Università degli Studi di Napoli

"Parthenope"

Dipartimento di Scienze e Tecnologie
Corso di Laurea: informatica 2021/2022

Relazione del Progetto "Calcolo Parallelo e Distribuito"

Proponenti:

Simone Cioffi Mat: 0124002047
Nicola Improta Mat: 0124002159
Mario Giordano Mat: 0124002104

Indice:

1. Definizione ed analisi del problema:

Traccia 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 leggere un vettore a, di dimensione N ed uno scalare beta. Quindi i core devono collaborare per verificare se nel vettore esista almeno un elemento uguale al valore beta.

Definizione del problema:

Come specificato dalla traccia, il nostro scopo è quello di verificare se, all'interno di un vettore a, ci sia uno o più valori uguali allo scalare beta. Il vettore, come dice la traccia, deve essere di dimensioni N, da inserire manualmente durante l'esecuzione del programma. Stessa cosa per quanto riguarda i valori interni e per lo scalare.

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

Occorre, prima di definire l'algoritmo, analizzare le operazioni presenti nell'algoritmo sequenziale:

- Definizione dimensioni vettore
- Allocazione dinamica del vettore
- Lettura ed inserimento degli elementi all'interno del vettore
- Lettura del valore dello scalare
- Confronto degli elementi presenti nel vettore con lo scalare, in caso di riscontro si aumenta l'indice che riguarda il numero di valori uguali. Il confronto avviene tramite un ciclo for che scala all'interno del vettore
- Stampa del valore dello scalare e del numero di riscontri ottenuti

2. Descrizione dell'approccio parallelo:

L'algoritmo sequenziale appena analizzato può essere parallelizzato.

In particolar modo, come richiesto dalla traccia, andremo a parallelizzare la parte di ricerca dello scalare all'interno del vettore. Tutta la parte precedente ad essa e la parte di stampa verrà eseguito in sequenziale dal core master.

Tenendo conto che l'ambiente di lavoro usato è di tipo MIMD-SM, abbiamo pensato, per l'approccio parallelo, di distribuire gli elementi dell'array tra i thread dei vari core a nostra disposizione, in questo modo ogni thread sarà responsabile della ricerca dello scalare all'interno della sezione del vettore di sua competenza. Il vettore verrà quindi distribuito tra vari thread, verrà diviso quindi il numero di elementi per il numero di thread utilizzati per la ricerca. Tale distribuzione permette di avere un bilanciamento della ricerca quanto più alto possibile.

Tale distribuzione è dimostrata anche quando il numero degli elementi risulta essere dispari, in questo caso si deve analizzare il resto tra la divisione degli n elementi e il numero di thread. Essendo il resto maggiore di zero, gli elementi rimanenti devono essere distribuiti in equal modo. In questo modo, anche se dei thread hanno più elementi da analizzare rispetto agli altri, ciò non causerà un peggioramento delle prestazioni.

Di seguito valutiamo l'efficienza dell'approccio parallelo utilizzato:

- **Complessità di Tempo:**

L'algoritmo sequenziale avrà una complessità di tempo pari a:

$$T(N) = O(N)$$

fatto questo possiamo definire il tempo di esecuzione, che sarà pari a:

$$T1(N) = N * Tcalc.$$

- **Tempo di esecuzione:**

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

$$T_p(N) = \left(\frac{N}{P}\right) * T_{calc}$$

- **Speed-up**

Dopo aver definito T1 ed Tp è possibile definire lo speed-up:

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

Questo dimostra, anche guardando l'algoritmo, che lo speed-up è pari a quello ideale.

- **Overhead**

Di conseguenza l'overhead è nullo, essendo tutto parallelizzabile. Tale affermazione si dimostra anche tramite il calcolo:

$$Oh = p * T_p(N) - T_1(N) = p * \left(\frac{N}{P}\right) * tcalc - N * tcalc = 0$$

- **Efficienza**

Per l'efficienza si ha:

$$E_p(N) = \frac{S_p(N)}{p} = \frac{p}{p} = 1$$

Questo afferma che l'algoritmo sfrutta al massimo il parallelismo del calcolatore.

- **Isoefficienza**

Essendo l'overhead nullo, si osserva che l'isoefficienza, è pari ad:

$$I(n_0, p_0, p_1) = \frac{Oh(n_1 * p_1)}{Oh(n_0 * p_0)} = \frac{0}{0}$$

Questo afferma che l'isoefficienza sia uguale ad una forma indeterminiate, per convenzione si afferma che l'isoefficienza sia uguale ad infinito, ovvero si puó utilizzare una qualunque costante moltiplicativa per andare a calcolare n1, ovvero controllare la scalabilità dell'algoritmo.

- **Ware-Amdhal**

Tramite la legge di Ware-Amdhal è possibile continuare l'analisi. Essa puó essere applicata in quanto la parte seriane dalla parte parallela dell'algoritmo è distinguibile, essendo nulla, calcolando lo speed-up con la legge si ha:

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

Dove si osserva:

$$1 - \alpha = p * \frac{\frac{N}{P}}{N} = 1 \rightarrow Sp = \frac{1}{\frac{1}{P}} = p \ (dove \ \alpha = 0)$$

Anche tramite legge di Ware-Amdhal si ottiene che lo speed-up è pari a quello ideale.

3. Descrizione dell'algoritmo parallelo:

L'algoritmo inizia con la dichiarazione delle variabili utilizzate. Subito dopo viene controllato il numero di threads presi in input tramite linea di comando assieme all'avviamento del software. Il programma controllerà il numero di thread preso in input, in caso sia nullo o minore uguale a 0 l'algoritmo darà errore e terminerà.

Dopo tale verifica , si controlla se il file d'input, pv, avente il vettore sia valida, in caso negativo termina il programma. In seguito per andar a controllare il numero di elementi, il size N, si inizializza un ciclo do-while dove si controlla il numero di elementi all'interno del file controllando ogni carattere ed aumentando di 1 N.

Di seguito, dopo aver preso il size, si inizializza il vettore a tramite richiamo di malloc, che lo alloca dinamicamente, prendendo in input:

$$int * a \rightarrow (int *) \ malloc(N * sizeof(int))$$

Dove malloc alloca dinamicamente al vettore a un blocco di memoria pari ad N moltiplicato per il size di un intero.

Dopodiché si fa la medesima cosa fatta per prendere in input il size anche per lo scalare beta. Infine, tramite l'utilizzo di N si resetta il pointer del file fv e si inizializza un ciclo while che inserisce gli elementi del file all'interno del vettore a.

Tutta questa parte viene eseguita dal core master, di seguito lo pseudo-codice:

```
begin
     if argc >1 and argv[1]
         numt:= int(argv[1])
     else
         end

    if (fv is null)
         end

    "Leggi il size del vettore contanto gli elementi all'interno del file"
    do
      c:= fgetc(fv)
      if (c:= '\n') N++
    while (c != EOF)

    "Allocazione Vettore a con malloc"

    "lettura elementi vettore a"
    while(i<N)
       "Leggi l'elemento nel file e inseriscilo in A"
       i++

    while(beta<0)
       "Leggi lo scalare beta"
```

Successivamente avviene la parte dell'algoritmo parallelizzato, ovvero la ricerca dello scalare all'interno del vettore. Prima di far partire la routine di parallelizzazione si richiama una funzione della libreria OpenMP: omp_get_wtime(), essa prende i tempi d'esecuzione della parte in parallelo, in seguito usata per calcolare il tempo complessivo dell'algoritmo.

Si richiama la direttiva, tramite #pragma, *omp parallel,* che crea un team di thread e avvia un'esecuzione in parallelo per tutta la serie di istruzioni successive. Per scalare all'interno del vettore per i vari threads si utilizza vicino a parallel anche *for*. Di seguito analizziamo anche le clausole ammesse, una di queste e *schedule* che stabilisce la modalità di distribuzione delle iterazioni del ciclo for da parallelizzare tra i vari threads. Questo è messo in atto tramite la tipologia di schedule, *static*, la quale divide il numero totale delle iterazioni tra i vari thread a disposizione. Insieme a schedule ci sono *shared* e *private*, la prima stabilisce gli elementi condivisi tra i vari threads che in questo caso sono: N, a, beta e findIndex (indice che indica il numero di occorrenze all'interno del vettore a) e private stabilisce gli elementi privati tra i vari threads, in questo caso sono: l'indice i (gli indici degli elementi da controllare per ogni thread). Infine, c'è la clausola *num_threads,* che va ad indicare il numero di threads da utilizzare per la parallelizzazione che prende in input numT (il numero di threads presi in input). Definito ciò inizia la ricerca degli elementi base, tramite un controllo, nel caso in cui l'elemento i-esimo del vettore a sia uguale allo scalare, allora si deve aumentare di uno l'indice *findIndex*, per aumentare ciò il thread entra nella direttiva *omp atomic update,* usata per garantire la mutua esclusione tra i vari threads, perché nel caso non venga garantita, si potrebbero perdere delle informazioni riguardanti l'indice. Di seguito lo pseudo-codice della parte parallela:

```
t0=omp_get_wtime();
#pragma omp parallel for schedule(static) shared(N,a,beta,findIndex) private(
i) num_threads(numT)
for i=0 to N {
    if a[i]==beta
       #pragma omp atomic update
            findIndex=findIndex+1;
```

Infine, conclusa la parte parallela, si richiama omp_get_wtime() nuovamente e si calcola la differenza tra il tempo pre-parallelizzazione e post-parallelizzazione. Dopo ottenuto il valore t si stampa tale valore ed il numero di occorrenze dello scalare beta all'interno del vettore a.

```
    t1=omp_get_wtime();
    t=t1-t0;
    "Stampa di t"
    "Stampa dello scalare beta ed findIndex"

    "Deallocazione di a"
end
```

4. Input & Output :

il software necessita di vari dati di input. L'unico preso tramite linea di comando e Il numero di threads. Gli elementi presi tramite terminale è lo scalare beta. Il vettore a ed il suo size N sono presi tramite lettura del file *"vertex.txt".*

In output verranno stampanti il tempo di esecuzione della parte parallela, il numero di occorrenze dello scalare beta all'interno del vettore a. Durante la fase di ricerca viene stampato ogni volta il thread che sta lavorando ed il numero di occorrenze in quel momento.

5. Routine implementate :

come analizzato nelle sezioni precedenti, l'unica routine utilizzata dall'algoritmo è *omp_get_wtime()* , routine della libreria OpenMP, per i tempi d'esecuzione della parte parallela.

6. Analisi delle performance del software:

In questo punto si vuole concentrare sulle analisi delle performance del software. Le seguenti analisi sono state effettuate sul seguente calcolatore:

MacBook Pro 2020

CPU: Apple Silicon M1 3.1GHz Octa-core

RAM: 8GB DDR4.

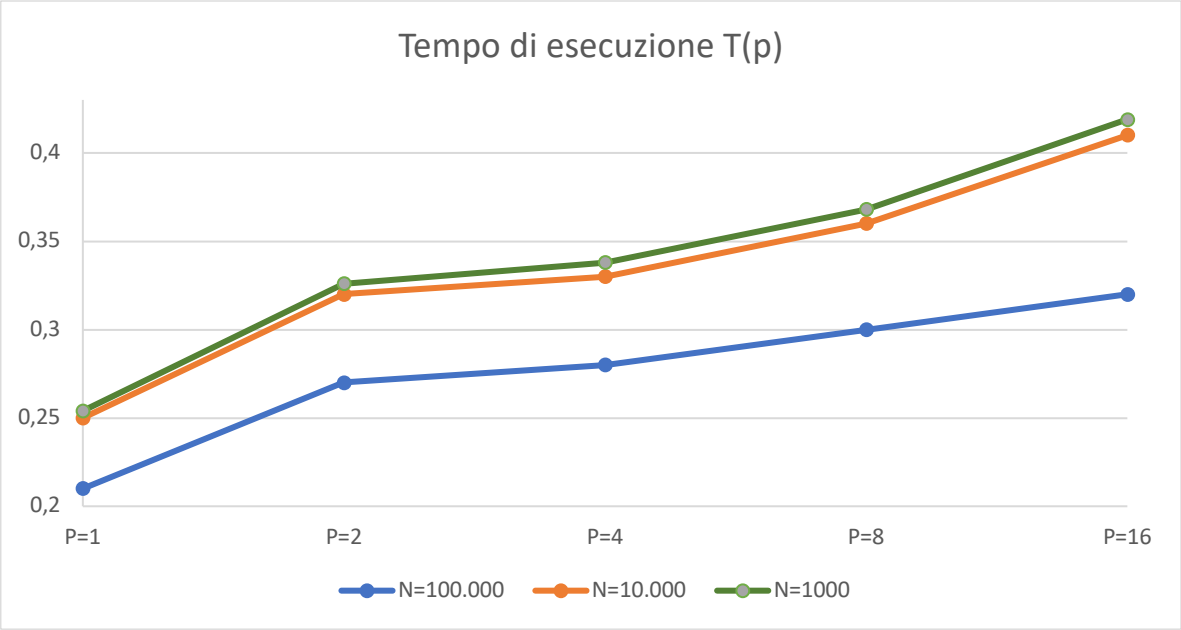
- Tempo di esecuzione:

Di seguito le tabelea con riportate i valori del tempo di esecuzione registrate al variare del numero di threads T, con ammesso size N del vettore, e lo scalare beta . tutti i valori relativi all'esecuzione sono riportati in secondi:

N	T	Beta	Execution Time (s)
$1 * 10^5$	1	3	0,21
$1 * 10^5$	2	3	0,27
$1 * 10^5$	4	3	0,28
$1 * 10^5$	8	3	0,30
$1 * 10^5$	16	3	0,32

N	T	Beta	Execution Time (s)
$1 * 10^4$	1	3	0,04
$1 * 10^4$	2	3	0,05
$1 * 10^4$	4	3	0,05
$1 * 10^4$	8	3	0,06
$1 * 10^4$	16	3	0,09

N	T	Beta	Execution Time (s)
$1 * 10^3$	1	3	0,004
$1 * 10^3$	2	3	0,006
$1 * 10^3$	4	3	0,008
$1 * 10^3$	8	3	0,008
$1 * 10^3$	16	3	0,009



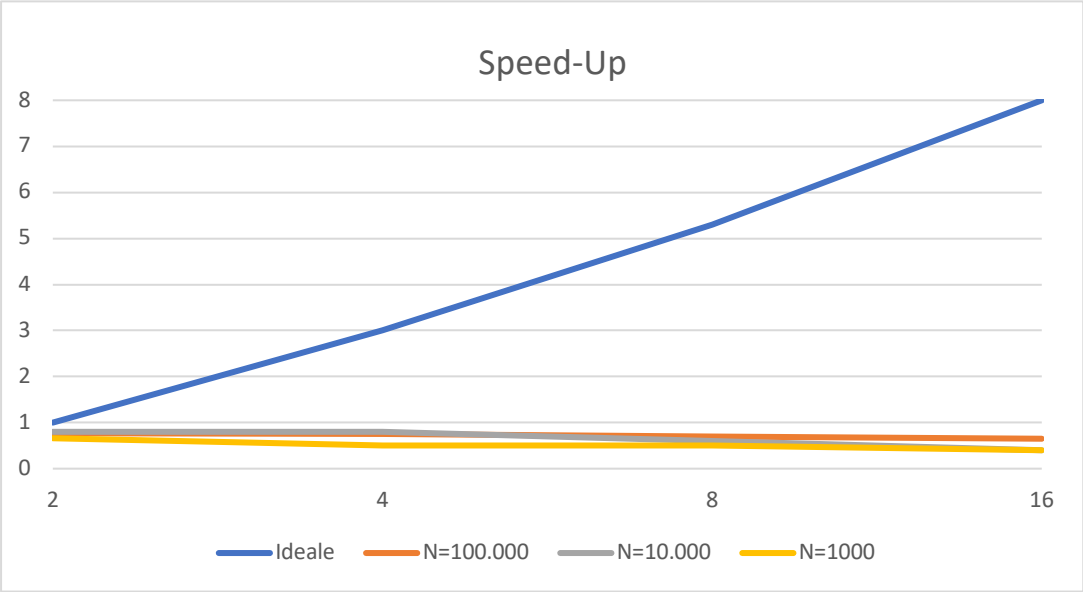
- Speed-Up

Di seguito le tabella con riportate i valori dello speed-up al variare della dimensione del vettore ed dei numeri dei thread:

N	T	Speed-up
$1 * 10^5$	2	0,77
$1 * 10^5$	4	0,75
$1 * 10^5$	8	0,70
$1 * 10^5$	16	0,65

N	T	Speed-up
$1 * 10^4$	2	0,8
$1 * 10^4$	4	0,8
$1 * 10^4$	8	0,6
$1 * 10^4$	16	0,4

N	T	Speed-up
$1 * 10^3$	2	0,66
$1 * 10^3$	4	0,5
$1 * 10^3$	8	0,5
$1 * 10^3$	16	0,4



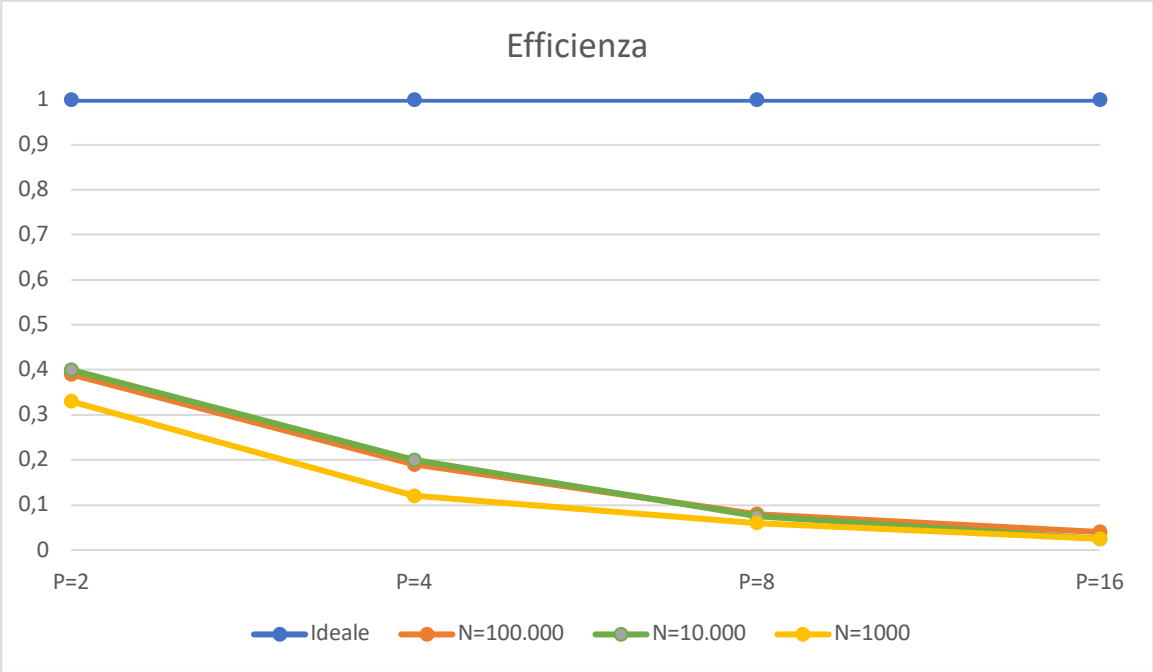
- Efficienza

Di seguito le tabelle con riportate i valori dell'efficienza al variare della dimensione del vettore ed dei numeri dei thread:

N	T	Efficienza
$1 * 10^5$	2	0,39
$1 * 10^5$	4	0,19
$1 * 10^5$	8	0,08
$1 * 10^5$	16	0,04

N	**T**	**Efficienza**
$\mathbf{1 * 10^4}$	**2**	**0,4**
$\mathbf{1 * 10^4}$	**4**	**0,2**
$\mathbf{1 * 10^4}$	**8**	**0,075**
$\mathbf{1 * 10^4}$	**16**	**0,025**

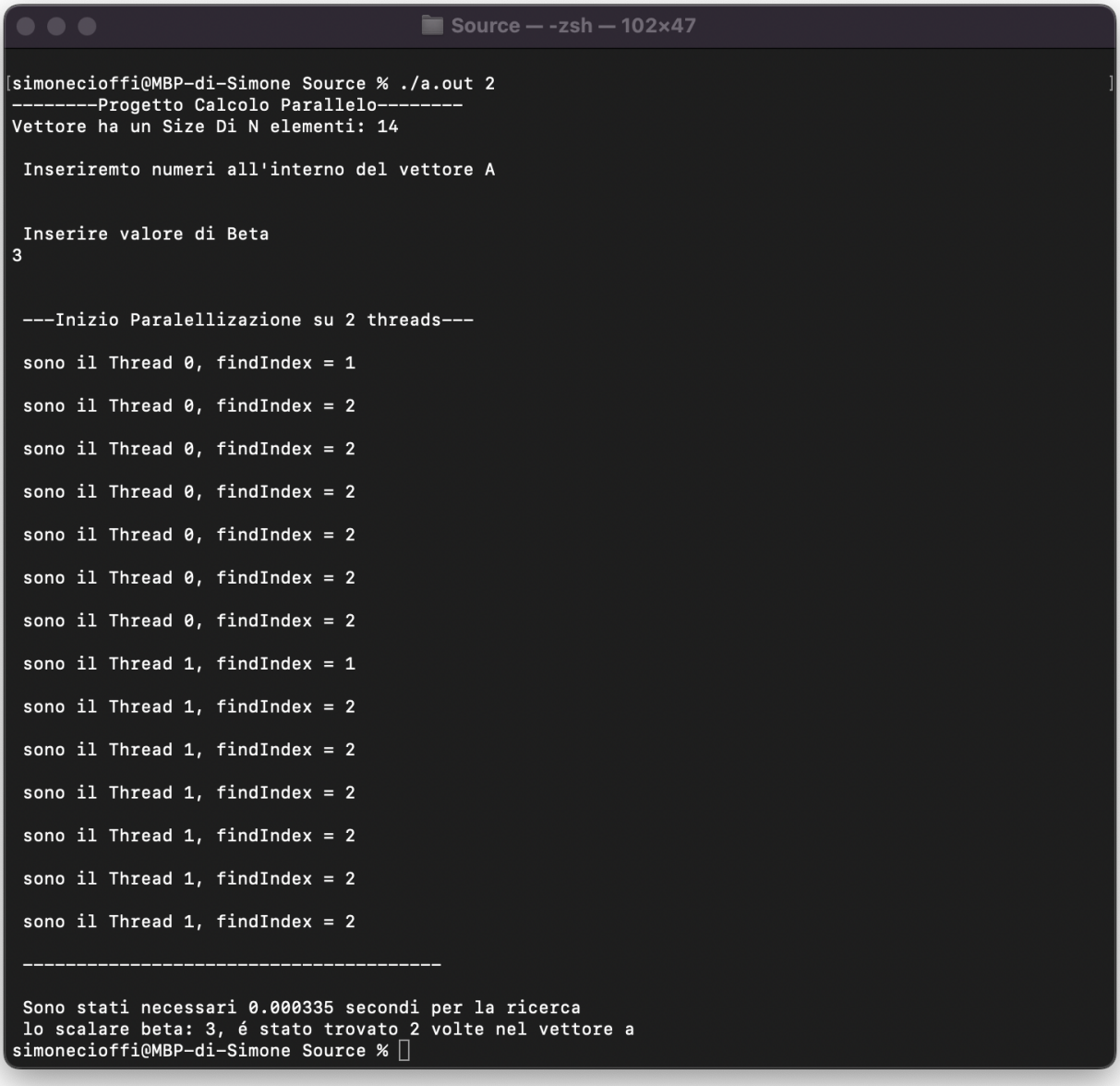
N	T	Efficienza
$1 * 10^3$	2	0,33
$1 * 10^3$	4	0,12
$1 * 10^3$	8	0,06
$1 * 10^3$	16	0,025

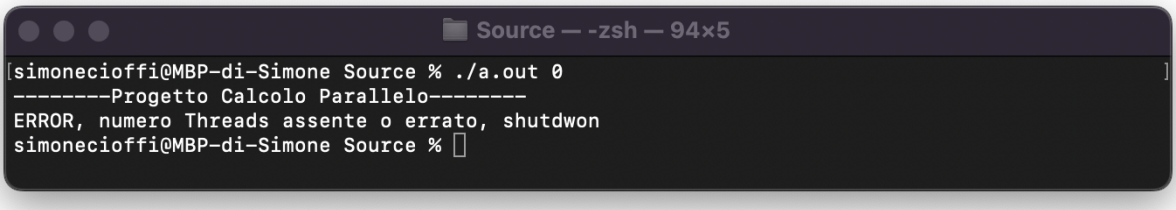


7. Esempi D’uso:

Di seguito vengono riportasti 2 esecuzioni del programma:

- La prima corretta
- La seconda errore di lettura dei thread





8. Appendice:

Di seguito si porta lo screen dell'implementazione in c del progetto:

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

int main(int argc, const char * argv[]) {
    int id_thread, N,fsize, beta, numT, *a,i,findIndex=0;
    double t0,t1,t;
    char c;
    FILE *fv;

    printf("--------Progetto Calcolo Parallelo--------\n");
    //controllo N Threads dati in input
    if(argc >1 && atoi(argv[1])) {
        numT = atoi(argv[1]);
    } else {
        printf("ERROR, numero Threads assente o errato, shutdwon \n");
        exit(EXIT_FAILURE);
    }

    //controllo file input
    if((fv=fopen("vector.txt","r"))==NULL){
        printf("Errore apertura file \n");
        exit(EXIT_FAILURE);
    }

    //Controllo N elementi
    N=1;
    do {
        c = fgetc(fv);
        if (c == '\n') N++;
    } while(c != EOF);

    //reset pointer
    rewind(fv);
    printf("Vettore ha un Size Di N elementi: %d \n",N);

    //allocazione moemoria vettore a
    a=(int*)malloc(N*sizeof(int));

    //Lettura vettore A
    printf("\n Inseriremto numeri all'interno del vettore A \n\n");

   //lettura elementi in A
    i=0;
    while (i<N) {
        fscanf(fv,"%d\n",&a[i]);
        i++;
    }
    //chiusura file dopo inserimento vettore
    fclose(fv);

     //Lettore scalare Beta
    beta=-1;
    while (beta<0) {
        printf("\n Inserire valore di Beta \n");
        scanf("%d",&beta);
        if(beta<0){
            printf("\n Attenzione!, valore di Beta minore di 0, riprovare");
        }
    }

     //prima che avviene la parallelizzazione
    t0=omp_get_wtime();
    printf("\n");

    printf("\n ---Inizio Paralellizazione su %d threads--- \n",numT);
    #pragma omp parallel for schedule(static) shared(N,a,beta,findIndex) private(i) num_threads(numT)
    for (i=0; i<N; i++) {
        if (a[i]==beta){
           #pragma omp atomic update
                findIndex=findIndex+1;
        }
        printf("\n sono il Thread %d, findIndex = %d \n",omp_get_thread_num(),findIndex);
     }


    //Seconda chiamata
    t1=omp_get_wtime();
    t=t1-t0;
    printf("\n ----------------------------------------\n");
    //Stampa risultato
    printf("\n Sono stati necessari %lf secondi per la ricerca",t);
    printf("\n lo scalare beta: %d, é stato trovato %d volte nel vettore a \n",beta,findIndex);

    //Free
    free(a);
    return 0;
}
```