Calcolo Parallelo e Distribuito A.A. 2023/24

OpenMP: sAXPY

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

Università di Napoli "Parthenope"

27 Maggio 2024

Problema:

Dati $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^n$, $a \in \mathbb{R}^m$ definiamo:

$$r = \alpha A \cdot b + \beta \cdot a;$$

dove $\alpha, \beta \in \mathbb{N}$.
L'output è definito:

$$\prod_{i=0}^{m-1} r_i$$

- Implementare un programma in linguaggio C/C++ per l'ambiente parallelo MIMD-SM con l'ausilio della libreria OpenMP per risolvere il problema proposto.

Problema:

Dati $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^n$, $a \in \mathbb{R}^m$ definiamo:

$$r = \alpha A \cdot b + \beta \cdot a;$$

dove $\alpha, \beta \in \mathbb{N}$.
L'output è definito:

$$\prod_{i=0}^{m-1} r_i$$

- Implementare un programma in linguaggio C/C++ per l'ambiente parallelo MIMD-SM con l'ausilio della libreria OpenMP per risolvere il problema proposto.

Algoritmo:

- l'ordine delle operazioni è indifferente, fermo restando che le operazioni di **prodotto** abbiano priorità rispetto a quelle di **somma**;
- dimensioni della matrice, scalari e numero di threads devono essere parametri di input;
- La matrice A e i vettori a, b devono essere generati, in modalità random, all'interno del codice, mentre gli scalari α e β devono essere dati di input;
- è necessario un controllo a priori sulla corretta divisibilità delle dimensioni per il numero di threaeds:

$$n\_loc_i = \frac{m}{\tau} \quad i = 0, \ldots, \tau - 1;$$

dove τ è il numero totale di threads.

Codice parallelo 1/2


```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  int main(int argc, char* argv[])
6  {
7    int i,j,N,M,alfa,beta,t;
8      double *A,*b,*a,*R,p=0.00f, t0, t1, tempotot;
9
10         if(argc>1)
11         {
12             t = atoi(argv[1]);
13            M = atoi(argv[2]);
14            N = atoi(argv[3]);
15             alfa = atoi(argv[4]);
16             beta = atoi(argv[5]);
17         }
18         else
19         {
20           printf("Error usage: ./exec <nThreads> <M> <N> <alpha> <beta>\n");
21           exit(EXIT_FAILURE);
22         }
23
24         if((M%2)!=0) exit(EXIT_FAILURE); //controllo divisibilità
25       //Allocazioni dinamiche
26         A = (double *)malloc((M*N)*sizeof(double ));
27         b = (double *)malloc(N*sizeof(double));
28         a = (double *)malloc(M*sizeof(double));
29         R = (double *)malloc(M*sizeof(double));
30
31         //Generazione di valori pseudo-casuali
32       for (i=0;i<M;i++)
33       {
34           a[i]=(double)((rand()%1000)+1)/1000;
35             for(j=0;j<N;j++)
36                   A[(i*N)+j]=(double)((rand()%1000)+1)/1000;
37       }
```

Codice parallelo 2/2


```
1   for(i=0;i<N;i++)
2           b[i]=(double)((rand()%1000)+1)/1000;
3      t0=omp_get_wtime();
4   #pragma omp parallel for schedule(static) shared(N,M,A,b,a,R,alfa,beta) private(i,j) num_threads(t)
5   for(i=0;i<M;i++)
6   {
7     for(j=0;j<N;j++)
8     {
9       R[i]+=alfa*A[(i*N)+j]*b[j];
10    }
11    R[i]+=(a[i]*beta);
12  }
13  #pragma omp parallel for shared(R,M,a) private(i) reduction(*: p) num_threads(t)
14    for(i=0;i<M;i++) p*=R[i];
15  t1=omp_get_wtime();
16  tempotot=t1-t0;
17
18  printf("Elapsed Time: %lfs \n", tempotot);
19  free(A);
20  free(b);
21  free(a);
22  free(R);
23
24  exit(EXIT_SUCCESS);
25 }
```

Sincronizzazione

Se volessimo un contatore relativo al numero totale di operazioni eseguite?

Idea:

- definire un contatore *ops*;
- impostare un accesso **atomico** per l'operazione di incremento.

Direttive

- critical: overhead elevato – lock a livello user;
- atomic: overhead ridotto – lock a livello hardware.

Entrambe assicurano un accesso in **mutua esclusione**, ma una risulta essere più lenta dell'altra.

Vediamo degli esempi.

Dettagli

Sincronizzazione

Se volessimo un contatore relativo al numero totale di operazioni eseguite?
Idea:

- definire un contatore *ops*;
- impostare un accesso **atomico** per l'operazione di incremento.

Direttive

- critical: overhead elevato – lock a livello user;
- atomic: overhead ridotto – lock a livello hardware.

Entrambe assicurano un accesso in **mutua esclusione**, ma una risulta essere più lenta dell'altra.

Vediamo degli esempi.

Sincronizzazione

Se volessimo un contatore relativo al numero totale di operazioni eseguite?
Idea:

- definire un contatore *ops*;
- impostare un accesso **atomico** per l'operazione di incremento.

Direttive

- critical: overhead elevato – lock a livello user;
- atomic: overhead ridotto – lock a livello hardware.

Entrambe assicurano un accesso in **mutua esclusione**, ma una risulta essere più lenta dell'altra.

Vediamo degli esempi.

Sincronizzazione

Se volessimo un contatore relativo al numero totale di operazioni eseguite?
Idea:

- definire un contatore *ops*;
- impostare un accesso **atomico** per l'operazione di incremento.

Direttive

- critical: overhead elevato – lock a livello user;
- atomic: overhead ridotto – lock a livello hardware.

Entrambe assicurano un accesso in **mutua esclusione**, ma una risulta essere più lenta dell'altra.

Vediamo degli esempi.

Codice parallelo sync – 1/2


```
1   int ops = 0;
2  for(i=0;i<N;i++)
3          b[i]=(double)((rand()%1000)+1)/1000;
4    t0=omp_get_wtime();
5  #pragma omp parallel for schedule(static) shared(N,M,A,b,a,R,alfa,beta,ops) private(i,j) num_threads(t)
6  for(i=0;i<M;i++)
7  {
8    for(j=0;j<N;j++)
9    {
10     R[i]+=alfa*A[(i*N)+j]*b[j];
11     #pragma omp critical
12     {
13         ops = ops + 1;
14     }
15   }
16   R[i]+=(a[i]*beta);
17 }
18 #pragma omp parallel for shared(R,M,a) private(i) reduction(*: p) num_threads(t)
19   for(i=0;i<M;i++) p*=R[i];
20 t1=omp_get_wtime();
21 tempotot=t1-t0;
22
23 printf("Elapsed Time: %lfs \nOps= %d\n", tempotot, ops);
24 free(A);
25 free(b);
26 free(a);
27 free(R);
28
29 exit(EXIT_SUCCESS);
30 }
```

Codice parallelo sync – 2/2


```
1   int ops = 0;
2  for(i=0;i<N;i++)
3          b[i]=(double)((rand()%1000)+1)/1000;
4    t0=omp_get_wtime();
5  #pragma omp parallel for schedule(static) shared(N,M,A,b,a,R,alfa,beta,ops) private(i,j) num_threads(t)
6  for(i=0;i<M;i++)
7  {
8    for(j=0;j<N;j++)
9    {
10     R[i]+=alfa*A[(i*N)+j]*b[j];
11     #pragma omp atomic update
12     {
13         ops = ops + 1;
14     }
15   }
16   R[i]+=(a[i]*beta);
17 }
18 #pragma omp parallel for shared(R,M,a) private(i) reduction(*: p) num_threads(t)
19   for(i=0;i<M;i++) p*=R[i];
20 t1=omp_get_wtime();
21 tempotot=t1-t0;
22
23 printf("Elapsed Time: %lfs \nOps = %d\n", tempotot, ops);
24 free(A);
25 free(b);
26 free(a);
27 free(R);
28
29 exit(EXIT_SUCCESS);
30 }
```

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.

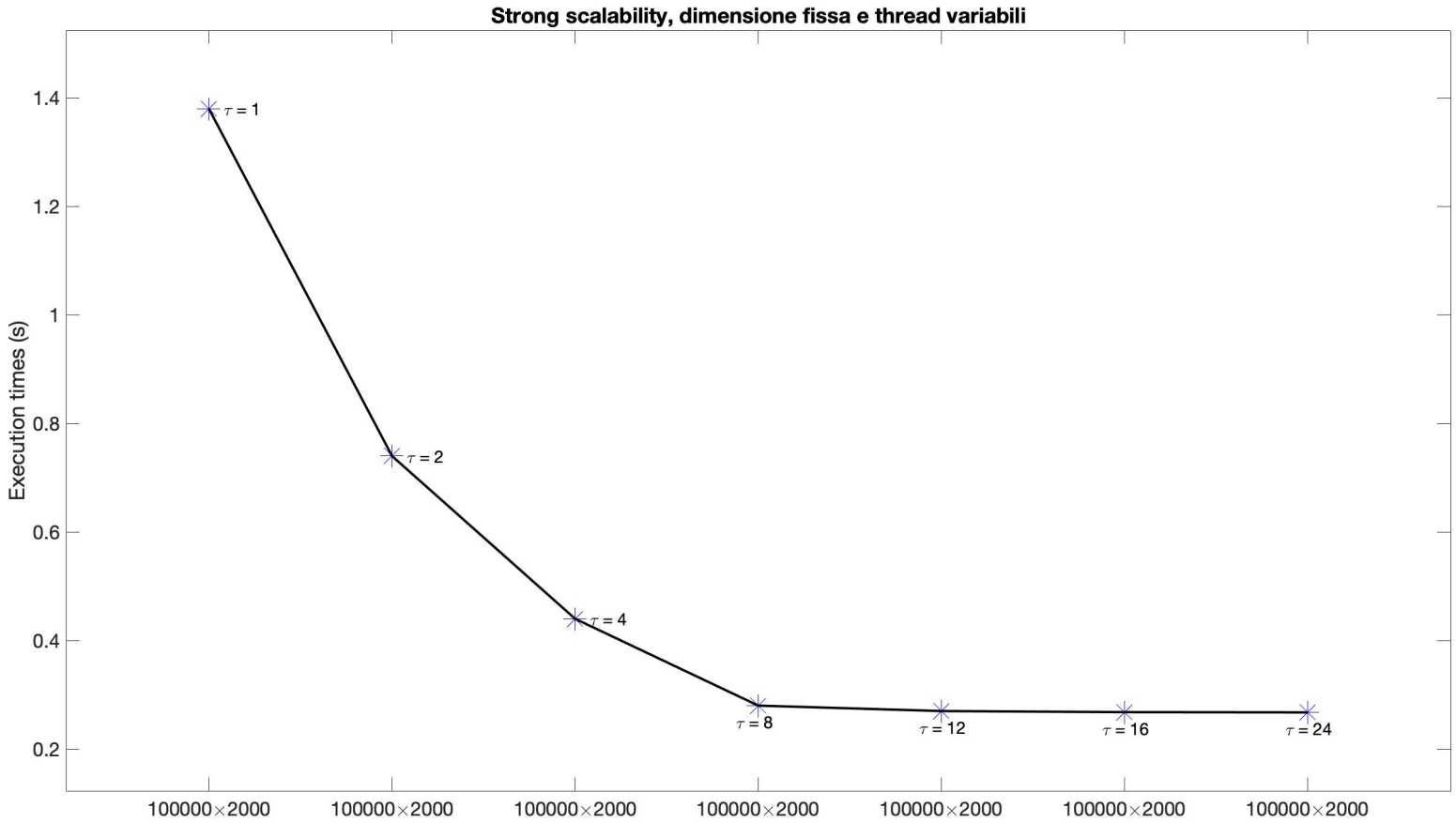
Caratteristiche tecniche:

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

Analisi delle performance

m	n	τ	Execution Time (s)
1×10^5	2×10^3	1	1.39
1×10^5	2×10^3	2	0.74
1×10^5	2×10^3	4	0.44
1×10^5	2×10^3	8	0.28
1×10^5	2×10^3	12	0.27
1×10^5	2×10^3	16	0.27
1×10^5	2×10^3	24	0.27

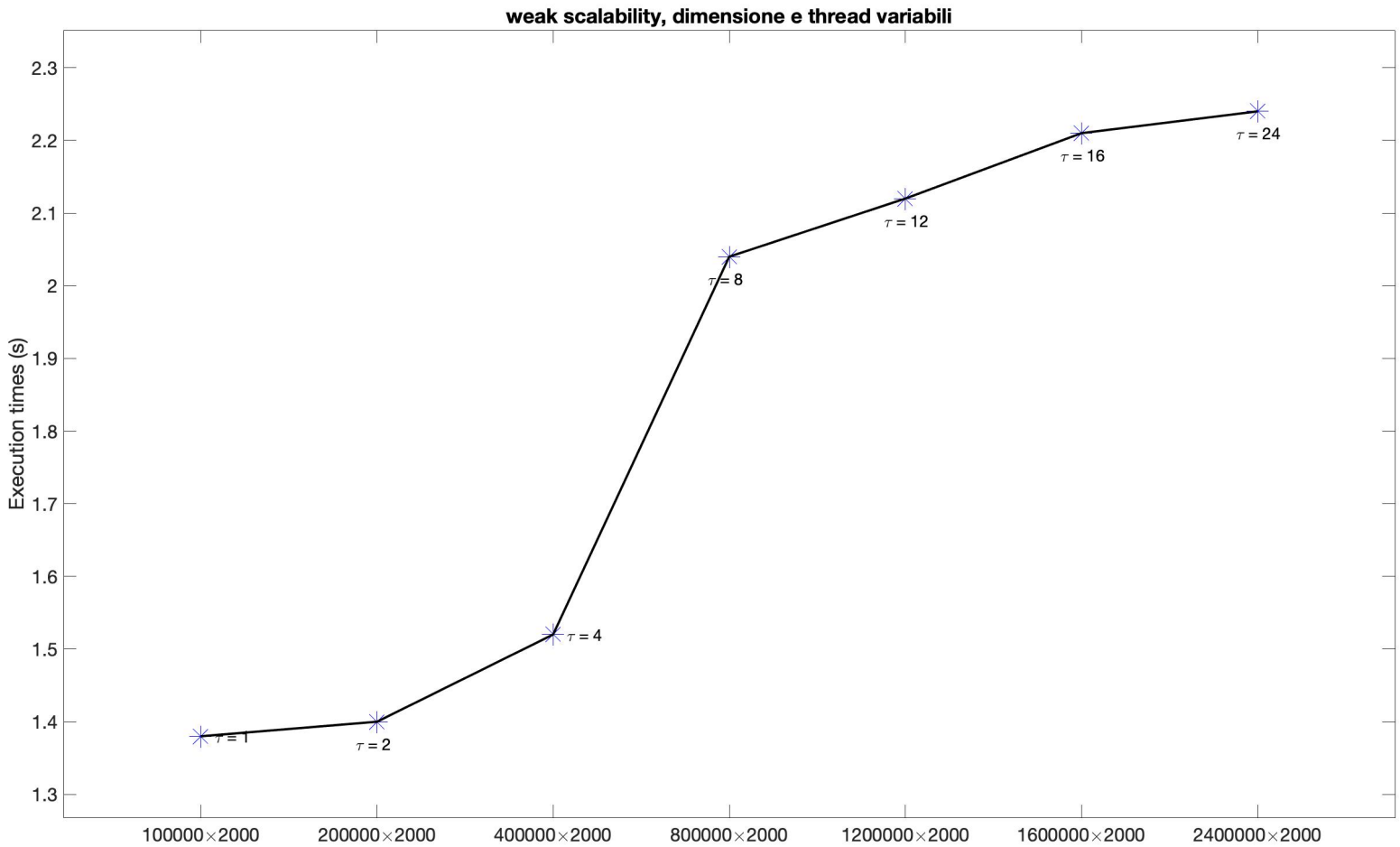
Analisi delle performance



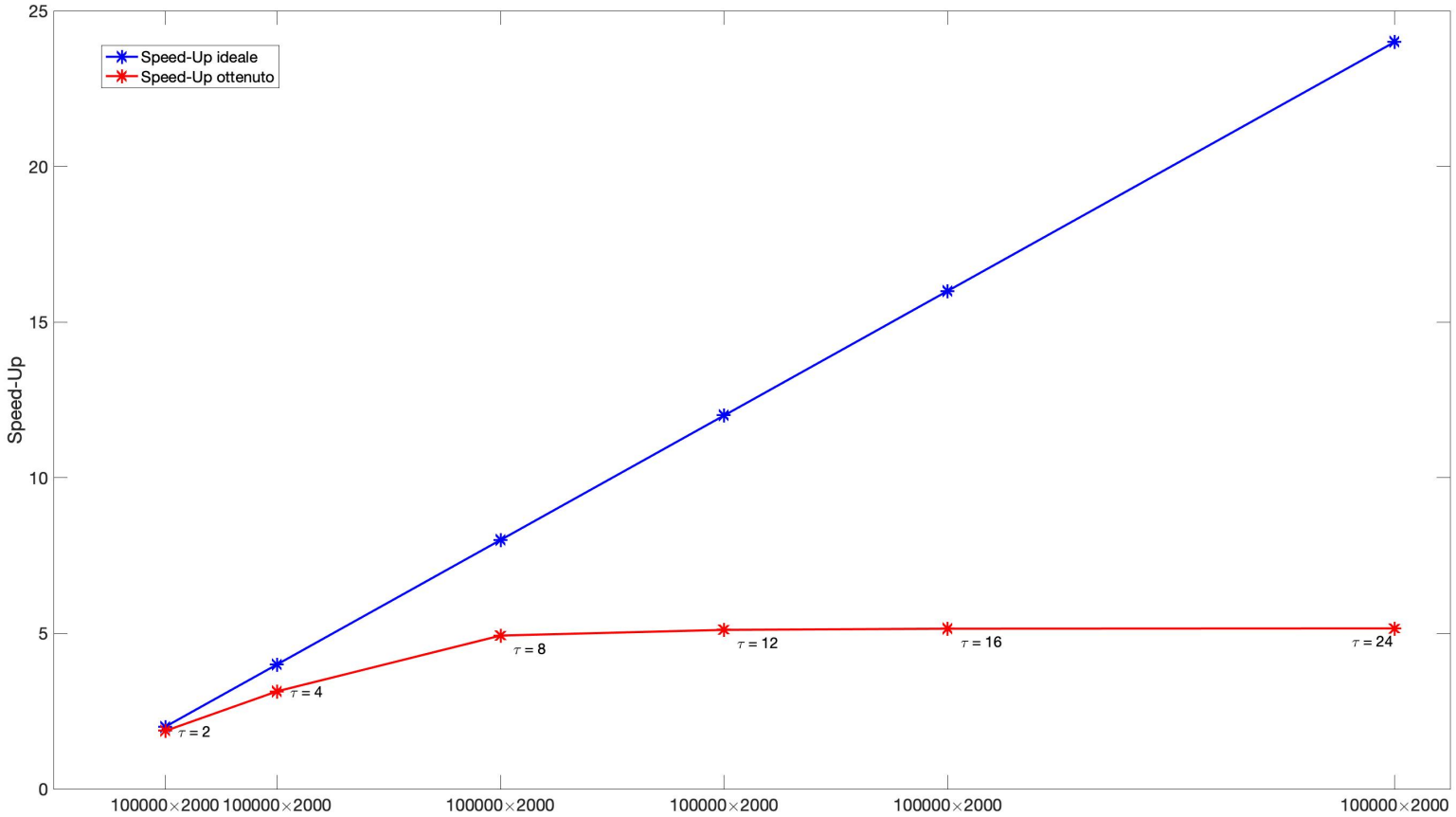
Analisi delle performance

m	n	τ	Execution Time (s)
1×10^5	2×10^3	1	1.39
2×10^5	2×10^3	2	1.40
4×10^5	2×10^3	4	1.51
8×10^5	2×10^3	8	2.02
12×10^5	2×10^3	12	2.10
16×10^5	2×10^3	16	2.21
24×10^5	2×10^3	24	2.24

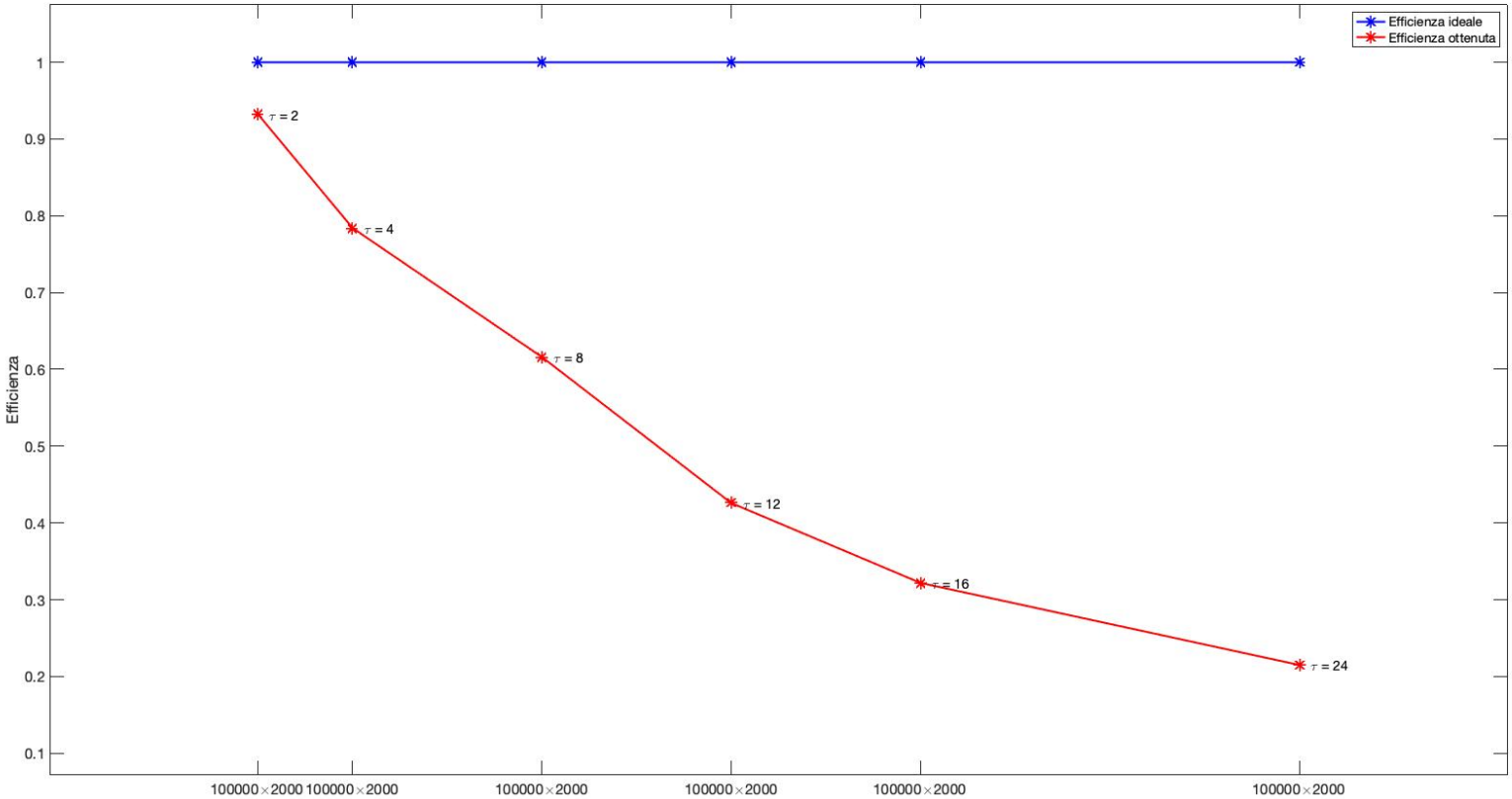
Analisi delle performance



Speed-up


Speed-Up ideale
Speed-Up ottenuto
Speed-Up
25
20
15
10
5
0
τ = 2
τ = 4
τ = 8
τ = 12
τ = 16
τ = 24
100000×2000
100000×2000
100000×2000
100000×2000
100000×2000
100000×2000

Efficienza


Efficienza ideale
Efficienza ottenuta
Efficienza
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
τ = 2
τ = 4
τ = 8
τ = 12
τ = 16
τ = 24
100000×2000
100000×2000
100000×2000
100000×2000
100000×2000
100000×2000

Atomic vs. Critical

m	n	τ	Critical Ex. Time (s)	Atomic Ex. Time (s)
1×10^5	2×10^3	1	7.06	3.28
1×10^5	2×10^3	2	21.56	10.16
1×10^5	2×10^3	4	38.92	11.47
1×10^5	2×10^3	8	71.99	13.17
1×10^5	2×10^3	12	74.96	12.59
1×10^5	2×10^3	16	76.04	12.43
1×10^5	2×10^3	24	70.93	7.16