Calcolo Parallelo e Distribuito A.A. 2023/24

OpenMP: Esercitazione

Docente: Prof. L. Marcellino
Tutor: Prof. P. De Luca

Università di Napoli "Parthenope"

22 Aprile 2024

1. Parallel search
   - Parallel binary search

2. Parallel sorting
   - Odd-even algorithm

Parallel search

Ricerca dicotomica

Algoritmo di ricerca che individua un determinato elemento (*token*) in una sequenza ordinata di elementi.

Algoritmo:

- suddivisione del vettore in due parti calcolando l'indice (*mid*) dell'elemento mediano;
- se il valore *token* è maggiore rispetto alla componente in indice *mid*, la ricerca si ripete considerando la metà di sinistra del vettore;
- in caso contrario la ricerca procede analizzando la parte destra del vettore suddiviso.

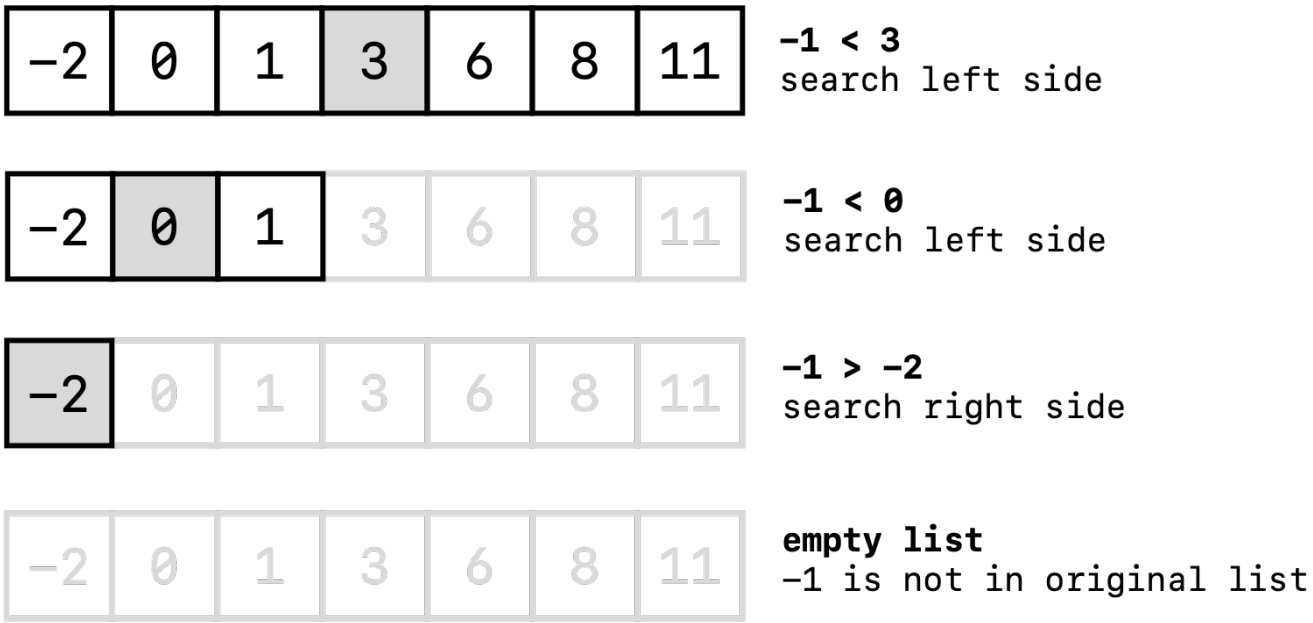
Algorithm 1 Pseudocodice Binary search

Input: A, l, r, token

1: **if** token $< A[l]$ or token $> A[r]$ **then**
2: return -1
3: **end if**
4: **while** $l \leq r$ **do**
5: mid := $\frac{(l+r)}{2}$
6: **if** $A[mid] == token$ **then**
7: return mid;
8: **end if**
9: **if** $A[mid] > token$ **then**
10: $r = mid - 1$;
11: **else**
12: $l = mid + 1$
13: **end if**
14: **end while**

Analisi della complessità computazionale

Ad ogni passo viene dimezzata la dimensione per cui abbiamo un limite asintotico $\mathcal{O}(log\ n)$.



Parallel search


```
1  int binary(int l,int r, int key,int n, int *a){
2  int index = -1;
3  int size=(r-l+1)/n;
4  if(size==0 || n==1){
5   #pragma omp parallel for
6   for(int i=l;i<=r;i++){
7    if(a[i]==key) index=i;
8   }
9  return index;
10
11 }
12 int left=l;
13 int right=r;
14 omp_set_num_threads(n);
15 omp_set_nested(1);
16 # pragma omp parallel
17 {
18  int id=omp_get_thread_num();
19  int lt=l+id*size;
20  int rt=lt+size-1;
21  if (id==n-1) rt=r;
22
23  if(a[lt]<=key && a[rt]>=key){
24   left=lt;
25   right=rt;
26  }
27 }
28 if (left==l && right ==r) return -1;
29 return binary(left,right,key,n, a);
30
31 }
```

Metriche ed ambiente di calcolo

Strong scalability

- La **strong scalability** misura lo speed-up per una dimensione fissa del problema rispetto al numero di processori ed è governato dalla legge di Amdahl;
- Input size fissa. Numero di threads incrementale.

Weak scalability

- La **weak scalability** misura lo speed-up per una dimensione del problema ridotta rispetto al numero di processori ed è regolato dalla legge di Gustafson;
- Input e numero di threads incrementali.

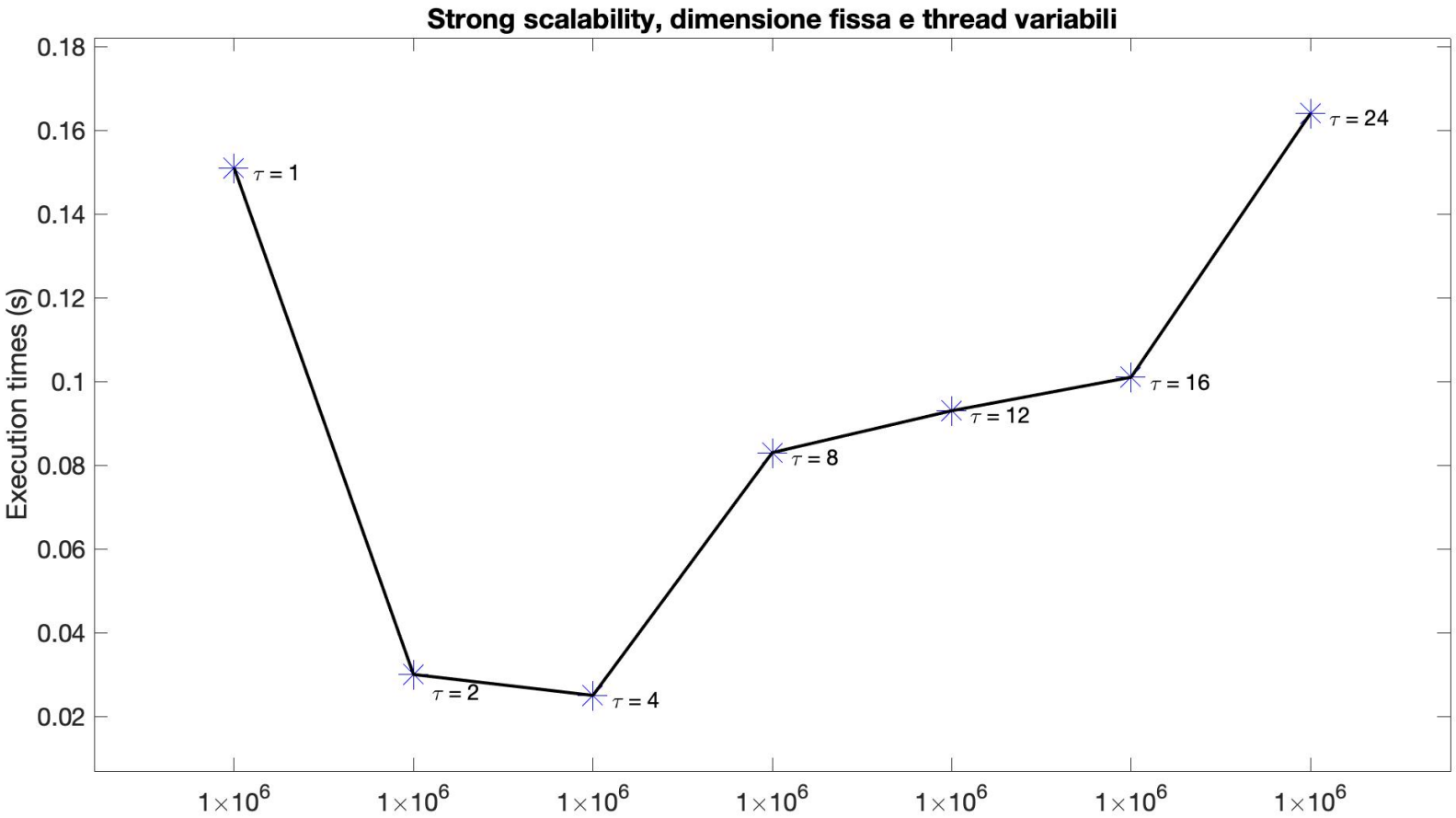
Caratteristiche tecniche:

- 2×16 IBM POWER9 AC922 @ 3.11 Ghz;
- 256 GB di RAM.

Analisi delle performance – Strong scalability

N	τ	Execution time (s)
1×10^6	1	0.151
1×10^6	2	0.030
1×10^6	4	0.025
1×10^6	8	0.083
1×10^6	12	0.093
1×10^6	16	0.101
1×10^6	24	0.162

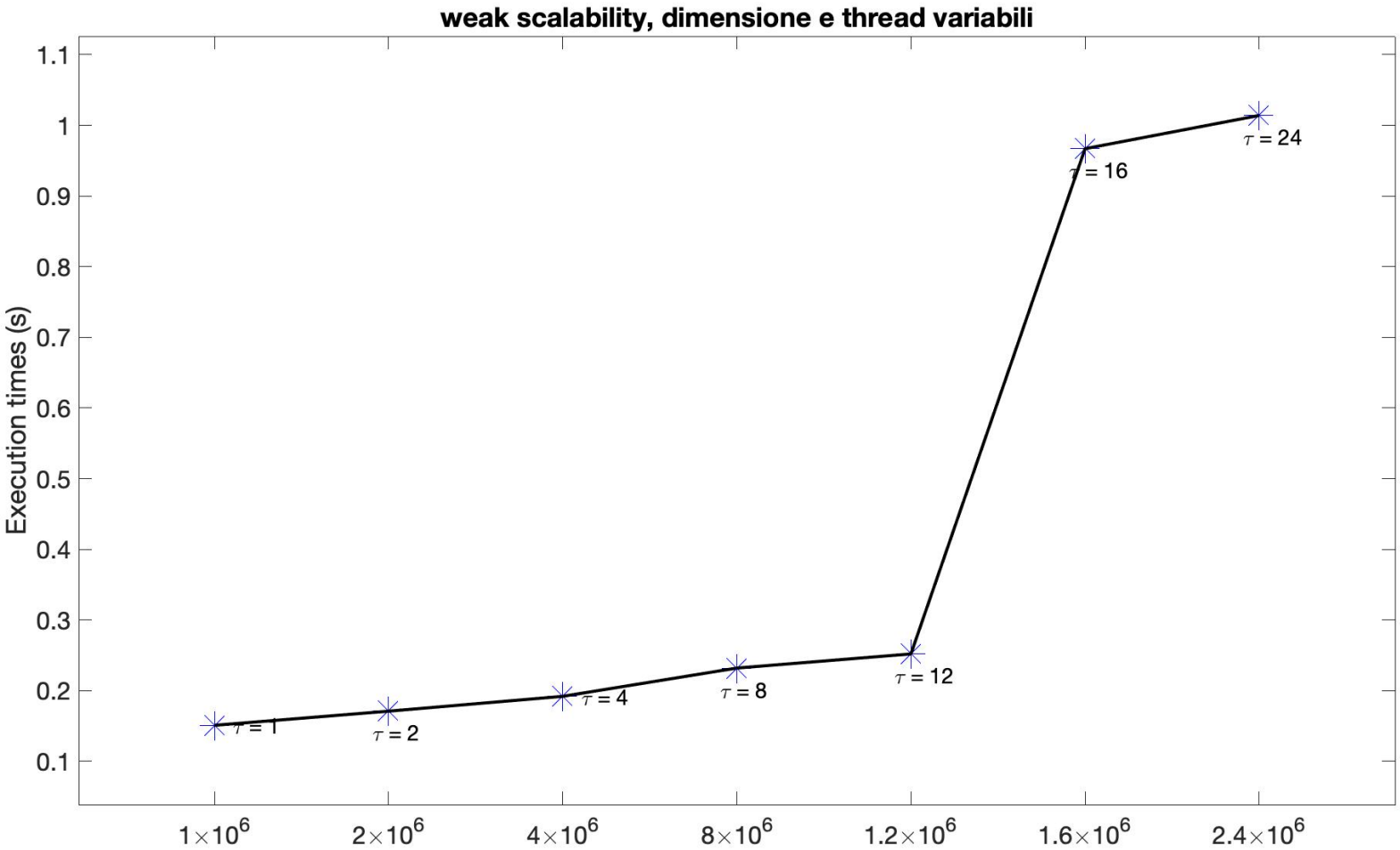
Analisi delle performance



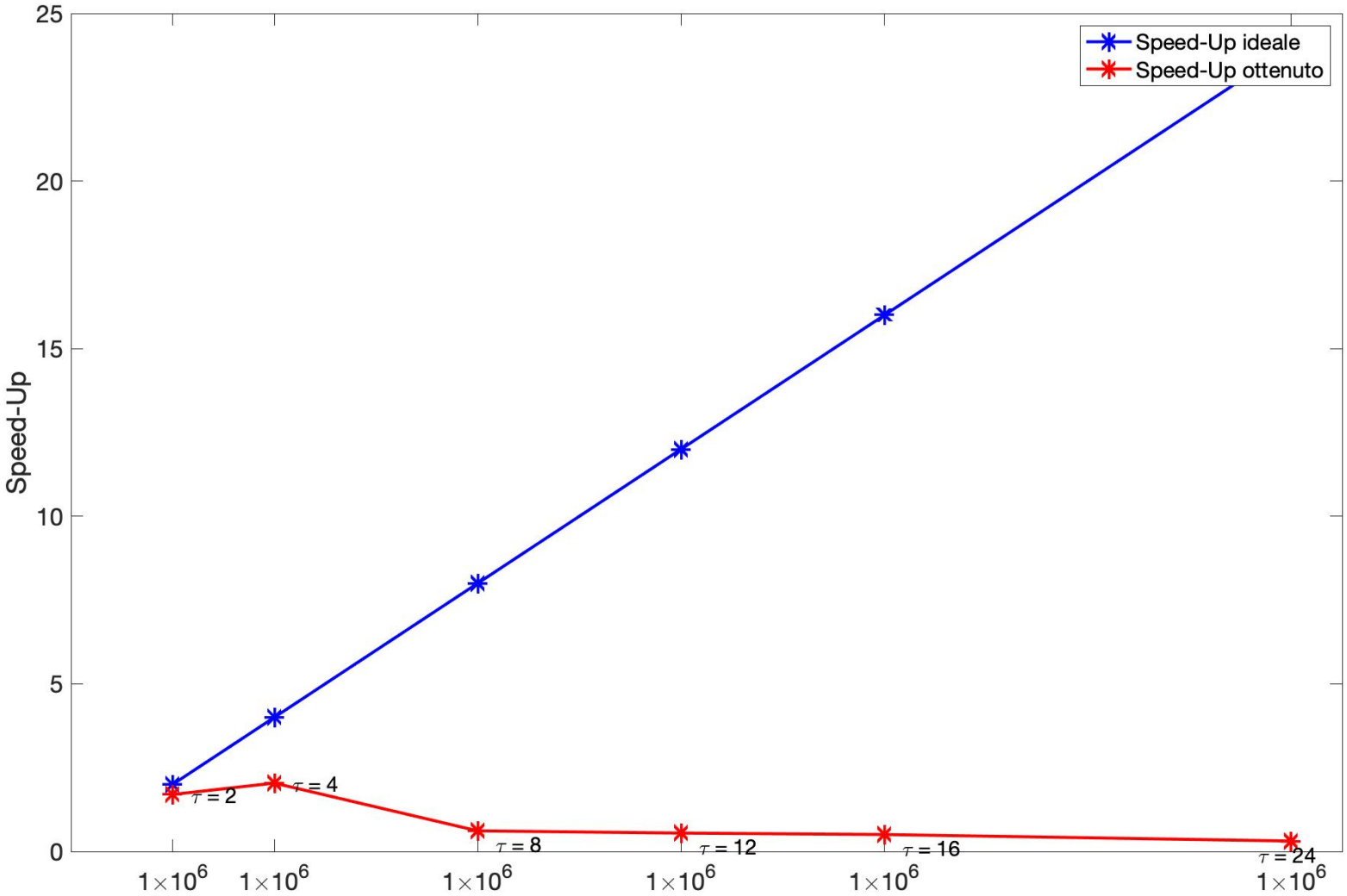
Analisi delle performance – Weak Scalability

N	τ	Execution time (s)
1×10^6	1	**0.151**
2×10^6	2	**0.171**
4×10^6	4	**0.192**
8×10^6	8	0.232
1.2×10^6	12	0.252
1.6×10^6	16	0.967
2.4×10^6	24	1.014

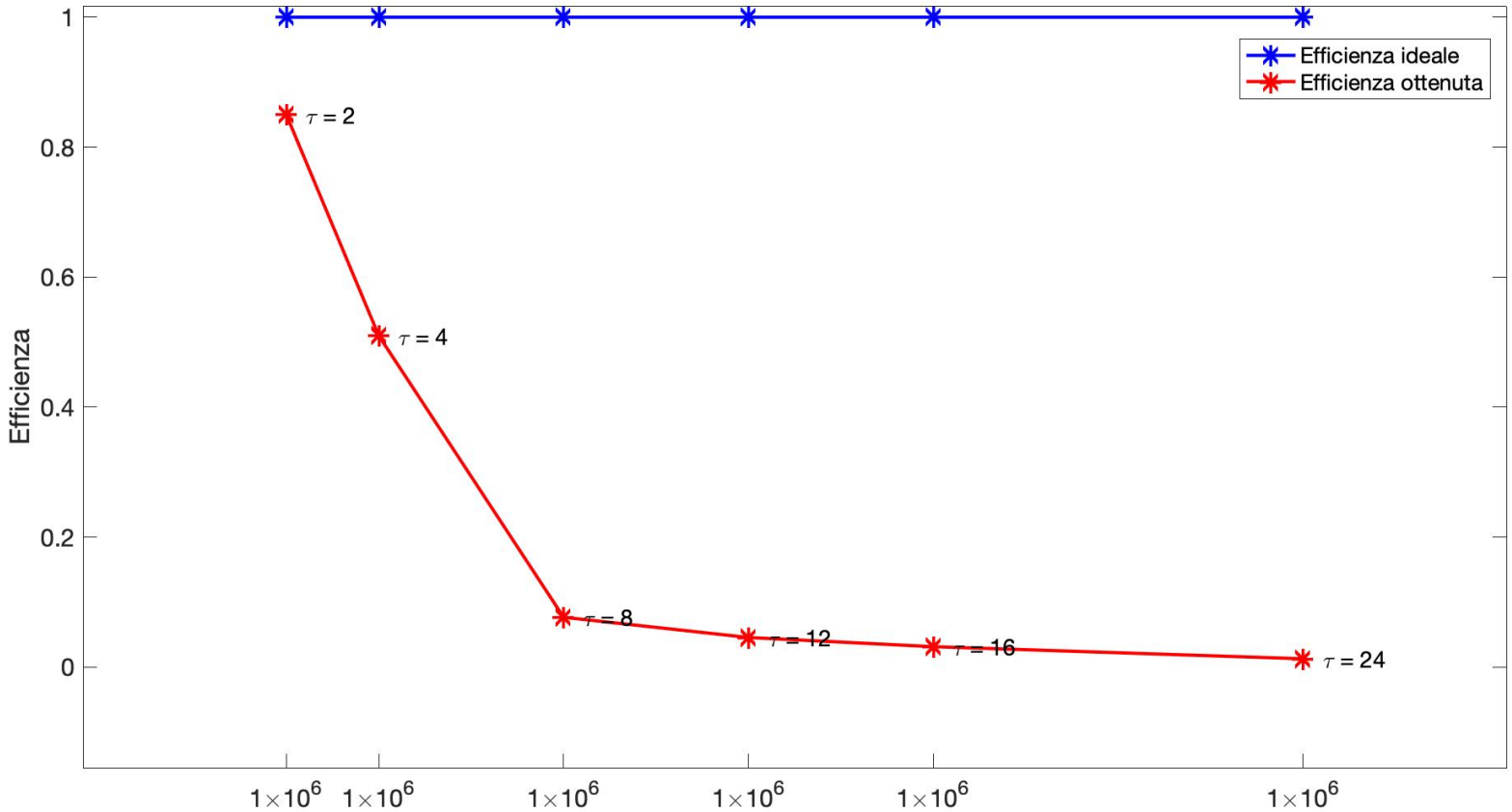
Analisi delle performance



Analisi delle performance – Speed Up



Analisi delle performance – Efficienza



Algoritmi di ordinamento

La caratterstica di base di un algoritmo di algoritmo è basata sul *confronto*. Molti confronti generano delle condizioni di **overhead**.

Algoritmo	Complessità di tempo		
	Migliore	Medio	Peggiore
Selection sort	$\Omega(n^2)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$
Bubble sort	$\Omega(n)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$
Insertion sort	$\Omega(n)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$
Heap sort	$\Omega(n\,log(n))$	$\Theta(n\,log(n))$	$\mathcal{O}(n\,log(n))$
Quick sort	$\Omega(n\,log(n))$	$\Theta(n\,log(n))$	$\mathcal{O}(n^2)$
Merge sort	$\Omega(n\,log(n))$	$\Theta(n\,log(n))$	$\mathcal{O}(n\,log(n))$

Odd-even sorting

È un algoritmo di ordinamento con pecularietà simili al *bubble sort*. Dal nome si capisce che opera su determinate proprietà dei numeri.

Algoritmo:

- Si confrono tutte le coppie pari e dispari degli elementi presenti in un vettore;
- se una coppia è nell'ordine sbagliato si scambia di posto i suoi elementi;
- l'algoritmo continua l'ordinamento finché tutte le coppie non sono ordinate.

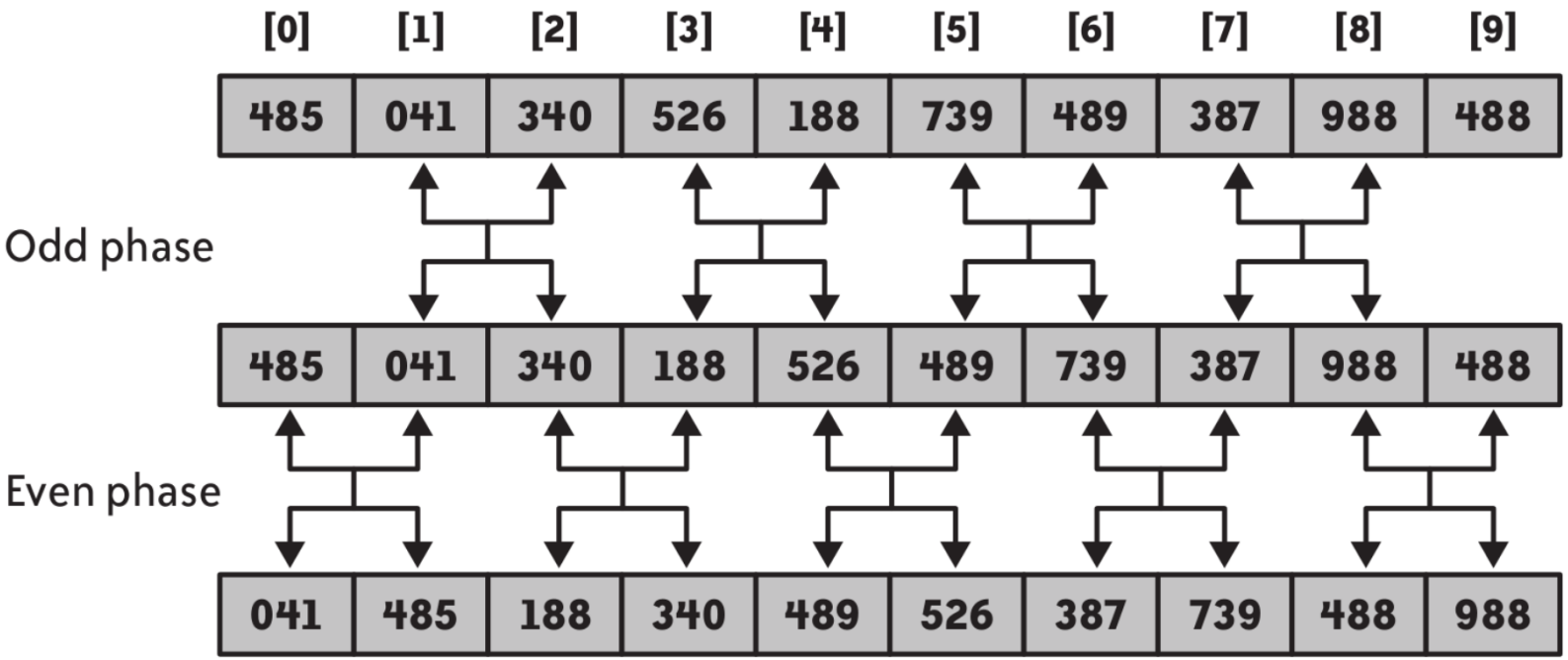
Algorithm 2 Pseudocodice Odd-even sorting

Input: list

```
1:  sorted := false
2:  while !sorted do
3:     sorted:= true;
4:     for i = 1; i < list.length()-1; i = i + 2  do
5:        if list[i]>list[i+1] then
6:           swap(list[i], list[i+1]);
7:           sorted := false;
8:        end if
9:     end for
10:    for i = 0; i < list.length()-1; i = i + 2  do
11:       if list[i]>list[i+1] then
12:          swap(list[i], list[i+1]);
13:          sorted := false;
14:       end if
15:    end for
16: end while
```

Analisi della complessità computazionale

L'elevato numero di confronti in funzione di n, strettamente legati alla pecularietà delle coppie di analizzare, la complessità di questo algoritmo è di $\mathcal{O}(n^2)$.



Parallel sorting


```
1  void oddEvenSort (int *a, int N)
2  {
3   int sw = 1, start = 0, i = 0;
4   int temp;
5
6   while (sw || start)
7   {
8      sw = 0;
9      for( i = start; i < N - 1; i += 2){
10         if(a[i]>a[i+1])
11         {
12             temp = a[i];
13             a[i] = a[i+1];
14             a[i+1] = temp;
15             sw = 1;
16         }
17     }
18     if (start == 0) start = 1;
19     else start = 0;
20  }
21 }
```

- Ogni confronto, all'interno di una fase, può essere eseguito in modo *parallelo*.

Parallel sorting


```
1  void oddEvenSort (int *a, int N)
2  {
3   int sw = 1, start = 0, i = 0;
4   int temp;
5
6   while (sw || start)
7   {
8      sw = 0;
9      for( i = start; i < N - 1; i += 2){
10         if(a[i]>a[i+1])
11         {
12             temp = a[i];
13             a[i] = a[i+1];
14             a[i+1] = temp;
15             sw = 1;
16         }
17     }
18     if (start == 0) start = 1;
19     else start = 0;
20  }
21 }
```

- Ogni confronto, all'interno di una fase, può essere eseguito in modo *parallelo*.

Parallel sorting

Decomposizione del problema:

- suddividendo il vettore in blocchi ed effettuando i confronti nel blocco;
- i confronti al bordo sono demandati al thread con indice più basso.

Punti chiave:

- Assenza di contesa: qualsiasi elemento, di entrambe le fasi, viene *toccato* una sola volta;
- barriera implicita: tutti i confronti sono completati prima di iniziare la fase succeessiva.

Parallel sorting – Codice parallelo 1/4


```
1   void oddEvenSort (int *a, int N)
2   {
3    int sw = 1, start = 0, i = 0;
4    int temp;
5
6    while (sw || start)
7    {
8       sw = 0;
9       #pragma omp parallel for private(temp)
10      for( i = start; i < N - 1; i += 2){
11          if(a[i]>a[i+1])
12          {
13              temp = a[i];
14              a[i] = a[i+1];
15              a[i+1] = temp;
16              sw = 1;
17          }
18      }
19      if (start == 0) start = 1;
20      else start = 0;
21   }
22  }
```

Parallel sorting – Codice parallelo 2/4

Analisi delle risorse

- **temp**, privata per ogni thread;
- **start**, non necessita di protezione poiché viene aggiornata fuori la regione parallela;
- **sw** è aggiornata nel ciclo *for* e poi viene letta fuori.

- Se volessimo contare il numero di scambi abbiamo bisogno di **sincronizzazione**.

Parallel sorting – Codice parallelo 2/4

Analisi delle risorse

- **temp**, privata per ogni thread;
- **start**, non necessita di protezione poiché viene aggiornata fuori la regione parallela;
- **sw** è aggiornata nel ciclo *for* e poi viene letta fuori.

- Se volessimo contare il numero di scambi abbiamo bisogno di **sincronizzazione**.

Parallel sorting – Codice parallelo 3/4


```
1  void oddEvenSort (int *a, int N)
2  {
3   int sw0, sw1 = 1, start = 0, i = 0, t = 0;
4
5   while (sw)
6   {
7      sw0 = 0;
8      sw1 = 0;
9      #pragma omp parallel
10     {
11      int temp;
12      #pragma omp for
13      for( i = 0; i < N - 1; i += 2){
14         if(a[i]>a[i+1])
15         {
16              temp = a[i];
17              a[i] = a[i+1];
18              a[i+1] = temp;
19              sw0 = 1;
20         }
21     }
22     if (sw0 || !t){
23     #pragma omp for
24     {
25     for( i = 1; i < N - 1; i += 2){
26         if(a[i]>a[i+1])
27         {
28              temp = a[i]; a[i] = a[i+1];
29              a[i+1] = temp;
30              sw1 = 1;
31         }
32     }
33     }
34     t = 1;
35   }
36  }
```

Parallel sorting – Codice parallelo 4/4

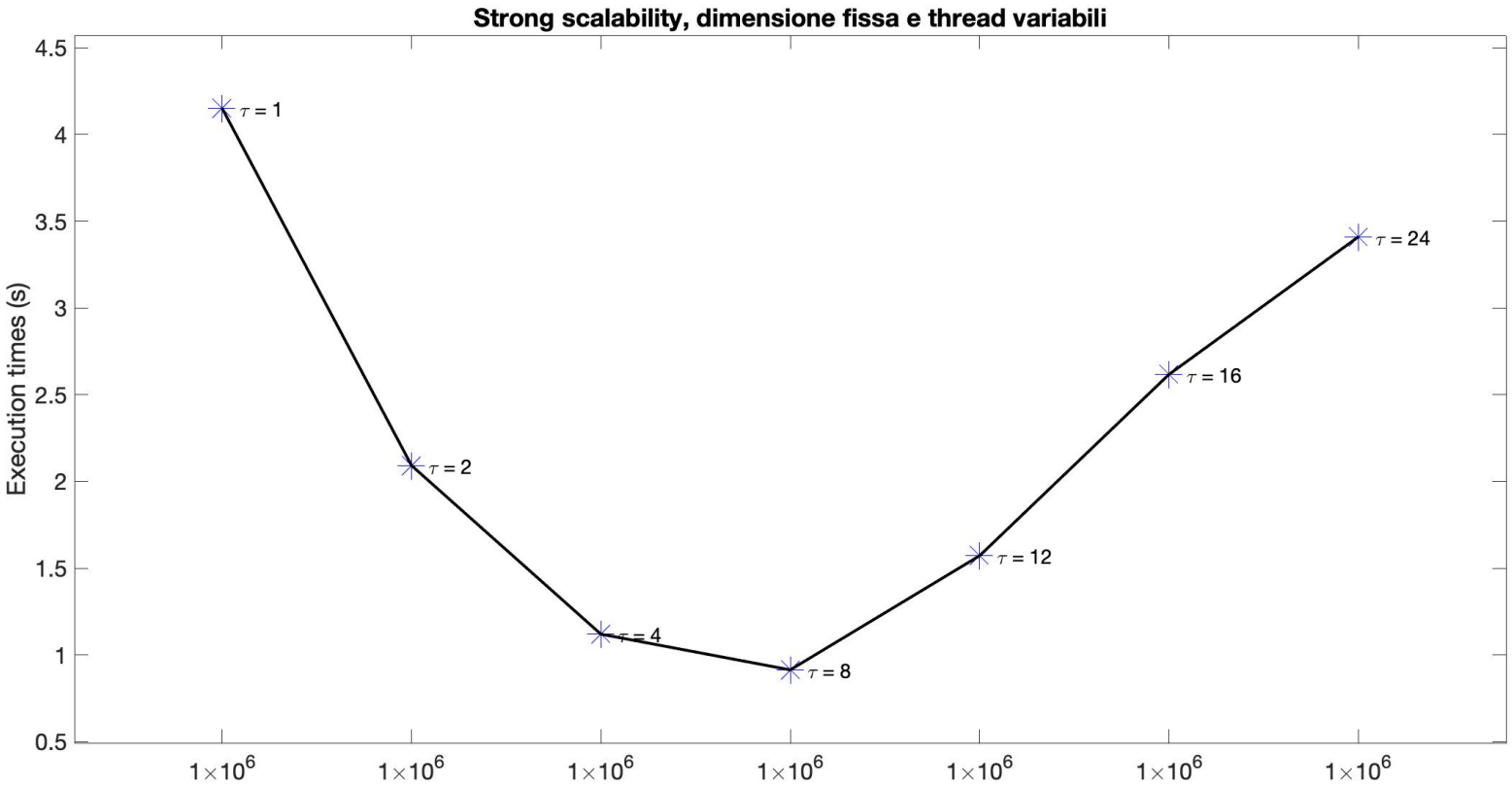
Algoritmo:

- utilizza due variabili per gli scambi (no contesa);
- un thread entra in regione parallela solo quando queste due variabili sono resettate;
- il primo ciclo *for* suddivide il vettore. Se qualche thread esegue uno scambio, imposta *sw0*;
- ogni thread imposta una barriera implicita testando *t* e *sw0*;
- se qualche thread ha effettuato uno scambio nel secondo *for*, allora *sw1* viene impostata;
- Ricontrollare *sw1* e continua.

Analisi delle performance – Strong scalability

N	τ	Execution time (s)
1×10^6	1	4.151
1×10^6	2	2.092
1×10^6	4	1.120
1×10^6	8	0.914
1×10^6	12	1.572
1×10^6	16	2.617
1×10^6	24	3.410

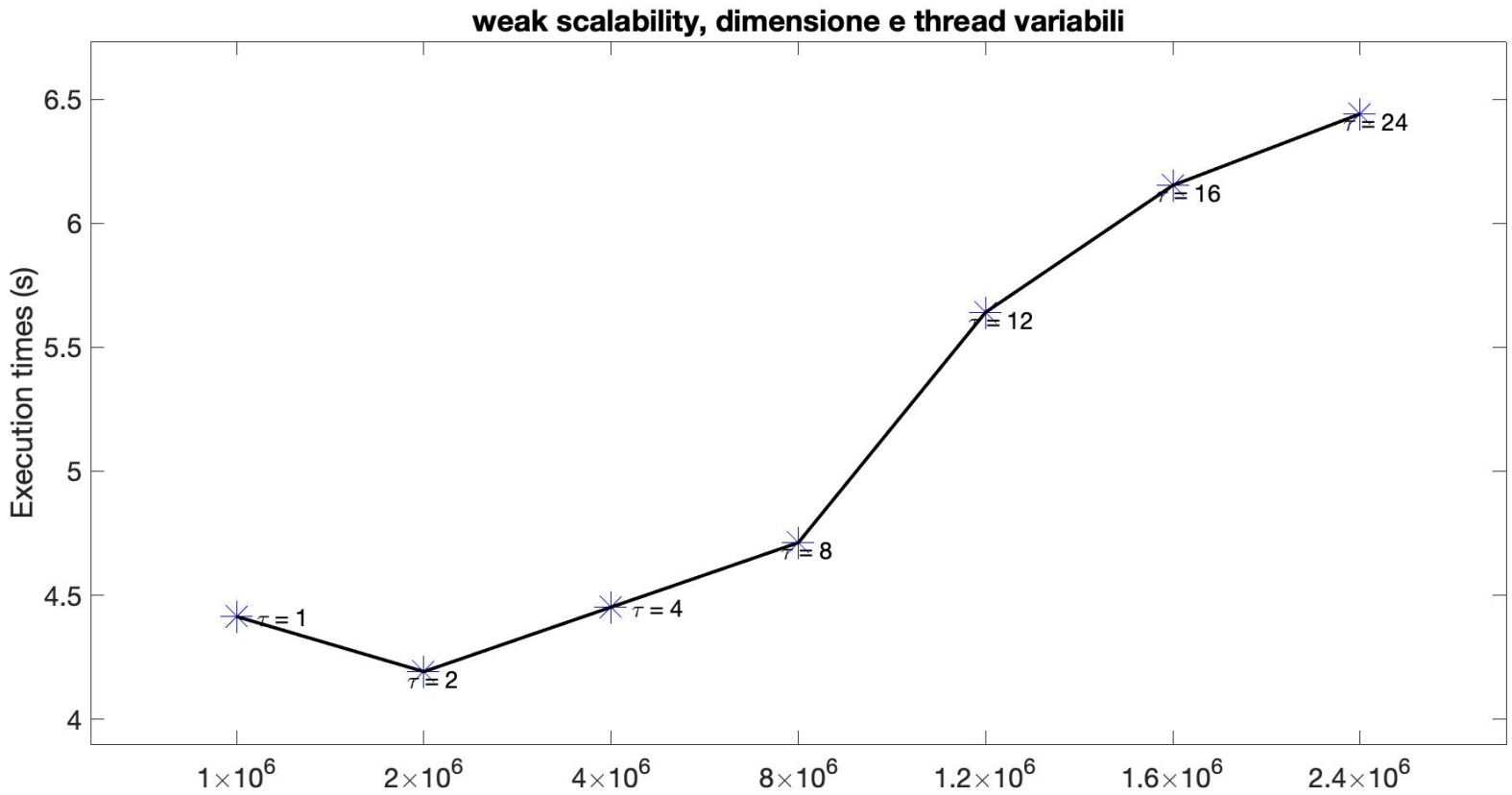
Analisi delle performance



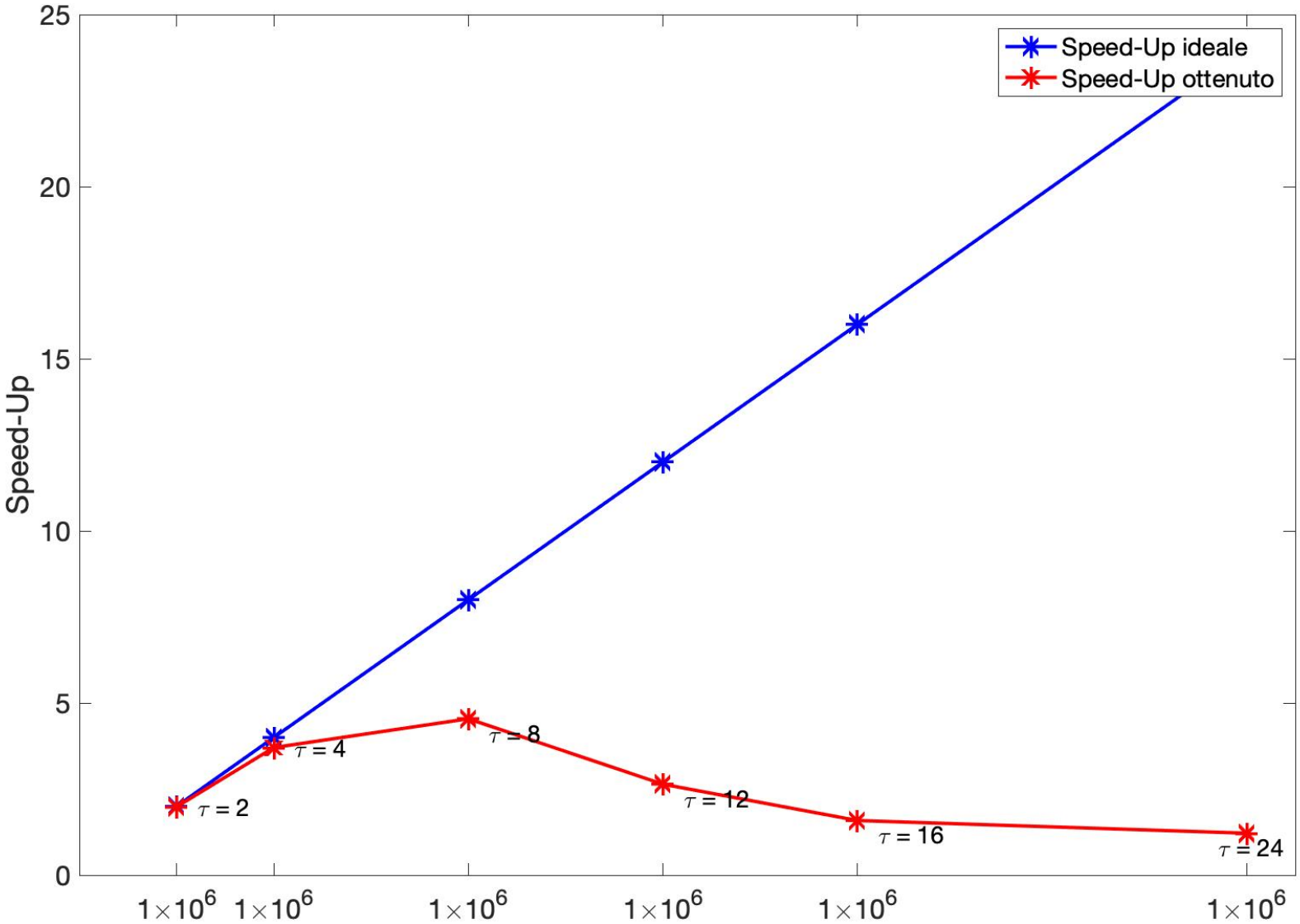
Analisi delle performance – Weak Scalability

N	τ	Execution time (s)
1×10^6	1	4.141
2×10^6	2	4.192
4×10^6	4	4.452
8×10^6	8	4.712
1.2×10^6	12	5.641
1.6×10^6	16	6.153
2.4×10^6	24	6.441

Analisi delle performance



Analisi delle performance – Speed Up



Analisi delle performance – Efficienza