttavia l'Overhead è sempre 0, dunque il rapporto sarebbe 0/0, ovvero una forma indeterminata. Per questo, per convenzione, l'isoefficienza è posta uguale ad **infinito**, ovvero posso usare **qualunque costante moltiplicativa** per calcolare n_1 e quindi controllare la scalabilità dell'algoritmo.

3 Descrizione dell'algoritmo parallelo

L'intero codice dell'algoritmo è presente nei file:

- **cpdProg9_2085.c**
- **printers_cpdProg9_2085.h** (per la visualizzazione)

In ogni caso, l'algortimo si basa su **due funzioni fondamentali**: parallelPunctualProduct() e partialPunctualProduct().
Un problema potrebbe presentarsi quando N è inferiore a P. In questo caso una possibile soluzione è far girare l'algoritmo con soli N core, tuttavia siccome siamo in ambiente parallelo diamo per scontato che abbiamo a che fare con una grande quantità di numeri e dunque non ritengo necessario implementare questa accortenza. Semplicemente, restituiamo un messaggio di errore.

Se p = 0 significa che si vogliono utilizzare tutti i core forniti dalla CPU, per fare ciò dobbiamo chiamare la funzione omp_get_num_threads() in una sezione parallela (altrimenti restituirebbe 1). È necessario che se ne occupi un solo thread, dunque lasciamo l'operazione al master.

La Matrice sarà generata casualmente. I numeri non potranno essere maggiori di un certo valore maxVal e non potranno mai essere = 0 (perché ovviamente annullerebbero la serie di prodotti, e per maxVal piccoli ed N grandi il rischio è che ne possano comparire molti).

Per comodità, ho definito il tipo "long long int" come "lli".

3.1 parallelPunctualProduct()

La sintassi è:

```
lli** parallelPunctualProduct(lli **A, int n, int p);
```

dove **A rappresenta una matrice di lli, n rappresenta il numero di righe/colonne, p rappresenta il numero di processori da utilizzare. L'algoritmo restituisce una matrice di lli, resVectors, la quale contiene i vettori calcolati dai core.
La funzione si occupa solo di inizializzare alcune variabili, allocare memoria, ed effettuare alcuni controlli prima di richiamare il cuore di questo algoritmo parallelo: partialPunctualProduct().

- **Alloca il numero di righe per resVectors:** per velocizzare il software in generale farò fare l'allocazione delle singole colonne ai core stessi.
- **Definisce la sezione parallela** nella quale verrano effettuati i calcoli.

Nella sezione parallela vogliamo passare A, n e p come variabili condivise in quanto vogliamo siano a disposizione di tutti i core, mentre il resto delle variabili saranno passate private. Qui specifichiamo anche il numero di thread/core da utilizzare.

Potrebbe presentarsi un problema quando n non è esattamente divisibile per p, per questo motivo possiamo assegnare una riga in più a tutti i processori il quale identificativo idt sia minore del resto calcolato da n/p.
Proprio perché implementiamo questa accortenza ogni core dovrà tener conto del numero di righe che dovrà considerare, e di uno step che possiamo considerare come un offset per tutti quei thread che NON DEVONO occuparsi del numero in più, e quindi "iniziano di qualche passo in avanti".

Infine, il core di posizione idt chiamerà la funzione partialPunctualProduct() alla quale passerà la propria porzione della matrice A, il numero di righe calcolato e il numero di colonne. Tale funzione resituirà il vettore risultato di posizione idt.

3.2 partialPunctualProduct()

La sintassi è:

```
lli* partialPunctualProduct(lli **A, int n, int m);
```

dove **A rappresenta una matrice di lli, n rappresenta il numero di righe, m rappresenta il numero di colonne da utilizzare.
Tale funzione si occupa di effettuare il prodotto puntuale vero e proprio.
L'algoritmo restituisce un vettore di lli, res, il quale contiene il risultato del prodotto puntuale per quella matrice (o sotto-matrice, nel nostro caso).
Sarà il core stesso ad occuparsi dell'allocazione della memoria per l'array.

Ad ogni nuova iterazione esterna, assegno a res[i] l'elemento i-simo della prima riga della matrice A (ovvero A[0][i]), questo perché se dobbiamo effettuare il prodotto fra tutti gli elementi presenti sulla stessa colonna di una riga mi conviene assegnare a res[i] il primo elemento di questa colonna. Potrei anche far partire res[i] da 1, tuttavia così facendo andrei ad affettuare un prodotto in più per ogni iterazione esterna, inoltre perderei ulteriore tempo per l'inizializzazione di tutti gli elementi di b ad 1.

(è simile al discorso del fare n somme quando inizializziamo sumtot = a[0] invece di sumtot = 0, così da risparmiarci di fare una somma).

Verrà da sé che res[i] all'iterazione j-sima rappresenterà il totale dei prodotti calcolati fino a quel momento per quella colonna.
j parte da 1 proprio perché è come se alla prima iterazione avessimo già fatto res[i] *= A[0][i], dove res[i] avrebbe avuto il valore iniziale di 1.

4 Input ed Output

Gli **input** devono essere passati come argomenti attraverso linea di comando. Se ne prevedono 4:

- **dim**: rappresenta il numero di righe/colonne;
- **maxVal**: rappresenta il valore massimo che può comparire durante la generazione della matrice;
- **numP**: rappresenta il numero di core da utilizzare. Se p=0 utilizziamo tutti i core messi a disposizione dal processore;
- **view**: indica se visualizzare graficamente l'algoritmo. Se view = 1 avviene la visualizzazione, altrimenti no.
  Per la visualizzazione è necessario che sia anche presente il file "printers_cpdprog9_2085.h".

Se view = 1, in **output** avremo:

- la matrice generata;
- la matrice generata suddivisa per blocchi di righe. Ogni blocco di righe rappresenta le righe assegnate ad ogni processore;
  Se N è esattamente divisibile per P, tutti i blocchi conterranno lo stesso numero di righe;
- la matrice che contiene i vettori risultato;
- la matrice che contiene i vettori risultato ma trasposta (questo per rendere la visualizzazione più semplice se N è molto grande).

5 Routine implementate

Le routine principali sono state già ampiamente descritte in precedenza, ma per completezza riassiumiamo tutte le routine presenti nei due file:

In **cpdProg9_2085.c**:

- createMatrix(): si occupa di generare una matrice casuale di dimensioni NxM, con valore massimo maxVal;
- parallelPunctualProduct(): si occupa di effettuare il prodotto puntuale fra i vettori di una matrice, dividendo la matrice in n/p (circa) righe e restituendo p vettori;
- partialPunctualProduct(): è il cuore di parallelPunctualProduct(). Applica l'algoritmo vero e proprio per effettuare il prodotto puntuale.

In **printers_cpdProg9_2085.h**:

- printMatrix(): stampa la matrice;
- printSubdividedMatrix(): stampa la matrice divisa per blocchi. Ogni blocco rappresenta il numero di righe assegnati ad ogni processore;
- printTrasposedMatrix(): stampa la matrice trasposta.

Viene inoltre utilizzata la libreria **OpenMP** per tutto ciò che concerne la parte parallela.

6 Analisi delle performance del Software

premessa:

nella pratica ovviamente non saremo in grado di ottenere tali risultati ideali, questo per via del k nella formula per il calcolo del tempo di esecuzione di un software: $\tau_p = k \cdot T(n) \cdot \mu$. Infatti noi nei calcoli precedenti abbiamo considerato l'algoritmo, e non il software.
Inoltre, dati lo stesso numero di core e la stessa dimensione del problema, τ_p potrebbe variare abbastanza da un'esecuzione all'altra sempre per via di k.
Per rendere questo k meno variabile ho deciso di eseguire lo stesso algoritmo, per ogni combinazione di p ed n, 100 volte, e di prendere la mediana di questi risultati così da ottenere una stima più probabile del tempo di esecuzione del software.

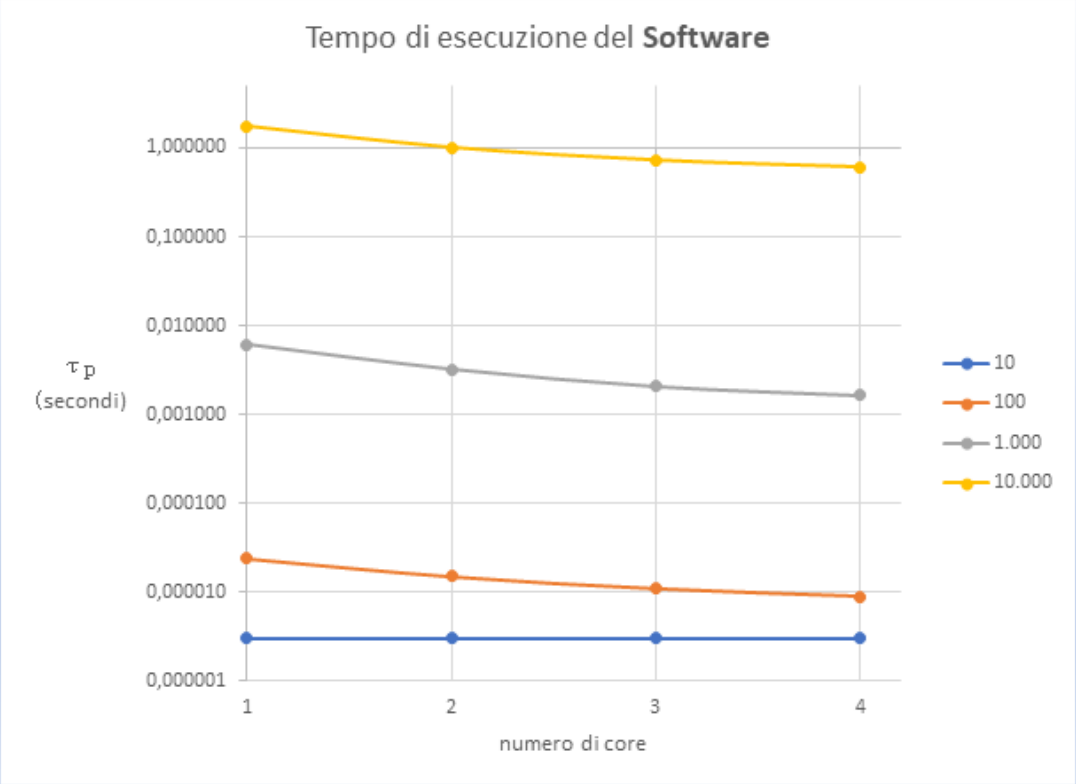
I tempi sono stati presi utilizzando la funzione omp_get_wtime().
Consideriamo il software eseguito con 1,2,3 e 4 core. Per ogni core, calcoliamo i tempi al variare di N (nello specifico: N = 10, N = 100, N = 1.000, N = 10.000).

Per convenzione: T_1 = tempo di esecuzione dell'**algoritmo parallelo su 1 processore**.

6.1 Tempo di esecuzione del software: τ_p

Sull'asse delle ascisse troviamo il numero di core, sull'asse delle ordinate il tempo in secondi (in scala logaritmica). Ogni curva rappresenta un diverso valore di N.

Ci aspettiamo una riduzione dei tempi all'aumentare del numero di core, ed infatti è quello che avviene in questo caso.
Possiamo notare come la curva N = 10 sia costante, questo probabilmente perché la dimensione del problema è così piccola da rendere irrilevante il numero di core utilizzati nell'esecuzione.



10000	1.769152	1.014286	0.725303	0.604703
1000	0.006119	0.003231	0.002065	0.001632
100	0.000024	0.000015	0.000011	0.000009
10	0.000003	0.000003	0.000003	0.000003
n/p	**1**	**2**	**3**	**4**

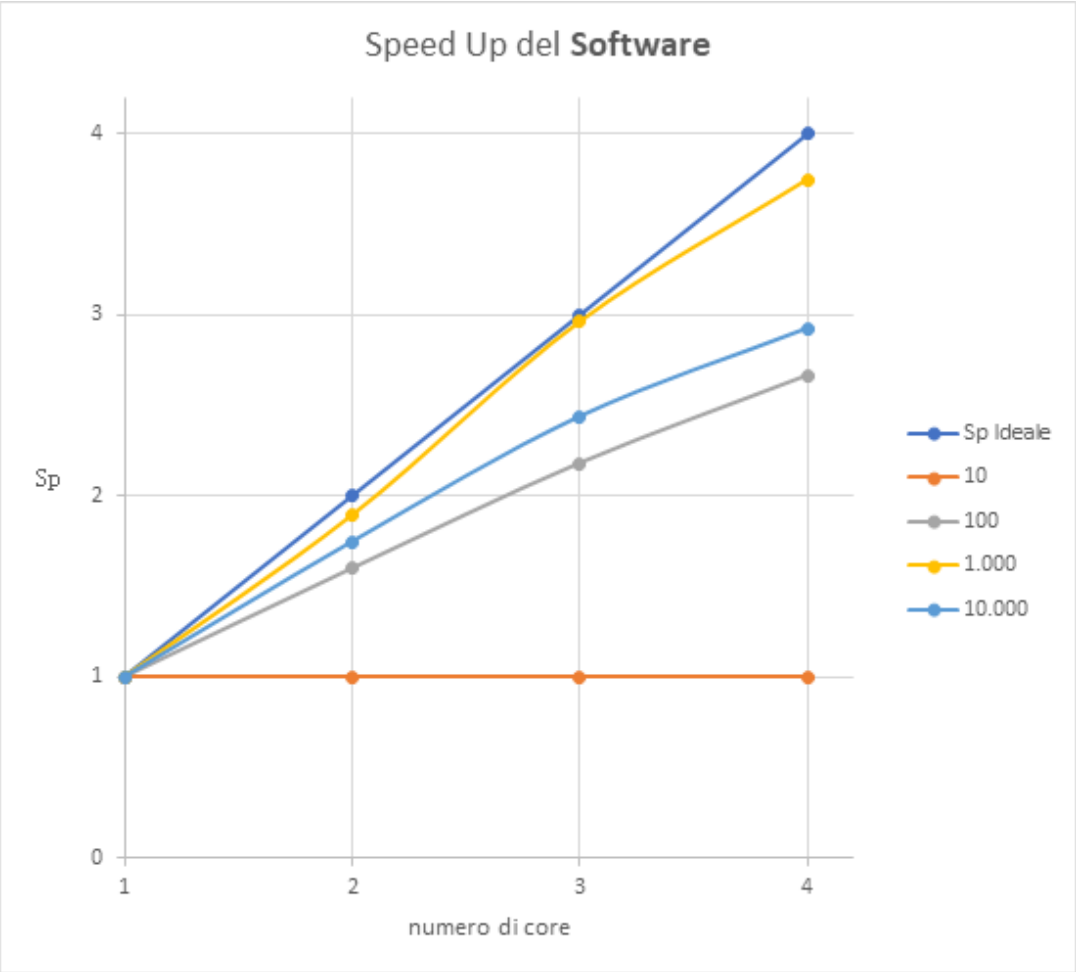
6.2 Speed Up del software: S_p

Sull'asse delle ascisse troviamo il numero di core, sull'asse delle ordinate lo Speed Up. Ogni curva rappresenta un diverso valore di N.

Normalmente ci aspettiamo che i valori siano al di sotto dello speedup ideale (bisettrice blu), ed infatti è quello che avviene in questo caso. Come già detto all'inizio della sezione, nella realtà è praticamente impossibile che un software abbia uno Speed Up = p.
Possiamo notare un paio di cose:

- la curva N = 10 è costante, questo perché lo era anche il tempo di esecuzione al variare dei core;

- otteniamo Speed Up piuttosto diversi al variare di N, sebbene in teoria dovrebbero essere uguali. Questo sempre perché il tempo di esecuzione del SW non combacia mai con quello dell'algoritmo.



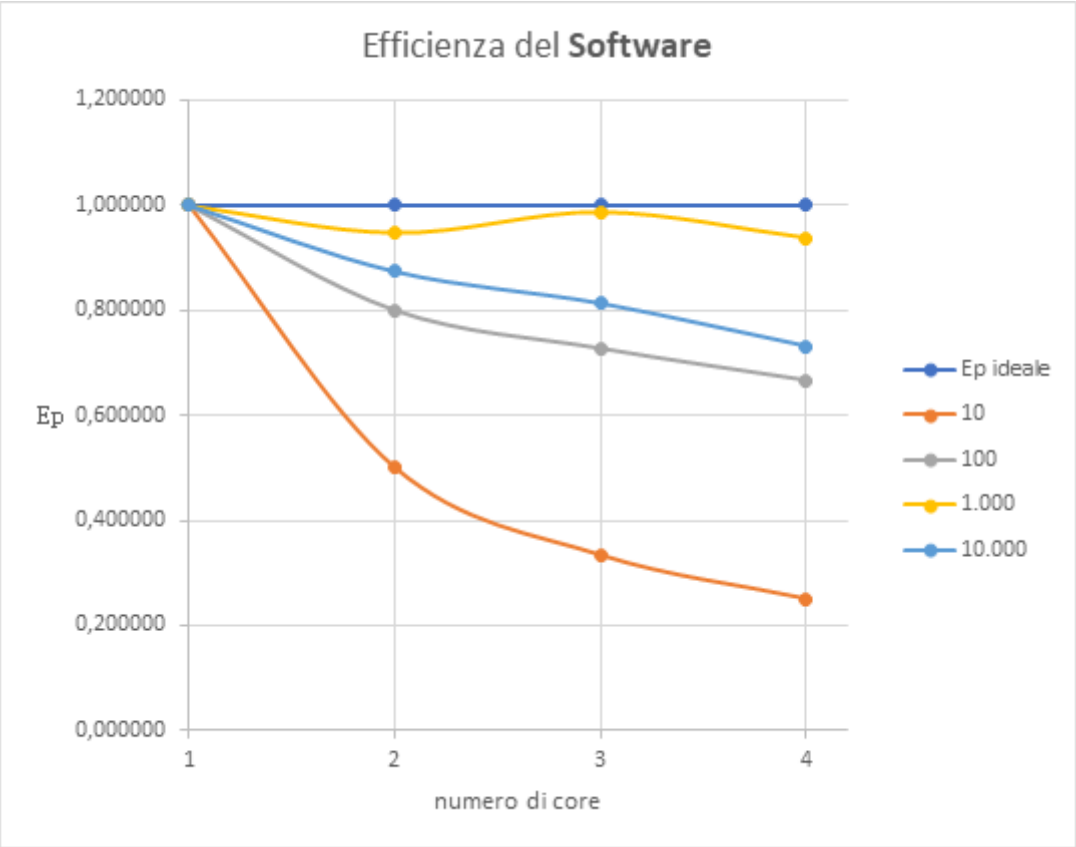
10000	1,000000	1,747434	2,439190	2,925654
1000	1,000000	1,893841	2,963196	3,749387
100	1,000000	1,600000	2,181818	2,666667
10	1,000000	1,000000	1,000000	1,000000
n/p	**1**	**2**	**3**	**4**

6.3 Efficienza del software: E_p

Sull'asse delle ascisse troviamo il numero di core, sull'asse delle ordinate l'Efficienza. Ogni curva rappresenta un diverso valore di N.

Nella realtà ci aspettiamo che i valori siano al di sotto di 1 (efficienza massima), e che all'aumentare del numero di core (per un N fissato) l'efficienza diminuisca.
In teoria potremmo usare una **qualunque costante moltiplicativa** per controllare la scalabilità dell'algoritmo. A dire il vero qui si presenta un caso anomalo: se N=1.000, abbiamo un'efficienza migliore con 3 core invece che con 2, quando in realtà dovrebbe essere il contrario (presumo ciò dipenda da altri fattori che non riguardano direttamente l'implementazione del puro algoritmo, come l'allocazione di memoria).

Inoltre notiamo anche come N = 10 cali a picco, in quanto il suo tempo di esecuzione rimane sempre lo stesso nonostante il numero di core aumenti.



10000	1,000000	0,873717	0,813063	0,731414
1000	1,000000	0,946920	0,987732	0,937347
100	1,000000	0,800000	0,727273	0,666667
10	1,000000	0,500000	0,333333	0,250000
n/p	**1**	**2**	**3**	**4**

7 Esempi d'uso

7.1 Output quando N è esattamente divisibile per P

```
michele@DESKTOP-O67PBAM:~/uni/cpd$ ./cpdProg9_2085.o 8 9 0 1
Generated Matrix =
1       4       7       3       6       2       1       9
5       3       6       4       4       6       8       1
5       7       6       4       9       2       2       4
3       6       9       4       9       8       6       1
9       3       1       3       2       1       2       6
1       7       7       3       1       6       3       4
1       8       7       9       7       8       2       1
5       1       2       4       6       7       2       3

Subdivided Generated Matrix =
1       4       7       3       6       2       1       9
5       3       6       4       4       6       8       1
---------------------------------------------------------------
5       7       6       4       9       2       2       4
3       6       9       4       9       8       6       1
---------------------------------------------------------------
9       3       1       3       2       1       2       6
1       7       7       3       1       6       3       4
---------------------------------------------------------------
1       8       7       9       7       8       2       1
5       1       2       4       6       7       2       3

Result Vectors =
5       12      42      12      24      12      8       9
15      42      54      16      81      16      12      4
9       21      7       9       2       6       6       24
5       8       14      36      42      56      4       3

Result Vectors (Transposed) =
5       15      9       5
12      42      21      8
42      54      7       14
12      16      9       36
24      81      2       42
12      16      6       56
8       12      6       4
9       4       24      3
```

7.2 Output quando N non è esattamente divisibile per P

```
michele@DESKTOP-O67PBAM:~/uni/cpd$ ./cpdProg9_2085.o 10 9 0 1
Generated Matrix =
6       7       7       9       9       9       5       4       4       7
9       6       5       3       6       3       5       8       6       3
1       8       2       6       6       6       1       6       4       9
7       7       4       4       4       3       1       8       5       4
3       2       7       6       4       3       6       8       9       2
9       7       7       8       1       4       4       9       9       5
8       6       2       1       7       5       3       7       1       5
8       2       6       6       7       8       6       3       4       5
4       3       1       8       1       1       2       5       9       9
9       6       3       9       4       1       2       4       5       3

Subdivided Generated Matrix =
6       7       7       9       9       9       5       4       4       7
9       6       5       3       6       3       5       8       6       3
1       8       2       6       6       6       1       6       4       9
-------------------------------------------------------------------------------
7       7       4       4       4       3       1       8       5       4
3       2       7       6       4       3       6       8       9       2
9       7       7       8       1       4       4       9       9       5
-------------------------------------------------------------------------------
8       6       2       1       7       5       3       7       1       5
8       2       6       6       7       8       6       3       4       5
-------------------------------------------------------------------------------
4       3       1       8       1       1       2       5       9       9
9       6       3       9       4       1       2       4       5       3

Result Vectors =
54      336     70      162     324     162     25      192     96      189
189     98      196     192     16      36      24      576     405     40
64      12      12      6       49      40      18      21      4       25
36      18      3       72      4       1       4       20      45      27

Result Vectors (Transposed) =
54      189     64      36
336     98      12      18
70      196     12      3
162     192     6       72
324     16      49      4
162     36      40      1
25      24      18      4
192     576     21      20
96      405     4       45
189     40      25      27
```

7.3 Output quando specifichiamo un preciso numero di processori

```
michele@DESKTOP-O67PBAM:~/uni/cpd$ ./cpdProg9_2085.o 10 9 2 1
Generated Matrix =
3       3       6       9       7       7       8       5       1       1
8       2       2       6       1       7       6       6       8       5
4       9       9       9       6       7       4       1       6       1
8       8       1       2       5       7       6       4       2       6
2       9       5       3       4       6       8       9       2       6
2       6       4       1       3       7       5       6       7       8
6       3       5       4       4       9       1       9       3       1
3       2       7       8       5       1       2       3       7       1
6       8       4       9       6       6       6       9       2       4
7       5       6       9       8       8       9       7       7       9

Subdivided Generated Matrix =
3       3       6       9       7       7       8       5       1       1
8       2       2       6       1       7       6       6       8       5
4       9       9       9       6       7       4       1       6       1
8       8       1       2       5       7       6       4       2       6
2       9       5       3       4       6       8       9       2       6
--------------------------------------------------------------------------------
2       6       4       1       3       7       5       6       7       8
6       3       5       4       4       9       1       9       3       1
3       2       7       8       5       1       2       3       7       1
6       8       4       9       6       6       6       9       2       4
7       5       6       9       8       8       9       7       7       9

Result Vectors =
1536    3888    540     2916    840     14406   9216    1080    192     180
1512    1440    3360    2592    2880    3024    540     10206   2058    288

Result Vectors (Transposed) =
1536    1512
3888    1440
540     3360
2916    2592
840     2880
14406   3024
9216    540
1080    10206
192     2058
180     288
```

7.4 Output quando non vengono inseriti sufficienti argomenti

```
michele@DESKTOP-O67PBAM:~/uni/cpd$ ./cpdProg9_2085.o 10 9 0
Not enough arguments. Insert: dim, maxVal, numP (default=0), view (0,1)
```

7.5 Output quando N è inferiore a P

```
michele@DESKTOP-O67PBAM:~/uni/cpd$ ./cpdProg9_2085.o 2 9 4 1
the number of cores cannot be less than N
```

8 Riferimenti bibliografici

- "Introduction to Parallel Computing", di Ananth Grama
- Slide del corso CPD 9 CFU A.A. 2021/2022, di Marcellino Livia
- Dispensa del corso CPD 9 CFU A.A. 2021/2022, di Marcellino Livia – Ambrosio Luigia

9 Appendice

9.1 cpdProg9_2085.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#include <time.h>
#include "printers_cpdProg9_2085.h"

typedef long long int lli;

lli** createMatrix(int n, int m, int maxVal){
  int i,j;

  lli **A = malloc(n*sizeof(lli*));
  for(i=0; i<n; i++)
    A[i] = malloc(m*sizeof(lli));

  for(i=0; i<n; i++)
    for(j=0; j<m; j++)
      A[i][j] = rand()%maxVal+1; //avoids zeros

  return A;
}

//algorithm to calculate the punctual product of a Matrix
lli* partialPunctualProduct(lli **A, int n, int m){

  //allocation for this core result Vector
  lli *res = malloc(m*sizeof(lli));

  int i,j;
  for(i=0; i<m; i++){
    res[i] = A[0][i];
    for(j=1; j<n; j++)
      res[i] *= A[j][i];
  }

  return res;
}

lli** parallelPunctualProduct(lli **A, int n, int p){

  int i,j,idt,nloc,r,step;
  double t,t1,t2;

  //allocation for the result Vectors
  lli **resVectors = malloc(p*sizeof(lli*));

  //t1 = omp_get_wtime();
  #pragma omp parallel shared(A,n,p) private(i,j,idt,nloc,r,step)
    num_threads(p)
  {
    //calculate the number of rows for each core
    idt = omp_get_thread_num();
```

```
52      nloc = n/p;
53      r = n%p;
54
55      if(idt < r){
56        nloc++;
57        step = 0;
58      } else {
59        step = r;
60      }
61
62      //get the result Vector from the function
63      resVectors[idt] = partialPunctualProduct(&A[idt*nloc+step],nloc
        ,n);
64    }
65    //t2 = omp_get_wtime();
66    //t = t2-t1;
67    //printf("Total Time with %d processors and n=%d: %f\n\n",p,n,t);
68
69    return resVectors;
70  }
71
72  int main(int argc, char** argv){
73
74    if(argc<5){
75      printf("Not enough arguments. Insert: dim, maxVal, numP (
        default=0), view (0,1)\n");
76      return -1;
77    }
78
79    srand(time(NULL));
80
81    int n = atoi(argv[1]);
82    int maxVal = atoi(argv[2]);
83    int p = atoi(argv[3]);
84    int view = atoi(argv[4]);
85
86    if(p == 0)
87      #pragma omp parallel master shared(p)
88      p = omp_get_num_threads();
89
90    if(n<p){
91      printf("the number of cores cannot be less than N\n");
92      return -2;
93    }
94
95    //CREATION OF A NxN MATRIX OF RANDOM NUMBERS
96    lli **A = createMatrix(n,n,maxVal);
97
98    //PUNCTUAL PRODUCT
99    lli **resVectors = parallelPunctualProduct(A,n,p);
100
101
102   //VISUALIZATION
103   if(view){
104     printMatrix("Generated Matrix",A,n,n);
105     printSubdividedMatrix("Generated Matrix",A,n,n,p);
106     printMatrix("Result Vectors",resVectors,p,n);
```

```
107     printTransposedMatrix("Result Vectors",resVectors,p,n);
108   }
109
110   return 0;
111 }
```

9.2 printers_cpdProg9_2085.h

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <omp.h>
4
5  typedef long long int lli;
6
7  void printMatrix(char* name, lli** A, int n, int m){
8
9    if(n == 0)
10     #pragma omp parallel master shared(n)
11     n = omp_get_num_threads();
12
13   printf("%s =\n",name);
14   for(int i=0; i<n; i++){
15     for(int j=0; j<m; j++){
16       printf("%lld\t", A[i][j]);
17     }
18     printf("\n");
19   }
20   printf("\n");
21 }
22
23 void printSubdividedMatrix(char* name, lli** A, int n, int m, int p
       ){
24
25   if(p == 0)
26     #pragma omp parallel master shared(p)
27     p = omp_get_num_threads();
28
29   int* nloc = calloc(p,sizeof(int));
30   int r = n%p;
31   for(int i=0; i<p; i++){
32     nloc[i] = n/p;
33     if(i < r)
34       nloc[i]++;
35   }
36
37   short flag = 0;
38   printf("Subdivided %s =\n",name);
39   int idx = 0;
40   for(int i=0; i<n; i++){
41     if(i%nloc[idx] == 0 && flag==1){
42       for(int k=0; k<m; k++)
43         printf("--------");
44       printf("\n");
```

```
45        idx++;
46      }
47      flag=1;
48      for(int j=0; j<m; j++){
49        printf("%lld\t", A[i][j]);
50      }
51      printf("\n");
52    }
53    printf("\n");
54  }
55
56  void printTransposedMatrix(char* name, lli** A, int n, int m){
57
58    if(n == 0)
59      #pragma omp parallel master shared(n)
60      n = omp_get_num_threads();
61
62    printf("%s (Transposed) =\n",name);
63    for(int i=0; i<m; i++){
64      for(int j=0; j<n; j++){
65        printf("%lld\t", A[j][i]);
66      }
67      printf("\n");
68    }
69    printf("\n");
70  }
```