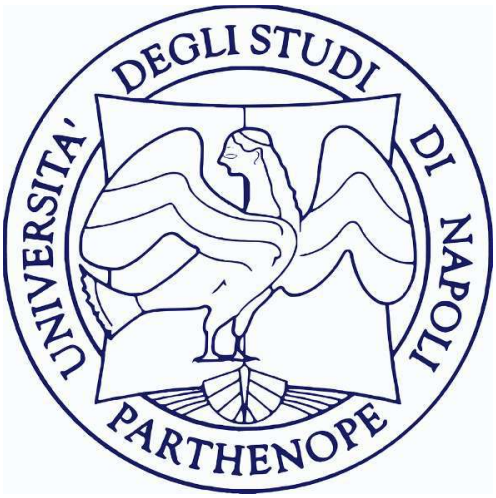


Università degli Studi di Napoli

"Parthenope"

Dipartimento di Scienze e Tecnologie

Corso di Laurea di Informatica

Anno Accademico: 2021/2022

RELAZIONE DEL PROGETTO DI "CALCOLO PARALLELO E DISTRIBUITO"

Proponente:

Calcopietro Francesco

Matricola:

0124002090

Professori:

Livia Marcellino

Pasquale De Luca

INDICE

Traccia 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 A di dimensione N×M. Quindi, i core devono collaborare per calcolare il minimo tra gli elementi delle colonne di propria competenza e conservare tale valore in un vettore c di dimensione M.

Definizione ed Analisi del problema

Come ben viene specificato dalla traccia, lo scopo da raggiungere, nella risoluzione di tale problema, è quello del calcolo dei minimi di ogni colonna di una matrice A generata con valori pseudo causali. La matrice, secondo la traccia, deve essere obbligatoriamente rettangolare, in quanto le dimensioni riportate sono N (numero di righe) x M (numero di colonne). Essendo M le colonne, e quindi essendoci M minimi nella matrice, si usufruisce di un vettore c, di dimensione M, per il salvataggio di questi ultimi. In particolare, i minimi di ogni colonna della matrice A verranno salvati in una posizione, all'interno di c, avente come indice lo stesso valore dell'indice di colonna corrispondente di A. Tale vettore dovrà essere visualizzato per il corretto raggiungimento dello scopo del software.

Obiettivo cardine di tale progetto è l'uso della libreria OpenMp, la quale permetterà di definire un algoritmo parallelo per la risoluzione di tale problema in un ambiente che sia MIMD-SM (*Multiple Instruction Multiple Data – Shared Memory*).

Prima di poter definire un algoritmo parallelo con tali caratteristiche è fondamentale analizzare l'omologo algoritmo sequenziale.

L'algoritmo sequenziale prevede le seguenti operazioni:

- Definizione delle dimensioni della matrice
- Allocazione dinamica della memoria per la matrice A secondo le dimensioni stabilite
- Generazione di valori pseudo casuali delle componenti della matrice
- Allocazione dinamica della memoria per il vettore c
- Ricerca del minimo di ogni colonna della matrice. Per fare ciò si hanno 2 cicli for. Quello esterno che scorre le colonne di A. Per ogni colonna c'è un ulteriore ciclo for che scorre ogni riga, in maniera tale da analizzare tutte le componenti della colonna corrente. Si inizializza la j-esima componente di c ad un valore massimo. Fatto ciò si confrontano le varie componenti della

colonna corrente con c[j], aggiornando quest'ultimo ogni volta che si trova un nuovo minimo.

- Stampa del vettore c e deallocazione della memoria.

Descrizione dell'approccio parallelo

L'algoritmo sequenziale proposto nella sezione precedente può essere parallelizzato.

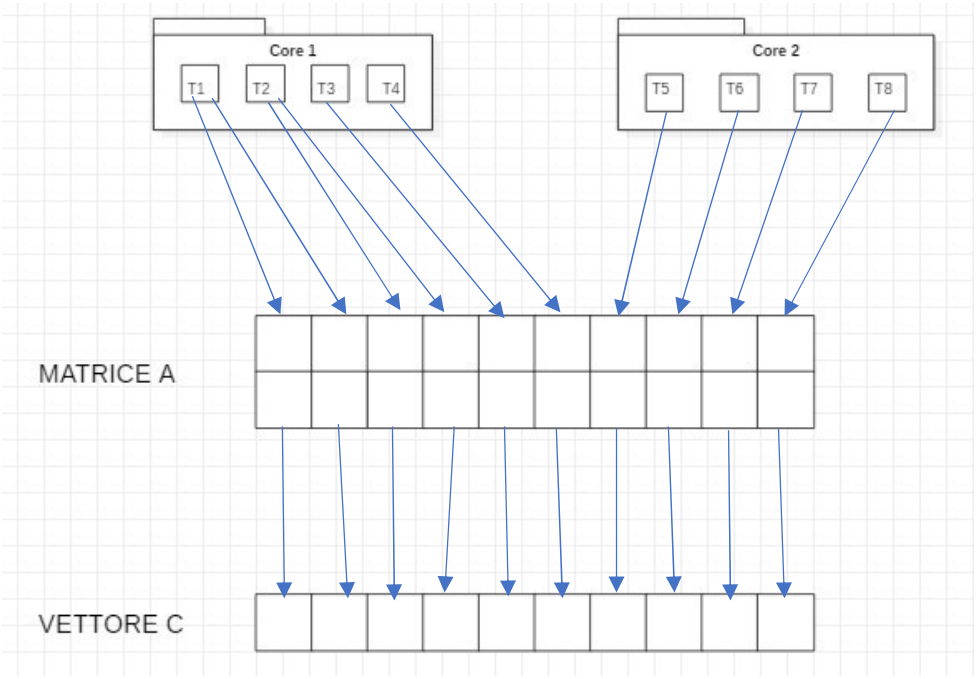
In particolare, secondo la traccia del problema e in riferimento all'algoritmo sequenziale, solo la sezione relativa alla ricerca dei minimi dovrà essere parallelizzata. Tutta la parte precedente ad essa e l'ultimo punto dovranno essere eseguiti in sequenziale dal core master.

Tenendo conto che l'ambiente di lavoro è MIMD-SM, e quindi lavorando con i core messi a disposizione dal calcolatore, si è pensato di distribuire le colonne della matrice A tra i thread dei vari core del calcolatore. In questo modo ogni thread sarà responsabile del calcolo dei minimi delle colonne della matrice A di sua competenza. In particolare, per capire quante colonne dovranno essere associate ai vari thread, si divideranno le colonne di A per il numero di thread usufruiti per l'esecuzione del programma. Questo modo di distribuire i dati permette di avere un equilibrio o bilanciamento del calcolo locale quanto più alto possibile. Questo è dimostrato anche nel caso in cui non c'è perfetta divisibilità tra il numero di colonne e il numero di thread. In tal caso verrà sfruttato lo standard della decomposizione del dominio. Esso prevede di analizzare il resto della divisione tra M e np. Non essendo 0 il valore del resto, tutte le colonne rimanenti verranno distribuite, a partire dal primo thread fino all'ultimo coinvolto, in maniera uniforme. In questo modo, anche se ci saranno thread responsabili di più colonne rispetto ad altri e quindi un piccolo sbilanciamento, questo farà ottenere al programmatore le migliori performance possibili.

Di conseguenza, grazie a questo modo di distribuire i dati tra i vari thread, questi ultimi potranno accedere solo alle corrispettive celle del vettore c e quindi collezionare in maniera sincronizzata i vari risultati evitando problematiche quali la race condition e perdita di integrità dei valori delle componenti del vettore c.

Per capire ancora meglio tale approccio, si consideri il seguente scenario.

Si ha a disposizione una macchina dual-core ove ogni core sfrutta 4 thread. Supponendo di avere una matrice 2x10, e siccome 10:8 dà come risultato 1 con resto di 2, allora l'approccio prevede che 2 thread siano responsabili di 2 colonne, mentre tutti gli altri di una sola colonna. Notare la seguente cattura.



Si può notare che con tale approccio si ha un algoritmo full parallel in quanto, ad eccezione del caso in cui si vogliano unire tutte le componenti del vettore c, non è necessario lo scambio di dati. Ci si limita semplicemente a far stampare i vari elementi dai singoli thread.

Si vuole, quindi, definire l'efficienza di tale approccio.

Innanzitutto, l'algoritmo sequenziale avrà una complessità di tempo:

$$T(NxM) = O(N*M)$$

Avendo ciò, si può definire il tempo d'esecuzione che sarà:

$$T_1(NxM)=[N*M]*tcalc$$

in quanto verranno effettuati N*M confronti.

Per quanto riguarda il tempo di esecuzione dell'algoritmo parallelo, tenendo conto della modalità di distribuzione dei dati precedentemente menzionata, sarà:

$$T_p(NxM)=[\frac{\mathbf{M}}{\mathbf{P}} * \ \boldsymbol{N}] \ *tcalc$$

A questo punto, è possibile calcolare lo speed-up pari a:

$$S_p(NxM)= \frac{\boldsymbol{T_1(NxM)}}{\boldsymbol{T_p(NXM)}}=\frac{[\boldsymbol{N*M}]}{[\frac{\boldsymbol{M}}{\boldsymbol{P}}* \boldsymbol{n}]}=P$$

Questo ci fa capire che, ma lo si poteva capire anche dal sapere che l'algoritmo è full parallel, che lo speed-up è pari a quello ideale.

Di conseguenza, l'Overhead risulta essere nullo, in quanto tutto è parallelizzabile.

Lo si dimostra anche attraverso i calcoli:

$$Oh = p*T_p(NxM) - T_1(NxM) = p*[\frac{M}{P} * N] * tcalc - [N*M]*tcalc = 0$$

Per quanto riguarda l'efficienza si ha che:

$$E_p(NxM) = \frac{Sp(NxM)}{p} = \frac{p}{p} = 1$$

Questo ci fa capire che l'algoritmo sfrutta al massimo il parallelismo del calcolatore.

Essendo Oh=0 ne consegue che l'isoefficienza, cioè la metrica che permette di capire se un algoritmo è scalabile o meno, è pari a:

$$I(n0,p0,p1) = \frac{Oh(n1*p1)}{Oh(n0*p0)} = \frac{0}{0}$$

cioè una forma indeterminata. Per convenzione, quindi, l'isoefficienza è posta uguale a ∞, ovvero si può usare qualunque costante moltiplicativa per calcolare n1 e quindi controllare la scalabilità dell'algoritmo.

Un'ultima analisi la si può fare calcolando lo Speed-Up con la Legge di Ware – Amdhal.

Essa può essere applicata in quanto è possibile distinguere la parte seriale (che non esiste) dalla parte parallela dell'algoritmo. In termini di formule si ha:

$$S_p = \frac{1}{\alpha + [\frac{1-\alpha}{p}]} \text{ ove}$$

$$1-\alpha = p * \frac{[\frac{M}{P} * n]}{[N*M]} = 1 \Rightarrow \frac{1}{P}$$

$$\alpha = 0$$

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

Anche attraverso la Legge di Ware – Amdhal si ottiene che lo speed – up è pari a quello ideale.

Descrizione dell'algoritmo parallelo

Il programma inizia con la dichiarazione delle variabili da utilizzare.

Subito dopo viene fatto un controllo sull'unico argomento di input del software (numero di thread da usufruire per eseguire il programma) in quanto si vuole

verificare se esso è stato definito e se ha un valore diverso da 0. In tal caso l'esecuzione continua; in caso opposto il programma termina.

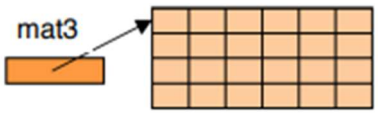
Si inizializza il seed per la generazione dei valori pseudo casuali tramite la *srand()* dopodiché si definiscono le dimensioni della matrice inserendo da tastiera, a run time, i valori di N e M. Vengono fatti i dovuti controlli sul valore di questi ultimi: N e M non dovranno essere uguali a 0 né uguali tra loro.

Si procede con l'allocazione dinamica della memoria sia per la matrice A che per il vettore c. In particolare, per la matrice A verrà utilizzata la seguente sintassi:

*int *A;*

A=(int)malloc(N*M*sizeof(int));*

cioè si alloca un blocco di memoria contiguo di N*M celle e la matrice è simulata come un array monodimensionale.



Malloc crea un blocco per tutta la matrice e accede agli elementi calcolandone la posizione (offset) riferita al primo elemento. Per accedere agli elementi della matrice si usa la seguente sintassi:

*A[IndiceRigaCorrente*M+indiceColonnaCorrente]*

Sintassi che verrà usufruita per il riempimento della matrice stessa.

Fatto ciò verrà allocata la memoria per il vettore c tramite la *calloc,* la quale, oltre a fare ciò, inizializza a 0 tutte le componenti di quest'ultimo.

Tutta questa serie di passi definisce la prima parte della fase sequenziale, in quanto viene eseguita dal solo core master. Di seguito si riporta lo pseudo-codice di tale parte:

```
begin
      if argc>1 && argv[1]
            numT := int(argv[1])
      else
            end
      "Generazione del seed con srand(time(0))"
      do
            "Lettura del numero di righe di A (N)"
      while(!N)
      do
            "Lettura del numero di colonne di A (M)"
      while(!M || M==N)
      "Allocazione della matrice A con malloc"
      for i = 0 to N do
            for j = 0 to M do
                  A[i*M+j] := 1+rand()%100
      "Allocazione del vettore c con calloc"
```

A questo punto si passa alla fase parallela e cioè al calcolo dei minimi delle varie colonne. Prima di far partire la regione parallela si richiama una routine della libreria OpenMp e cioè *omp_get_wtime().* Essa prende i tempi d'esecuzione della regione parallela permettendoci di effettuare, in seguito, le dovute considerazioni sulle performance dell'algoritmo.

Si richiama, quindi, la direttiva *#pragma omp parallel,* la quale crea un team di thread ed avvia un'esecuzione parallela di tutta la serie di istruzioni successiva. Siccome, però, verranno usufruiti 2 cicli for, uno innestato all'altro, l'idea è distribuire le iterazioni del solo ciclo for esterno, che scorre le colonne di A, tra i vari thread del team formatosi in precedenza. Per tal motivo la direttiva completa è *#pragma omp parallel for.* Le iterazioni del ciclo for interno saranno eseguite singolarmente da ciascun thread. A seguito della dichiarazione di tali direttive c'è, contestualmente, anche la dichiarazione di una serie di clausole necessarie al processore per l'utilizzo della libreria OpenMp. La prima clausola di cui discutere è *schedule* la quale stabilisce la modalità di distribuzione delle iterazioni del ciclo for da parallelizzare tra i vari thread in gioco. L'idea della distribuzione da adottare è stata menzionata nella sezione precedente. Quest'ultima è messa in atto tramite la modalità *static* la quale divide il numero totale delle iterazioni per il numero di thread garantendo un'uniformità e bilanciamento del carico più alto possibile. Tale clausola è prettamente del costrutto for. C'è anche la clausola *shared,* tipica del costrutto parallel che stabilisce cosa sarà condiviso tra i vari thread in gioco. In questo caso saranno condivise: N, M, A e c. C'è anche la clausola *private*, sempre tipica del costrutto parallel, che consente di definire tutto ciò che invece sarà privato per i thread in gioco. In tal caso si ha di privato gli indici i e j, necessari per scorrere il range di componenti della matrice A tra i vari thread. Sono privati in quanto si vuole, anche in questo caso, evitare race condition. Come ultima clausola c'è *num_threads* il quale stabilisce quanti thread dovranno concorrere nell'esecuzione del ciclo for seguente. Prenderà in ingresso l'input dell'intero programma, cioè numT.

Definito ciò parte la regione parallela consistente del doppio ciclo for citato poc'anzi. Di seguito lo pseudo codice della regione parallela:

```
t0 := omp_get_wtime()
#pragma omp parallel for schedule(static) shared(N,M,A,c) private(i,j) num_threads(numT)
      for j = 0 to M
            c[j] := 101
            for i = 0 to N
                  if A[i*M+j] < c[j]
                        c[j] := A[i*M+j]
```

Conclusasi tale regione, si richiama la *omp_get_wtime()* nuovamente, in maniera tale da fare la differenza con il valore ottenuto con la precedente chiamata.

L'algoritmo si conclude con la stampa del vettore risultato e del tempo d'esecuzione ottenuto dalla differenza e con la deallocazione della memoria occupata dalle varie strutture.

```
        t1 := omp_get_wtime()
        t := t1-t0
        "stampa di t"
        "stampa di c"
        "deallocazione della memoria per A"
        "deallocazione della memoria per c"
end
```

Input e Output

Il software necessita, appena viene utilizzato, di un unico parametro di input: un intero che stabilisce quanti thread verranno usufruiti per l'esecuzione del programma. Tale valore sarà salvato nella variabile numT. E' importante sottolineare il fatto che tale argomento non dovrà essere minore o uguale a 0.

Durante l'esecuzione verrà richiesto all'utente di stabilire quante righe e colonne dovranno comporre la matrice A. Anche queste non potranno essere pari a 0 né uguali tra loro.

Avendo fatto ciò, verranno visualizzate tutte le componenti della matrice A.

Dopodiché, durante la fase di calcolo locale, saranno visualizzate M stringhe che illustrano: quale thread ha lavorato, l'indice della nuova componente di c calcolata e il suo valore.

Conclusasi la fase di calcolo parallela, verrà visualizzato il tempo di esecuzione di quest'ultima e il vettore c, il quale rappresenterà il vero e proprio output del software.

Routine implementate

Come menzionato nelle sezioni precedenti si fa uso della routine *omp_get_wtime()* della libreria OpenMP per ottenere i tempi d'esecuzione della regione parallela.

Analisi delle performance del software

A questo punto, ci si vuole concentrare sulle tabelle e i grafici relativi al calcolo del tempo d'esecuzione, speed-up e dell'efficienza. Tutta la seguente analisi è stata effettuata avendo a disposizione un calcolatore:

HP 250 G7 Notebook PC

Intel® Core™ i3-7020U CPU @ 2.30GHz, 2300 MHz, 2 core, 4 processori logici

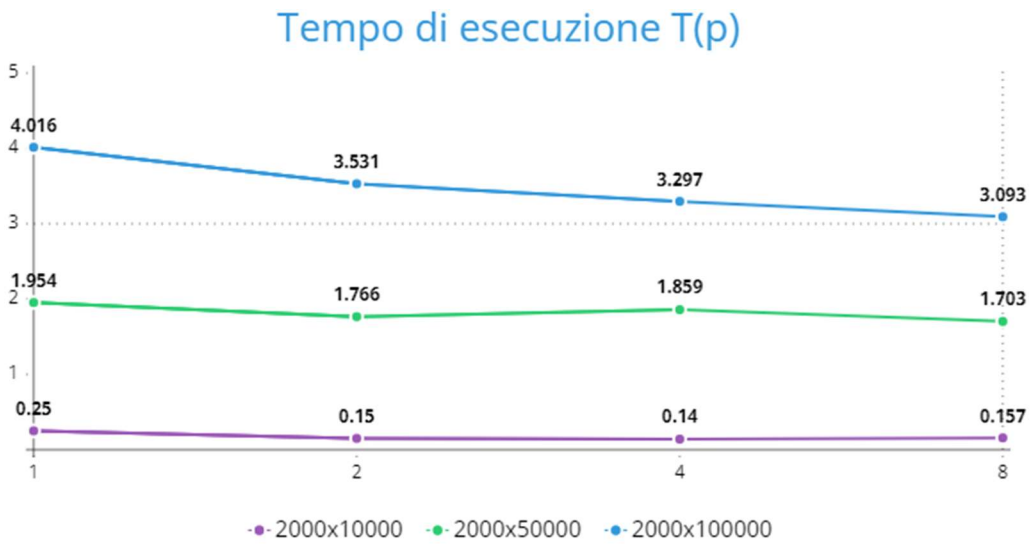
RAM: 4,00 GB

Tempo d'esecuzione

Nella tabella che segue sono riportati i valori del tempo di esecuzione registrati al variare del numero di thread T e del numero di righe (N) e colonne (M) della matrice A. Tutti i valori relativi ai tempi sono espressi in secondi (s).

M	N	T	Execution Time (s)
1×10^5	2×10^3	1	4,016
1×10^5	2×10^3	2	3,531
1×10^5	2×10^3	4	3,297
1×10^5	2×10^3	8	3,093

Tabella 1: Tempo di esecuzione



Tempi di esecuzione al variare del numero di threads e della dimensione della matrice

Speed-up

Nelle tabelle che seguono sono riportati i valori dello speed-up al variare della dimensione della matrice e del numero di thread. Si ricorda che esso misura la riduzione del tempo d'esecuzione dell'algoritmo parallelo rispetto all'algoritmo sequenziale, a parità della dimensione del problema.

M	N	T	Speed-up
1×10^5	2×10^3	2	1,13735485
1×10^5	2×10^3	4	1,21807703
1×10^5	2×10^3	8	1,29841577

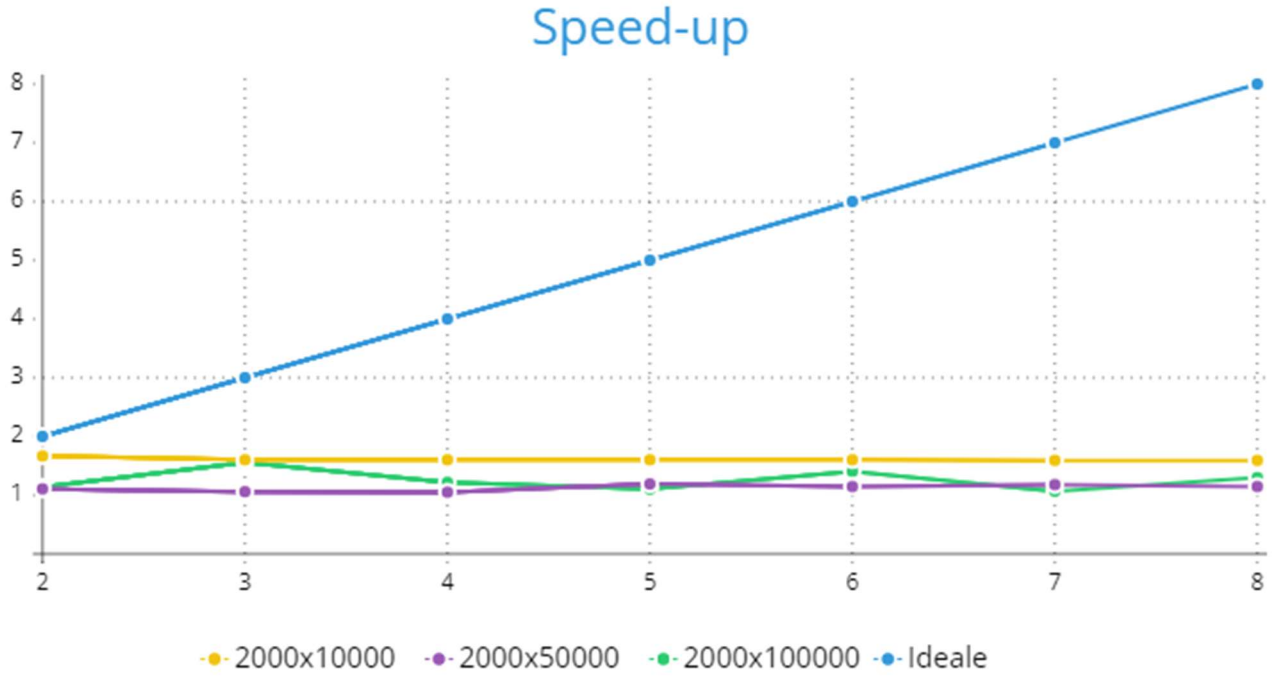
Tabella 2: Speed-up (1)

M	N	T	Speed-up
5×10^4	2×10^3	2	1,10645526
5×10^4	2×10^3	4	1,05110274
5×10^4	2×10^3	8	1,14738696

Tabella 3: Speed-up (2)

M	N	T	Speed-up
1×10^4	2×10^3	2	1,66666666
1×10^4	2×10^3	4	1,78571428
1×10^4	2×10^3	8	1,59235668

Tabella 4: Speed-up (3)



Speed-up al variare del numero di threads e della dimensione della matrice

Efficienza

Nelle tabelle che seguono sono riportati i valori dell'efficienza al variare della dimensione della matrice e del numero di thread. Essa indica quanto l'algoritmo sta sfruttando il parallelismo del calcolatore.

M	N	T	Efficienza
1×10^5	2×10^3	2	0,568677425
1×10^5	2×10^3	4	0,304519258
1×10^5	2×10^3	8	0,162301971

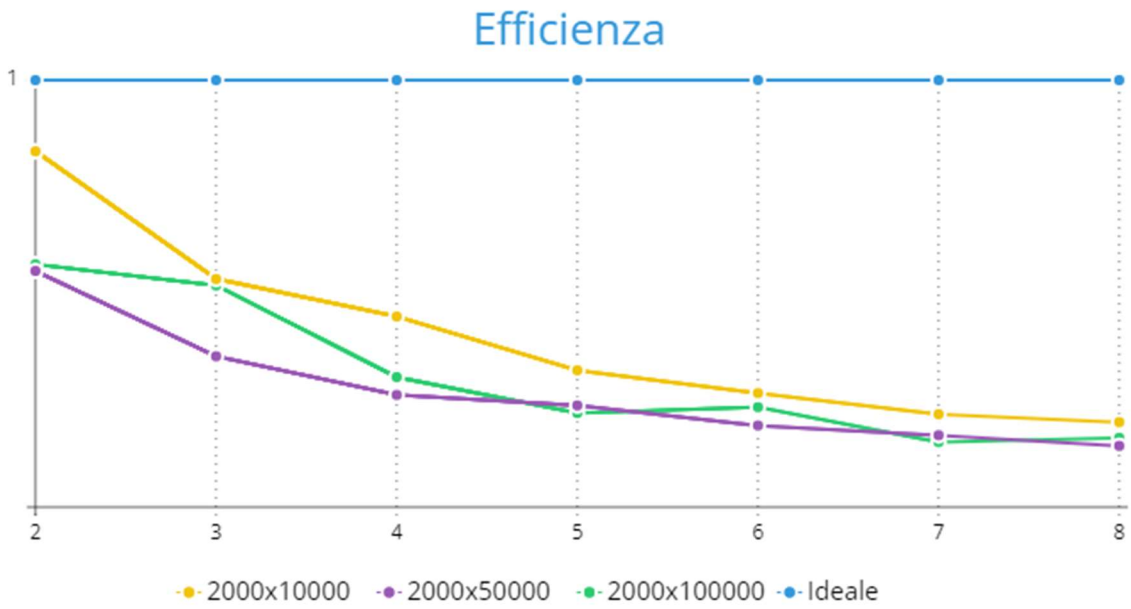
Tabella 4: Efficienza (1)

M	N	T	Efficienza
5×10^4	2×10^3	2	0,55322763
5×10^4	2×10^3	4	0,262775685
5×10^4	2×10^3	8	0,14342337

Tabella 5: Efficienza (2)

M	N	T	Efficienza
1×10^4	2×10^3	2	0,83333333
1×10^4	2×10^3	4	0,44642857
1×10^4	2×10^3	8	0,199044585

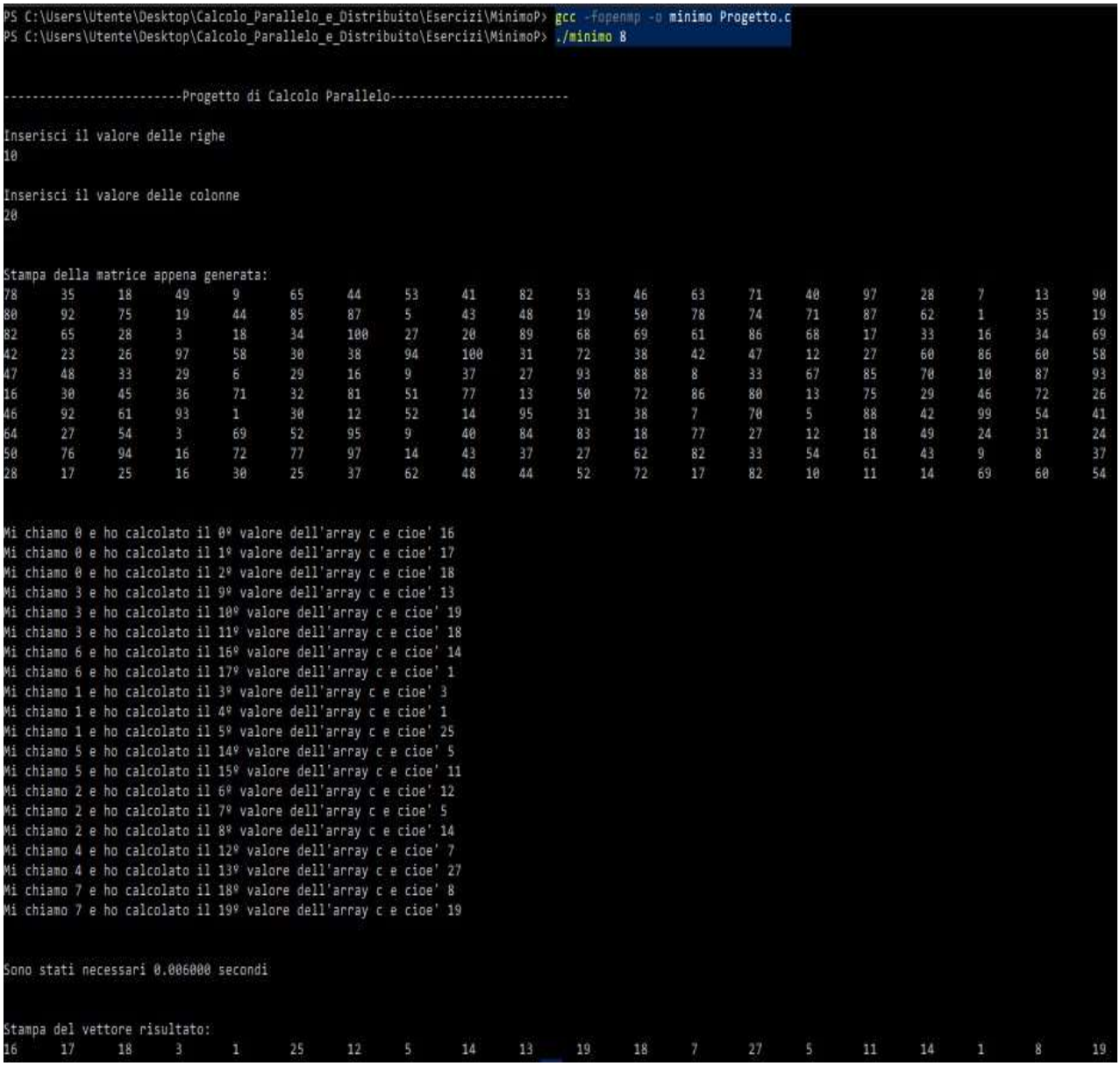
Tabella 6: Efficienza (3)



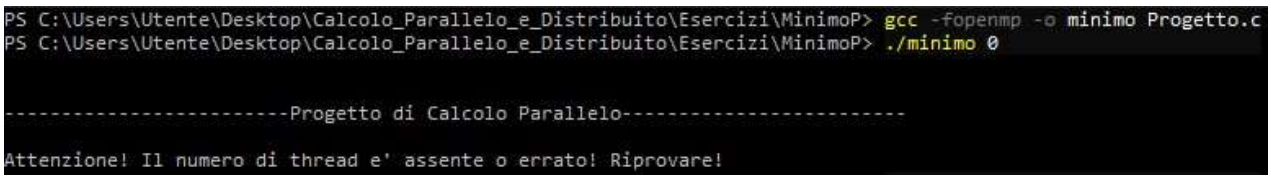
Efficienza al variare del numero di threads e della dimensione della matrice

Esempi d'uso

Di seguito verranno riportate 2 esecuzioni del programma; la 1° corretta, la 2° presenta un errore dovuto al passaggio di un valore errato al programma in input.



```
PS C:\Users\Utente\Desktop\Calcolo_Parallelo_e_Distribuito\Esercizi\MinimoP> gcc -fopenmp -o minimo Progetto.c
PS C:\Users\Utente\Desktop\Calcolo_Parallelo_e_Distribuito\Esercizi\MinimoP> ./minimo 8


-----------------------Progetto di Calcolo Parallelo-----------------------

Inserisci il valore delle righe
10

Inserisci il valore delle colonne
20


Stampa della matrice appena generata:
78      35      18      49      9       65      44      53      41      82      53      46      63      71      40      97      28      7       13      90
80      92      75      19      44      85      87      5       43      48      19      50      78      74      71      87      62      1       35      19
82      65      28      3       18      34      100     27      20      89      68      69      61      86      68      17      33      16      34      69
42      23      26      97      58      30      38      94      100     31      72      38      42      47      12      27      60      86      60      58
47      48      33      29      6       29      16      9       37      27      93      88      8       33      67      85      70      10      87      93
16      30      45      36      71      32      81      51      77      13      50      72      86      80      13      75      29      46      72      26
46      92      61      93      1       30      12      52      14      95      31      38      7       70      5       88      42      99      54      41
64      27      54      3       69      52      95      9       40      84      83      18      77      27      12      18      49      24      31      24
50      76      94      16      72      77      97      14      43      37      27      62      82      33      54      61      43      9       8       37
28      17      25      16      30      25      37      62      48      44      52      72      17      82      10      11      14      69      60      54


Mi chiamo 0 e ho calcolato il 0º valore dell'array c e cioe' 16
Mi chiamo 0 e ho calcolato il 1º valore dell'array c e cioe' 17
Mi chiamo 0 e ho calcolato il 2º valore dell'array c e cioe' 18
Mi chiamo 3 e ho calcolato il 9º valore dell'array c e cioe' 13
Mi chiamo 3 e ho calcolato il 10º valore dell'array c e cioe' 19
Mi chiamo 3 e ho calcolato il 11º valore dell'array c e cioe' 18
Mi chiamo 6 e ho calcolato il 16º valore dell'array c e cioe' 14
Mi chiamo 6 e ho calcolato il 17º valore dell'array c e cioe' 1
Mi chiamo 1 e ho calcolato il 3º valore dell'array c e cioe' 3
Mi chiamo 1 e ho calcolato il 4º valore dell'array c e cioe' 1
Mi chiamo 1 e ho calcolato il 5º valore dell'array c e cioe' 25
Mi chiamo 5 e ho calcolato il 14º valore dell'array c e cioe' 5
Mi chiamo 5 e ho calcolato il 15º valore dell'array c e cioe' 11
Mi chiamo 2 e ho calcolato il 6º valore dell'array c e cioe' 12
Mi chiamo 2 e ho calcolato il 7º valore dell'array c e cioe' 5
Mi chiamo 2 e ho calcolato il 8º valore dell'array c e cioe' 14
Mi chiamo 4 e ho calcolato il 12º valore dell'array c e cioe' 7
Mi chiamo 4 e ho calcolato il 13º valore dell'array c e cioe' 27
Mi chiamo 7 e ho calcolato il 18º valore dell'array c e cioe' 8
Mi chiamo 7 e ho calcolato il 19º valore dell'array c e cioe' 19


Sono stati necessari 0.006000 secondi


Stampa del vettore risultato:
16      17      18      3       1       25      12      5       14      13      19      18      7       27      5       11      14      1       8       19
```



```
PS C:\Users\Utente\Desktop\Calcolo_Parallelo_e_Distribuito\Esercizi\MinimoP> gcc -fopenmp -o minimo Progetto.c
PS C:\Users\Utente\Desktop\Calcolo_Parallelo_e_Distribuito\Esercizi\MinimoP> ./minimo 0


------------------------Progetto di Calcolo Parallelo------------------------

Attenzione! Il numero di thread e' assente o errato! Riprovare!
```

Appendice

Di seguito si riporta il codice scritto dell'algoritmo implementato in C.

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

int main(int argc, char *argv[])
{
    printf("\n\n------------------------Progetto di Calcolo Parallelo------------------------\n\n");
    int N,M,id_thread,i,j,numT;
    double t0,t1,t;

    //Controllo del parametro di input
    if(argc>1 && (atoi(argv[1])))
        numT=atoi(argv[1]);
    else
    {
        printf("Attenzione! Il numero di thread e' assente o errato! Riprovare!\n");
        exit(EXIT_FAILURE);
    }

    //Generazione del seed
        srand(time(0));

    //Lettura delle dimensioni della matrice A e conseguenti controlli
    do {
        printf("Inserisci il valore delle righe\n");
        scanf("%d",&N);
        if(N==0)
            printf("\nAttenzione! La matrice deve avere almeno una riga!\n\n");
    }
    while(N==0);
    do {
        printf("\nInserisci il valore delle colonne\n");
        scanf("%d",&M);
        if(M==N)
            printf("\nAttenzione! Il numero di colonne della matrice deve essere diverso dal numero di righe!\n\n");
        if(M==0)
            printf("\nAttenzione! La matrice deve avere almeno una colonna!!\n\n");
    }
    while(M==N || M==0);

    //Allocazione della memoria per la matrice A
        int *A;
        A=(int*)malloc(N*M*sizeof(int));
```

```
    //Generazione e stampa dei valori pseudo casuali della matrice A
    printf("\n\nStampa della matrice appena generata:\n");
    for(i=0;i<N;i++)
        {
        for(j=0;j<M;j++)
            {
                A[i*M+j]=1+rand()%100;
                printf("%d\t",A[i*M+j]);
            }
            printf("\n");
        }
    //Allocazione della memoria per il vettore c
        int *c=calloc(M,sizeof(int));
    printf("\n\n");

    //Prima chiamata a omp_get_wtime
    t0=omp_get_wtime();

    //Regione parallela
        #pragma omp parallel for schedule(static) shared(N,M,A,c) private(i,j) num_threads(numT)
            for(j=0;j<M;j++)
            {
            c[j]=101;
                for(i=0;i<N;i++)
                {
                    if(A[i*M+j]<c[j])
                    c[j]=A[i*M+j];
                }
            printf("Mi chiamo %d e ho calcolato il %d%c valore dell'array c e cioe' %d\n",omp_get_thread_num(),j+1,167,c[j]);
            }

    //Seconda chiamata a omp_get_wtime()
    t1=omp_get_wtime();
    t=t1-t0;

    //Stampa dei risultati
    printf("\n\nSono stati necessari %lf secondi\n",t);
    printf("\n\nStampa del vettore risultato:\n");
        for(i=0;i<M;i++)
        printf("%d\t",c[i]);

    //Deallocazione della memoria
    free(A);
    free(c);
    return 0;
}
```