Conteggio di bigrammi e trigrammi

Parallel computing

Elaborato di midterm

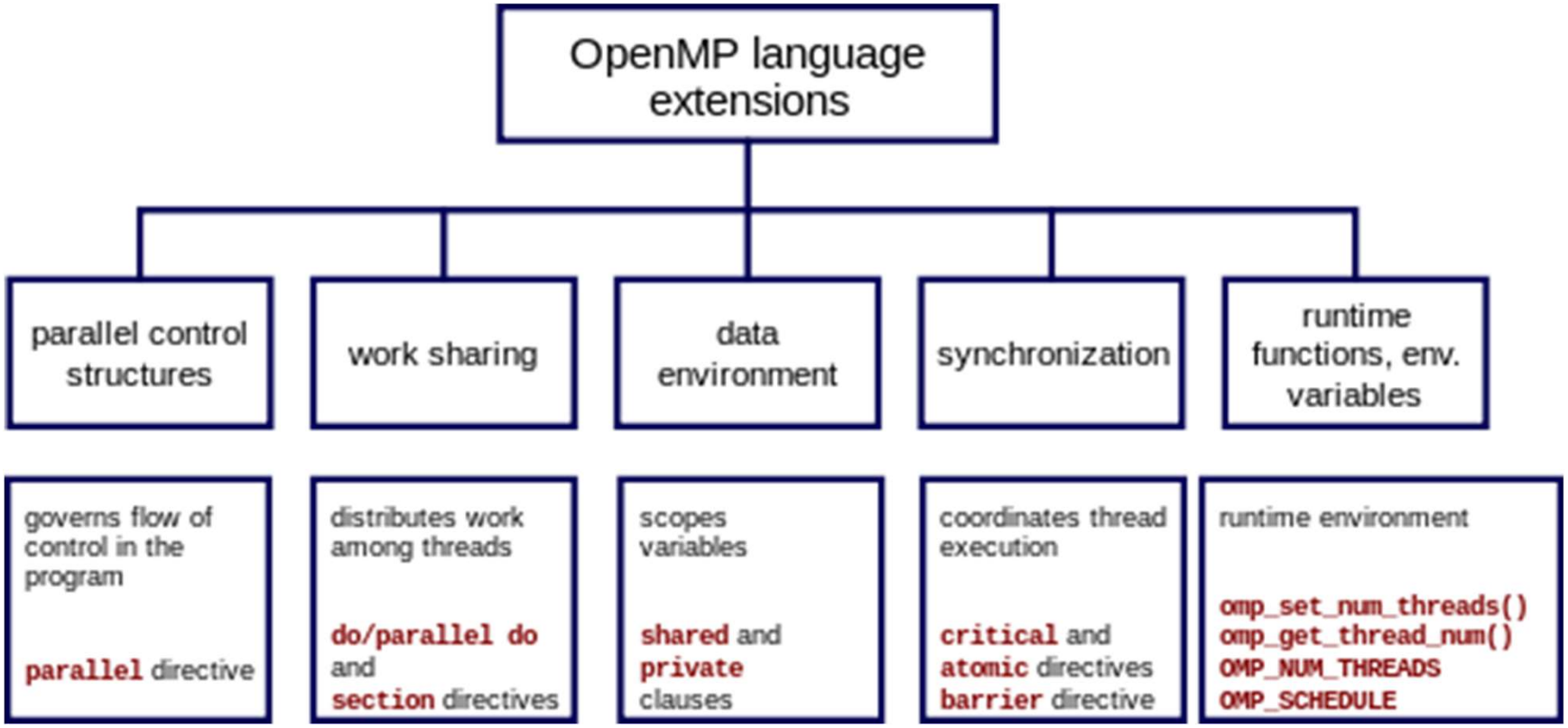
Massimiliano Mancini

Il conteggio dei bigrammi e trigrammi trova applicazione sia nello studio della struttura dei linguaggi che in alcune tecniche di crittoanalisi classica come, per esempio, l'analisi delle frequenze o in presenza del cifrario playfair

Il codice presentato con questo elaborato mette in luce come si possa sfruttare il parallelismo offerto da *openmp* al fine di migliorare le performance di calcolo. In particolare vedremo come, rispetto a un codice sequenziale, possiamo ricorrere a pattern di tipo *for loop* e *reduce* per analizzare un file di testo e conteggiare bigrammi e trigrammi

OpenMP è un’interfaccia applicativa (API) che semplifica la parallelizzazione del codice fornendo strumenti che consentono l’applicazione di pattern standard. Può essere utilizzata con diversi sistemi operativi per i linguaggi FORTRAN e C++

OpenMP si compone di diversi moduli



Il tipo di problema che dobbiamo risolvere presenta una prima parte molto lenta e non parallelizzabile (a meno di non passare a sistemi distribuiti) che riguarda la lettura dei dati da disco.

Una seconda parte che è oggetto di parallelizzazione e che riguarda il conteggio dei bigrammi e trigrammi.

Infine una terza parte, anch'essa non parallelizzabile, che riguarda la scrittura dei risultati su disco

Partendo da una versione sequenziale del codice, è possibile parallelizzarlo utilizzando direttive per il compilatore

```
#pragma omp parallel
```

Nel nostro codice, dopo aver letto il file da analizzare, suddividiamo in diversi thread l'intero contenuto del file in modo che ognuno di essi conduca parallelamente il conteggio dei bigrammi e trigrammi. La memorizzazione avviene in apposite strutture dati (array) locali a ogni thread. Infine in modo del tutto trasparente al programmatore openmp esegue una riduzione delle strutture dati locali

```
#pragma omp for reduction (+:freqb,freqt) schedule (static, blocksize)
```

La seguente istruzione indica che si sta entrando in una sezione ad esecuzione parallela. Deve essere utilizzato un pattern di tipo for-loop con dimensione statica dei blocchi, ognuno di dimensione blocksize.

Alla fine del loop deve essere condotta una riduzione per somma dei valori contenuti negli array freqb e freqt che conteggiano rispettivamente i bigrammi e i trigrammi

```
#pragma omp for reduction (+:freqb,freqt) schedule (static, blocksize)
```

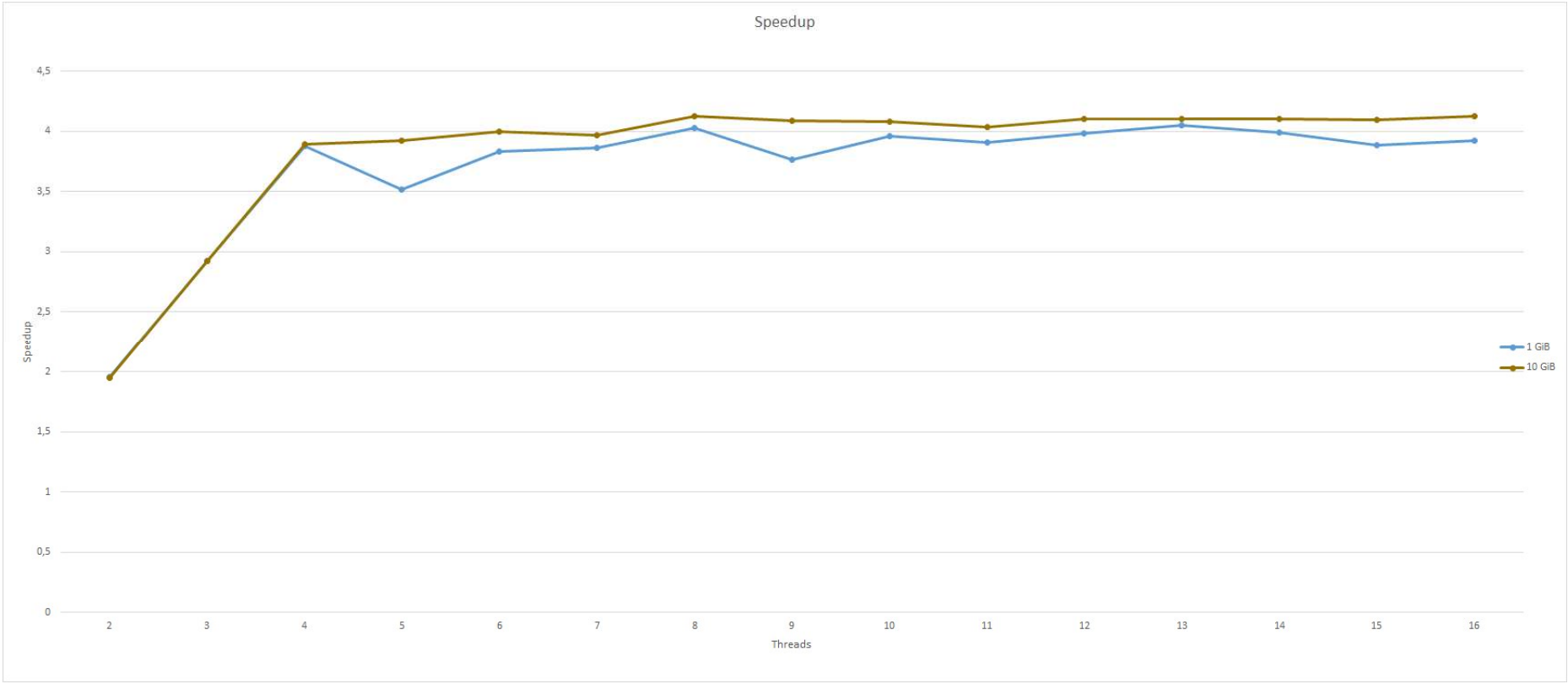
OpenMP consente l'utilizzo di istruzioni richiamabili a runtime. Il numero di thread infatti viene impostato tramite

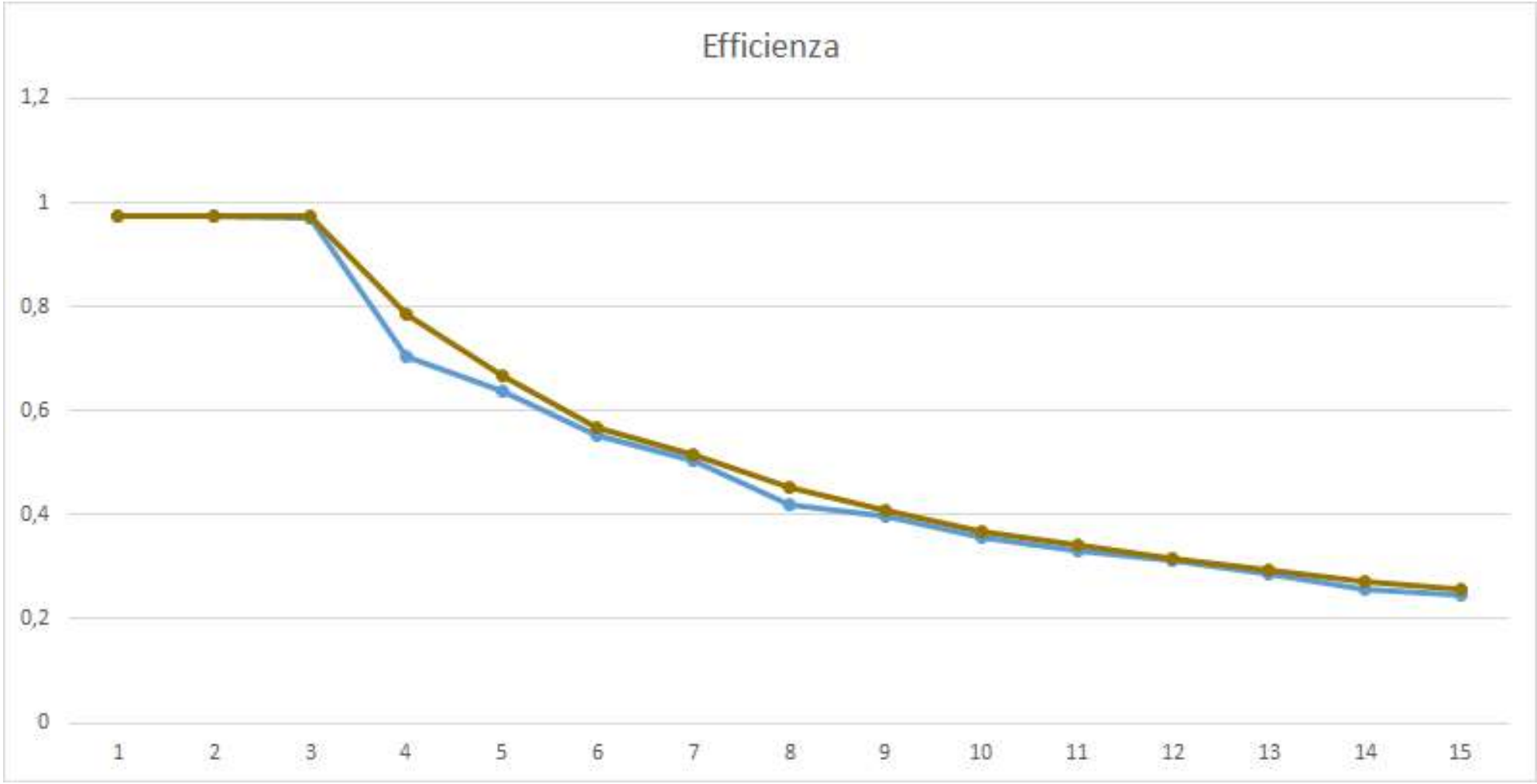
```
omp_set_num_threads(nthreads);
```

In alternativa si possono utilizzare delle variabili di ambiente

```
OMP_NUM_THREADS
```

Lo speedup viene calcolato solo per la parte parallela del codice, quella seriale infatti, avendo a che fare con lettura e scrittura da disco, risulta relativamente lenta. Il suo conteggio rischia di introdurre una componente numericamente rilevante ma poco interessante


Speedup
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
Speedup
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Threads
1 GiB
10 GiB


Efficienza
1,2
1
0,8
0,6
0,4
0,2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Decriptazione di password

Parallel computing

Elaborato di endterm

Massimiliano Mancini

L’elaborato mostra come utilizzare *POSIX Threads* in linguaggio C per sfruttare al meglio la capacità di elaborazione allo scopo di decriptare una password di 8 caratteri selezionati nell'alfabeto [./0-9A-Za-z] della quale conosciamo solo la codifica hash ottenuta con funzione DES.

Si tratta di un semplice attacco di forza bruta con uno spazio delle chiavi di cardinalità pari a $64^8 = 2^{48} = 281.474.976.710.656$

L’approccio utilizzato segue la logica della forza bruta, quindi si suddivide lo spazio delle chiavi in parti uguali tra tutti i thread in modo che ognuno di essi conduca i tentantivi parallelamente.

Si tratta quindi di un semplice pattern for-loop in cui i dati da esaminare sono suddivisi tra i diversi processori

I thread sono ottenuti tramite funzioni fork-join.

Una variabile condivisa, protetta da mutex, viene utilizzata per coordinare i diversi thread. In particolare viene utilizzata per indicare ai thread che la password è stata trovata e che quindi posso uscire dal ciclo while principale

Il linguaggio C non ha un supporto nativo per il parallelismo. E' però possibile ricorrere all'interfaccia applicativa *POSIX Threads* o *pthreads* che fornisce una serie di tipi, funzioni e costanti attraverso l'utilizzo di un header

```
pthread.h
```

oltre a una libreria linkata in fase di compilazione

```
-lpthread
```

Pthread consente controlli fini sui diversi aspetti legati al parallelismo e ai thread; al costo di una certa complessità di utilizzo.

Sono presenti funzioni di

- creazione dei thread
- join
- mutex
- read-write locks
- variabili condizione con sistemi di segnalazione (signal, wait)
- barriere
- sincronizzazione

Il codice presenta una prima parte in cui lo spazio delle chiavi viene suddiviso tre i diversi thread assegnando ad ognuno di essi un sottospazio

```
for (int i = 0; i < thread_idx; i++) {
    ci[i].start = size * i;
    ci[i].size = size;
    pthread_create(&tid[i], NULL, worker, (void *) &ci[i]);
}
```

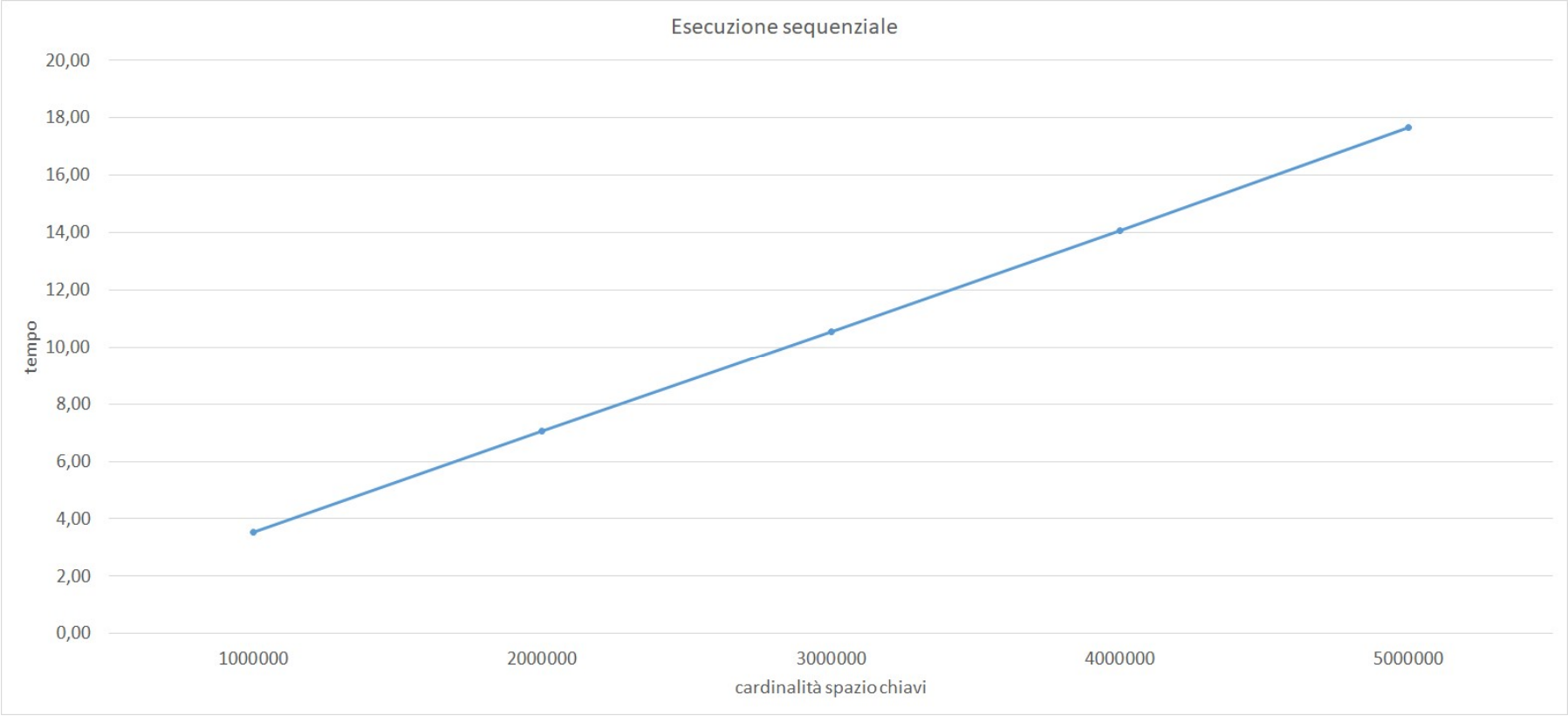
In ogni thread vengono generate e verificate le chiavi

```
while ((i < ci->size) && (strcmp(pwdes, testdes) != 0) && (! found)) {
        index[7]++;
        if (index[7] == 64) {
            index[7] = 0;
            test[7] = '.';
            index[6]++;
             …
```

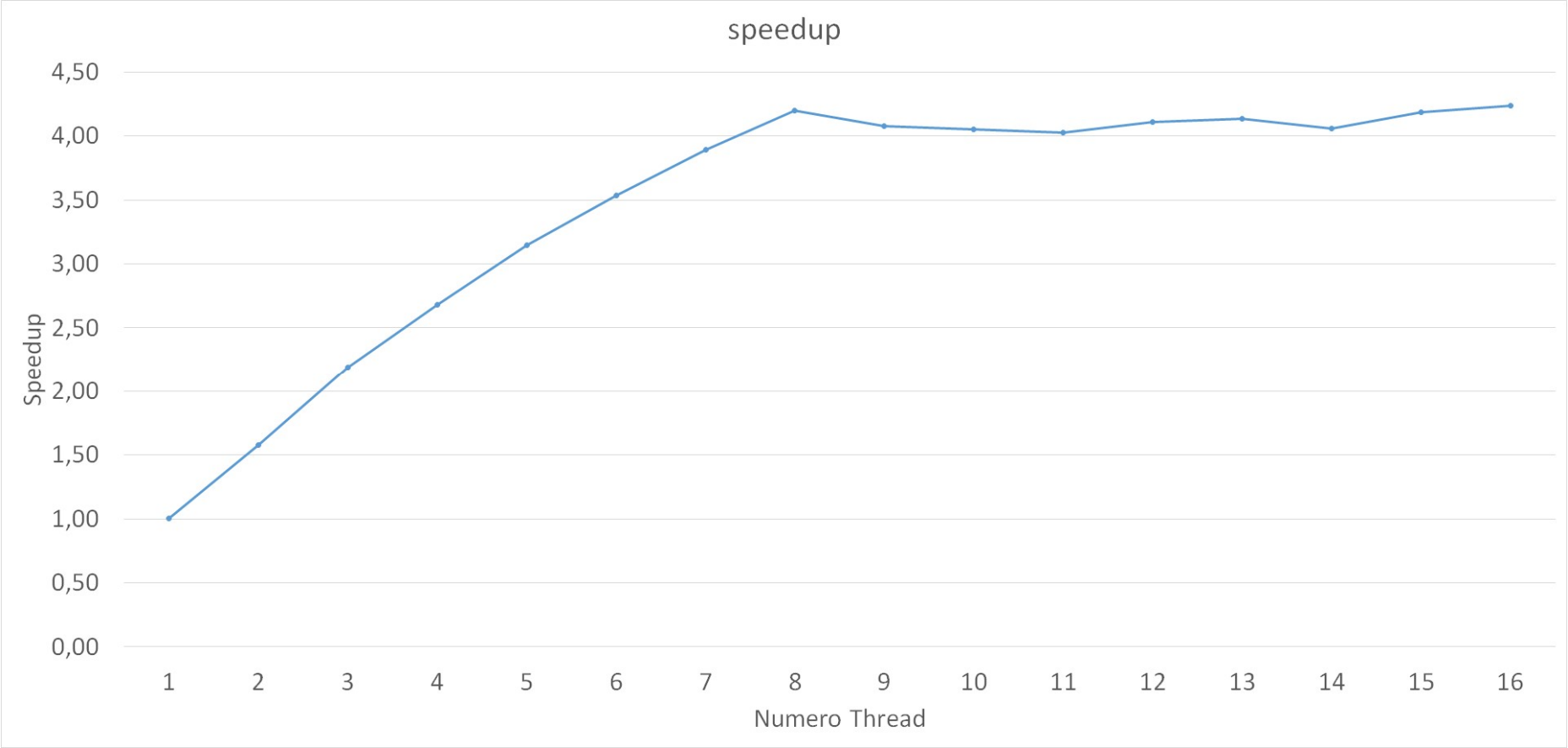
In ogni thread vengono generate e verificate le chiavi

```
strncpy(testdes, crypt(test, "00"),13);
if (strcmp(pwdes, testdes) == 0) {
    pthread_mutex_lock(&lock);
    found = 1;
    strcpy(found_pw, test);
    pthread_mutex_unlock(&lock);
    pthread_exit(NULL);
}
```

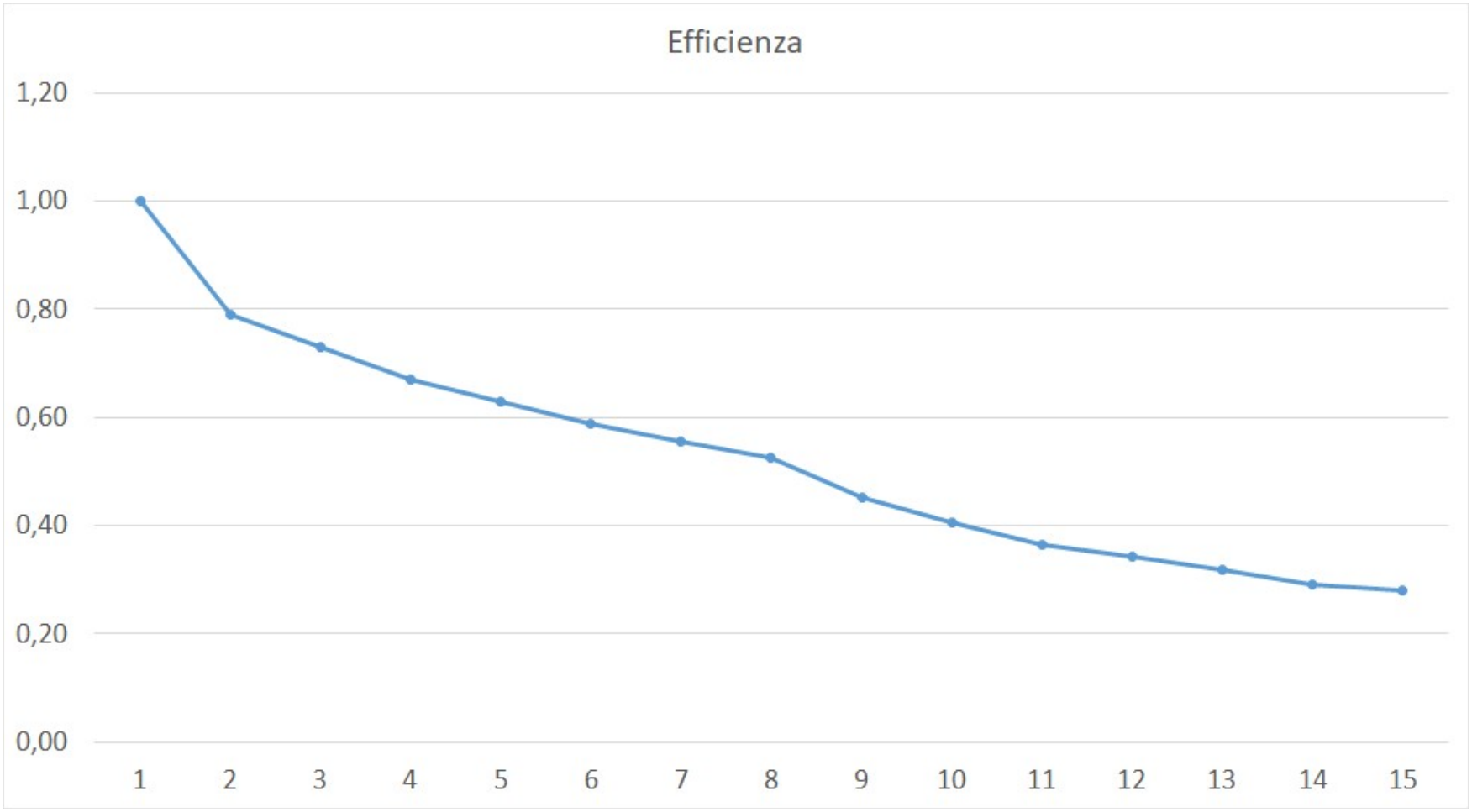
Il codice è molto stabile in termini di performance e il tempo di esecuzione in modalità sequenziale è direttamente proporzionale al numero di chiavi analizzate



Lo speedup ottenuto è sub lineare fino al raggiungimento degli 8 thread



Efficienza


Efficienza
1,20
1,00
0,80
0,60
0,40
0,20
0,00
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15