Stima di π attraverso il Metodo Monte Carlo

Riccardo Nigrelli - Matr. 846986

17 ottobre 2019

Abstract

All'interno di questa relazione si andrà a confrontare rispetto ad alcune caratteristiche quali **Execution Time, Speedup, Efficiency** l'implementazione seriale con quella parallela della stima di π calcolata utilizzando il *Metodo Monte Carlo*.

Introduzione

I *Metodi Monte Carlo* sono una vasta classe di algoritmi computazionali che si basano su un campionamento casuale ripetuto per ottenere risultati numerici. Sono per lo più usati in problemi di natura fisica o matematica e vengono adottati quando mancano le condizioni per poter applicare metodi esatti. Essi sono principalmente utilizzati in due classi di problemi: ottimizzazione e integrazione numerica.

Ma come viene sfruttata dai Metodi Monte Carlo la ripetizione di numerosi esperimenti casuali?

Secondo la legge dei grandi numeri, gli integrali descritti dal valore atteso di alcune variabili casuali possono essere approssimati prendendo la media campionaria di campioni indipendenti della variabile.
Quindi tanto maggiore è il numero degli esperimenti che compio, tanto migliore sarà la stima fornitami dai *Metodi Monte Carlo*.
Quando la distribuzione di probabilità della variabile è parametrizzata, spesso viene usato un campionatore *Markov Chain Monte Carlo*.

I metodi Monte Carlo sono diversi tra loro ma seguono tutti lo stesso schema generale:

1. definire un dominio di possibili input;
2. generare in modo casuale degli input da una distribuzione di probabilità sul dominio;
3. eseguire un calcolo deterministico sugli input;
4. aggregare i risultati.

Implementazione

Per stimare il valore di π attraverso i *Metodi Monte Carlo* ho deciso di utilizzare il seguente metodo:

- si consideri una circonferenza di raggio unitario inscritta all'interno di un quadrato;
- successivamente si generi un numero definito di punti all'interno del quadrato;
- si conti il numero di punti contenuti nella circonferenza, per cui i punti con una distanza minore o al più pari a 1;
- il rapporto tra il numero di punti nella circonferenza e il numero totale di punti generati è una stima del rapporto delle due aree, pari a $\pi/4$;
- moltiplicando quindi il rapporto ottenuto in precedenza per 4 si ottiene la stima di π.

In figura 1 è possibile vedere una rappresentazione grafica del problema definito in precedenza.

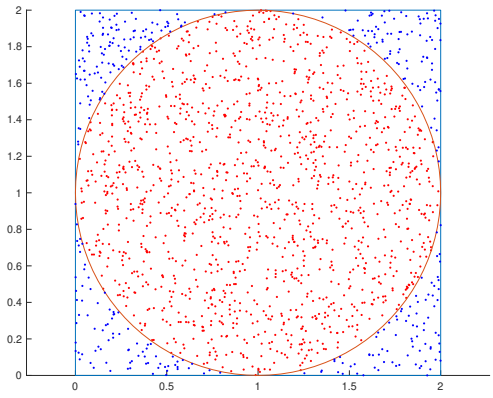

Figure 1: Schema risolutivo

Per cui, essendo l'area del cerchio pari a $A_c = \pi r^2$ e quella del quadrato $A_s = 4r^2$, come si diceva prima il rapporto delle due aree è proprio:

$$\frac{\pi r^2}{4r^2} = \frac{\pi}{4}, \tag{1}$$

quindi, se si moltiplica per 4 il rapporto tra le due aree si ottiene la stima di π che si stava cercando:

$$\pi \approx 4\frac{A_c}{A_s} \tag{2}$$

Implementazione seriale

In questa parte viene illustrato e commentato l' *algoritmo* dell'implementazione seriale che ho realizzato. All'interno della Listing 1, si può vedere l'algoritmo che ho scritto per la risoluzione del problema preso in esame.

```
#define SEED time(NULL)

double pi_serial_version(const long long int interval) {

  srand(SEED);

  double x, y;
  int i, points = 0;

  for (i = 0; i < interval; i ++) {
    x = (double) rand() / RAND_MAX;
    y = (double) rand() / RAND_MAX;

    if (x * x + y * y <= 1.0) points ++;
  }

  return (double) points / interval * 4.0;
}
```

Listing 1: Implementazione seriale

Considerazioni

Come si può vedere, l'algoritmo si presenta breve e semplice.
Preso in ingresso il numero di *samples* ($\equiv$ **s**) da generare, l'algoritmo inizia andando ad impostare il *seed* per la generazione dei numeri pseudo-casuali. Successivamente, si procede con il generare **s** coppie di numeri casuali che rappresentano le coordinate cartesiane dei punti. Ogni volta che una coppia di numeri viene generata, utilizzando la formula

$$\sqrt{x^2 + y^2} \leq \sqrt{R}, \tag{3}$$

andiamo a verificare che il punto sia contenuto all'interno della circonferenza. Nel caso di risposta affermativa viene incrementato il contatore `points`.
Una volta che sono state generate **s** coppie di punti utilizzando la formula 2, enunciata in precedenza, viene calcolata la stima di π.

Parallelizzazione

All'interno di questa sezione verranno considerati singolarmente due diversi metodi per *parallelizzare* il codice seriale esaminato in precedenza.
Gli strumenti utilizzati per la parallelizzazione sono stati: *MPI e OpenMP*.

MPI

MPI, acronimo di **M**essage **P**assing **I**nterface, come si può intuire dal nome viene utilizzato per lo scambio di messaggio tra processi. Più precisamente,

MPI è una specifica e *non* un'implementazione infatti, ne esistono diverse.
Per realizzare una computazione distribuita è necessaria una comunicazione tra i vari processi. Esistono principalmente due tecniche: **shared-memory** e **message-passing**.
Utilizzando la prima tecnica, il programma non si preoccupa degli altri processi e può accedere ad un'area di memoria condivisa tra tutti i processi. Se si utilizza la seconda tecnica invece, il programma deve occuparsi della comunicazione tra i vari processi e non esiste la nozione di memoria condivisa; infatti, ogni processo può accedere solo alla propria area di memoria.
MPI è lo standard del *message-passing*, infatti prevede la definizione di primitive per la comunicazione tra diversi processi. Possiamo vederlo come un *middelware* che si colloca tra il linguaggio di programmazione ed il sistema operativo.
Si faccia attenzione che più sono le unità coinvolte nella computazione più si rischia che la comunicazione diventi pesante.

OpenMP

Open specifications for **M**ulti **P**rocessing (OpenMP), è un API multi piattaforma per la creazione di applicazioni parallele su sistemi a memoria condivisa. È composto da un insieme di direttive di compilazione, routine di librerie e variabili d'ambiente che ne definiscono un corretto funzionamento a run-time.
OpenMP è un'implementazione del concetto di **multithreading**, ovvero di un metodo di parallelizzazione per cui un *master thread* crea un certo numero di *slave thread* e il task da eseguire è diviso tra questi ultimi. I thread vengono eseguiti in modo concorrente mentre il *runtime system* alloca i thread sui processori disponibili.

Implementazioni parallele

Una prima sostanziale differenza tra l'implementazione seriale e quella parallela sta nella funzione utilizzata per generare i numeri pseudo-casuali.
Mentre nell'algoritmo seriale si utilizzava la funzione `rand()`, nelle implementazioni parallele viene sostituita dalla funzione `rand_r()` alla quale viene passato il *seed*. Questo è dovuto dal fatto che la funzione `rand()` non è **thread-safe**[1], poiché utilizza uno stato nascosto che viene modificato ad ogni chiamata.

MPI

```
#define SEED time(NULL)

void pi_mpi_version(int argc, char **argv) {

  int rank, size;
  unsigned int seed;
  double x, y, start, end;
```

[1]Una porzione di codice viene definita *thread-safe* se funziona correttamente durante l'esecuzione simultanea di più thread.

```
  long long int i, all_point, points = 0, all_intern;

  MPI_Init(&argc, &argv);
  start = MPI_Wtime();

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  seed = SEED + rank;

  for ( i = 0; i < atoll(argv[1]); i++ ) {
    x = (double) rand_r(&seed) / RAND_MAX;
    y = (double) rand_r(&seed) / RAND_MAX;

    if ( x * x + y * y <= 1.0 ) points++;
  }

  MPI_Reduce(&points, &all_intern, 1, MPI_LONG_LONG_INT, MPI_SUM, 0,
    MPI_COMM_WORLD);
  all_point = atoll(argv[1]) * size;

  MPI_Barrier(MPI_COMM_WORLD);
  end = MPI_Wtime();

  if ( rank == 0 ) {
    printf("\u03C0 \u2248 %Lf\n", (long double) all_intern / all_point
    * 4.0);
    printf("Time elapsed: %.4f\n", end - start);
  }

  MPI_Finalize();
}
```

Listing 2: Parallelizzazione tramite MPI

Considerazioni

La prima istruzione che salta all'occhio, tralasciando le dichiarazioni di alcune variabili, è la funzione `MPI_Init()` attraverso la quale si dichiara l'inizio della computazione di *MPI*. Successivamente, dopo aver fatto partire il timer, che mi indicherà il tempo di esecuzione, troviamo altre due funzioni tipiche di MPI:

- MPI_Comm_rank,
- MPI_Comm_size,

con le quali andiamo a determinare rispettivamente l'identificatore del processo e il numero totale di processi utilizzati nella computazione. Si noti che in entrambe la funzioni viene passata la medesima costante: `MPI_COMM_WORLD` ovvero il **communicator**[2] che viene creato quando viene inizializzato un ambiente MPI.
Tenendo conto delle considerazioni fatte all'inizio della sezione, l'algoritmo prosegue nella generazione di n coppie di numeri pseudo-casuali. Ogni volta che una coppia è stata generata si controlla che il punto che individua nel piano sia contenuto all'interno della circonferenza: in caso di risposta affermativa la variabile contatore *points* viene incrementata.

[2] Il *communicator* è una delle strutture più importanti, la quale contiene uno o più gruppi di processi.

Dato che in *MPI* il codice viene eseguito da tutti i processi presenti all'interno del *communicator*, ognuno avrà una propria variabile *points*. A questo riguardo, utilizzando la funzione MPI_Reduce, andiamo a sommare tutte le variabili all'interno della variabile *all_point*.
Quando la funzione viene invocata, le vengono passati come parametri:

- *const void *sendbuf*: l'indirizzo della variabile contatore;
- *void *recvbuf*: l'indirizzo della variabile che conterrà il numero di tutti i punti interni alla circonferenza;
- *int count*: impostiamo questo campo ad 1 in quanto la *sendbuf* contiene un solo elemento;
- *MPI_Datatype datatype*: come valore assegniamo la constante **MPI_LONG_LONG_INT** (in modo da non avere problemi di overflow durante la fase di testing);
- *MPI_Op op*: dato che dobbiamo risommare tutti i valori della variabile *points*, impostiamo questo campo **MPI_SUM**;
- *int root*: inviamo tutto al processo con rank pari a zero, quindi settiamo questo campo a 0;
- *MPI_Comm comm*: il *communicator* nel quale stiamo computando, **MPI_COMM_WORLD**.

Un'altra cosa da notare è che, se si decide di generare n numeri pseudo-casuali, ogni processo ne genererà n; per cui avremo un totale di numeri generati pari a $p * n$.
Attraverso l'invocazione della funzione MPI_Barrier attendiamo che tutti i processi arrivino a quel punto dopo di che stoppiamo il timer.
Il calcolo effettivo della stima di π e quello del tempo di esecuzione viene fatto solo ed esclusivamente dal processo il cui *rank* è zero.
Il programma termina invocando la funzione MPI_Finalize con la quale viene terminata la computazione.

MPI V. 2

```
#define SEED time(NULL)

void pi_mpi_v2_version(int argc, char **argv) {

  int rank, size;
  unsigned int seed;
  double x, y, start, end;
  long long int i, all_point, points = 0;
  long long int sum = 0;

  MPI_Init(&argc, &argv);
  start = MPI_Wtime();

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
  seed = SEED + rank;

  long long int receiver[size];

  if ( rank != 0 ) {
    for ( i = 0; i < atoll(argv[1]); i++ ) {
      x = (double) rand_r(&seed) / RAND_MAX;
      y = (double) rand_r(&seed) / RAND_MAX;

      if ( x * x + y * y <= 1.0 ) points++;
    }

    for ( i = 0; i < size; i++ )
      MPI_Send(&points, 1, MPI_LONG_LONG_INT, 0, 1, MPI_COMM_WORLD);
  }

  all_point = atoll(argv[1]) * size;

  MPI_Barrier(MPI_COMM_WORLD);
  end = MPI_Wtime();

  if ( rank == 0 ) {
    for ( i = 0; i < size; i++ )
      MPI_Recv(&receiver[i], size, MPI_LONG_LONG_INT, MPI_ANY_SOURCE,
    1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    for ( i = 0; i < size; i++ ) sum += receiver[i];

    printf("\u03C0 \u2248 %Lf\n", (long double) sum / all_point * 4.0);
    printf("Time elapsed: %.4f\n", end - start);
  }

  MPI_Finalize();
}
```

Listing 3: Parallelizzazione tramite MPI Versione 2

Considerazioni

Questa versione di *MPI* si discosta dalla precedente per il modo in cui i vari processi si scambiano i "messaggi". Nella soluzione proposta in precedenza ogni processo generava n coppie di numeri pseudo-casuali e incrementava la propria variabile contatore e successivamente, utilizzando una *reduce*, si otteneva la somma totale di tutti i punti che erano contenuti all'interno della circonferenza.
Questa implementazione differisce dalla prima non tanto per il modo in cui vengono generati i samples, ma, come si diceva prima, per il modo in cui i processi comunicano tra di loro. Vediamo ora nel dettaglio.
Si può subito notare che il codice è diviso principalmente in due blocchi:

- se il rank è diverso da zero;
- se il rank è zero.

Tutti i processi il cui rank è diverso da zero saranno quelli che genereranno le coppie di numeri pseudo-casuali e poi invieranno il loro contatore all'unico processo il cui rank è zero.
Per cui, una volta che ogni processo ha terminato di generare i suoi n cam-

pioni, attraverso la funzione `MPI_Send` invierà al processo il cui rank è zero il numero di punti che ha generato e che sono contenuti all'interno della circonferenza. Questa funzione riceve come parametri:

- *const void *buf*: l'indirizzo della variabile contatore;
- *int count*: lo settiamo ad 1, inviamo un solo valore alla volta;
- *MPI_Datatype datatype*: impostiamo come valore la constante **MPI_LONG_LONG_INT** (in modo da non avere problemi di overflow durante la fase di testing);
- *int dest*: essendo la destinazione il processo con *rank* zero, impostiamo il valore a 0;
- *int tag*: il campo tag identifica il messaggio che viene mandato, in questo caso è impostato a 1;
- *MPI_Comm comm*: il *communicator* nel quale stiamo elaborando, **MPI_COMM_WORLD**.

Una volta che tutti i processi hanno generato e inviato le loro n coppie di numeri al processo di *rank* zero, quest'ultimo utilizzando la funzione `MPI_Recv` salva all'interno di un array tutti i dati che riceve dagli altri processi. Questa funzione prende come parametri:

- *void *buf*: l'indirizzo della cella dell'array nel quale salvare il valore;
- *int count*: lo settiamo ad *size* in quanto deve ricevere da tutti i processi;
- *MPI_Datatype datatype*: impostato ad **MPI_LONG_LONG_INT**;
- *int source*: dato che sono tutti i processi ad inviare dei dati lo impostiamo a **MPI_ANY_SOURCE** in modo che possa riceve da tutti;
- *int tag*: impostato ad 1;
- *MPI_Comm comm*: il *communicator* nel quale stiamo elaborando, **MPI_COMM_WORLD**;
- *MPI_Status *status*: dato che lo vogliamo ignorare, lo impostiamo a **MPI_STATUS_IGNORE**.

A questo punto, ci troviamo con un array di dimensione *size* le cui celle contengono i singoli contatori dei vari processi. Ora si procede con il sommare il contenuto di tutte le celle dell'array per poi applicare la formula 2 e trovare la stima di π che stiamo cercando.
Anche in questo caso, il programma termina con l'invocazione della funzione `MPI_Finalize()` .

OpenMP

```
#define SEED time(NULL)

double pi_openmp_version(const long long int interval, const int thread
    , double *timer) {

  unsigned int seed;
  int i, points = 0;
  double x, y, start;

  start = omp_get_wtime();
  #pragma omp parallel num_threads(thread) private(seed, x, y)
    reduction(+:points)
  {

    seed = SEED * omp_get_thread_num();

    #pragma omp for
    for ( i = 0; i < interval; i++ ) {
      x = (double) rand_r(&seed) / RAND_MAX;
      y = (double) rand_r(&seed) / RAND_MAX;

      if (x * x + y * y <= 1.0) points ++;
    }
  }
  (*timer) = omp_get_wtime() - start;

  return (double) points / interval * 4.0;
}
```

Listing 4: Parallelizzazione tramite OpenMP

Considerazioni

A differenza della versione seriale, come si può notare la funzione ha un prototipo diverso. Questo è dovuto al fatto che ho preferito lasciare all'utente la decisione di scegliere oltre a quanti campioni generare anche il numero di thread sui quale lanciare il programma. Oltre al valore stimato di π viene ritornato anche il tempo di esecuzione parallela, calcolato usando la funzione di libreria `omp_get_wtime()`.
Si possono notare subito alcuni nuovi costrutti come:

- #pragma omp parallel;
- #pragma omp for.

Il primo è il costrutto *parallel* di OpenMP, dove viene effettivamente fatta la fork[3]; il secondo invece è una direttiva di *work-sharing*[4], la quale suddivide in automatico un ciclo `for` tra i thread di una data regione parallela, rendendo automaticamente la variabile di ciclo **privata** e suddivisa fra i vari thread.

La prima istruzione che si incontra all'interno della regione parallela è un

[3]Dal master thread, colui designato ad esegue il codice, viene creato un *team* di thread che eseguono il calcolo in parallelo. Alla fine, il team di thread cessa di esistere e rimane solo il thread master che prosegue la computazione in maniera seriale.

[4]Una costrutto di work-sharing consente ai thread di gestire il modus operandi in maniera parallela. Viene chiamato da tutti i thread e contiene una barriera implicita in modo tale che tutti i thread prima di passare oltre al costrutto devo aver terminato il proprio lavoro.

assegnamento. Attraverso quest'ultimo, andiamo ad impostare il seed che poi verrà passato al generatore di numeri pseudo-casuali. Per rendere univoco il seed moltiplichiamo il valore di ritorno della funzione `time(NULL)` per l'*id* del thread.
A questo punto si arriva al cuore dell'algoritmo.
Infatti, troviamo la seconda direttiva di OpenMP utilizzata. Come si diceva in precedenza, il ciclo `for` viene suddiviso tra i vari thread per cui ciascuno di questi genera e verifica che *interval / thread* punti siano contenuti all'interno della circonferenza e in caso affermativo, come nella versione seriale, viene incrementato un contatore.
Dato che il numero di samples da generare è suddiviso tra i vari thread, ognuno avrà il suo contatore; per cui, attraverso la specifica `reduce(+:points)` , facciamo si che alla fine in *points* sia contenuto il conteggio totale di tutti i punti contenuti all'interno della circonferenza.
L'algoritmo si conclude ritornando il tempo di esecuzione utilizzando la tecnica *pass-by-reference* ed il valore stimato di π calcolato sempre utilizzando la formula 2.

Metriche di confronto

Execution Time

Sia T_s (**serial runtime**) il tempo trascorso dall'inizio alla fine dell'esecuzione della versione seriale dell'algoritmo; inoltre sia T_p (**parallel runtime**) il tempo speso per la risoluzione di un problema con p processi.
Infine si definisce T_o (**total parallel overhead**) come la differenza tra il tempo impiegato collettivamente da tutti gli elementi di elaborazione e il tempo richiesto dell'algoritmo sequenziale più veloce di cui siamo a conoscenza.

Speedup

Con il termine **speedup** (S) si rappresenta il guadagno in termini di prestazioni che si ottiene andando a parallelizzare una data applicazione a partire dall'implementazione seriale.
Lo speedup è quindi una misura che mi indica il vantaggio di risolvere un problema in parallelo.
Per cui S viene definito come:

$$S = \frac{T_s}{T_p}. \tag{4}$$

Il valore massimo che S potrebbe assumere è p infatti lo speedup ha un limite superiore pari a $S \leq p$ questo perché se S fosse più grande di p allora ogni processo impiegherebbe meno di T_s / p a risolvere il problema.

Efficiency

L'**Efficiency** rappresenta la frazione di tempo per cui un elemento di elaborazione viene utilmente impiegato. In termini matematici, è rappresentato dal

rapporto tra lo *speedup* e il numero di processi impiegati:

$$E = \frac{S}{p} \tag{5}$$

La situazione perfetta è, come si può immaginare, quella in cui si ha $S \equiv p \rightarrow E = 1$. Purtroppo questa situazione non può verificarsi nella realtà dal momento che rappresenta un sistema ideale.

Analisi dei risultati

Ho voluto valutare le implementazioni parallele rispetto a quella seriale utilizzando come termini di confronto: **Execution Time, Speedup, Efficiency**, spiegati all'interno della sezione precedente.
Prima di calcolare i valori delle metriche, ho deciso di eseguire ogni singola implementazione una decina di volte. Questo in modo tale da avere un valore stazionario al quale i valori si avvicinano di più.
Per le diverse simulazioni che ho eseguito, ho sempre utilizzato un input crescente, da 1e2 fino a 1e9 inoltre, per le implementazioni parallele ho utilizzato 20 processi/thread.

Come spiegato durante le analisi degli algoritmi realizzati, i tempi di esecuzione sono stati presi utilizzando opportune funzioni di libreria. In figura 2 si può vedere il comportamento di ciascun algoritmo al crescere dell'input. Come ci si può aspettare, il tempo di esecuzione dell'algoritmo seriale cresce

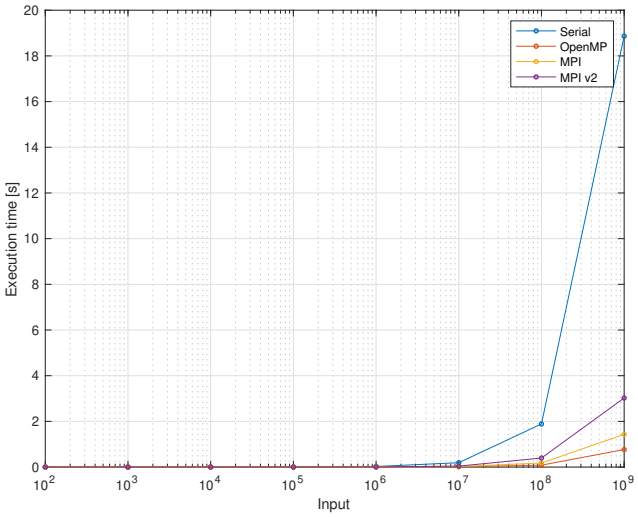

Figura 2: Confronto input - tempo

molto velocemente con il crescere dell'input. Infatti, con input massimo il tempo raggiunge un picco di circa 18 s. Per quanto riguarda gli algoritmi paralleli, possiamo notare come per input piccoli hanno lo stesso comportamento

della versione seriale ma invece, come si immagina, per input di dimensioni maggiori, essendo il carico di lavoro distribuito fra i vari processi/thread, si registra un tempo inferiore. Dall'analisi dei risultati ottenuti, sull'input più grande, le implementazioni parallele in media impiegano all'incirca 17 s.

Una volta aver esaminato e calcolato i vari tempi di tutti gli algoritmi, andando ad applicare la formula 4 sono andato a calcolare lo speedup. Come si evince dalla figura 3, ci son dei casi in cui l'algoritmo scritto utilizzando *OpenMP* assume un valore maggiore rispetto al limite superiore che sappiamo essere nel nostro caso pari a 20.
La domanda viene lecita: *se S deve essere minore di p, perché succede il contrario con determinati input?* Questa situazione prende il nome di **Superlinear Speedup** ed è dovuto al fatto che ciascun elemento di elaborazione impiega meno di T_s/p per la risoluzione del problema.

Un'altro metodo per accorgersi che abbiamo a che fare con un sistema superlineare è andando a calcolare l'efficienza. Infatti, applicando la formula 5 ai risultati ottenuti dalle simulazioni otteniamo l'andamento che si trova in figura 4.
Come si può notare, sempre nel caso dell'implementazione in *OMP*, abbiamo dei casi in cui $E > 1$. Anche in questo caso ci si può porre la domanda: *ma com'è possibile avere un'efficienza maggiore di uno?* La risposta è la medesima, stiamo trattando un sistema superlineare per cui nella versione parallela dell'algoritmo ogni processo fa meno lavoro della controparte seriale.
Mentre, per quanto riguarda le versioni scritte utilizzando *MPI* abbiamo una situazione regolare in cui lo speedup, mantenendo costante il numero di processi utilizzati, cresce al crescere dell'input.

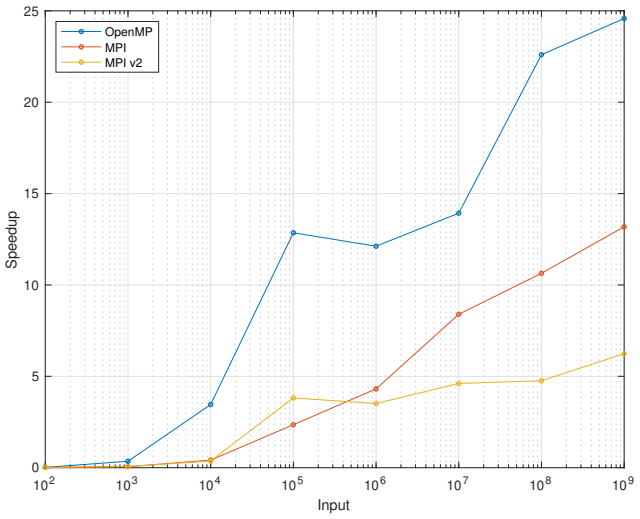

Figura 3: Confronto input - speedup

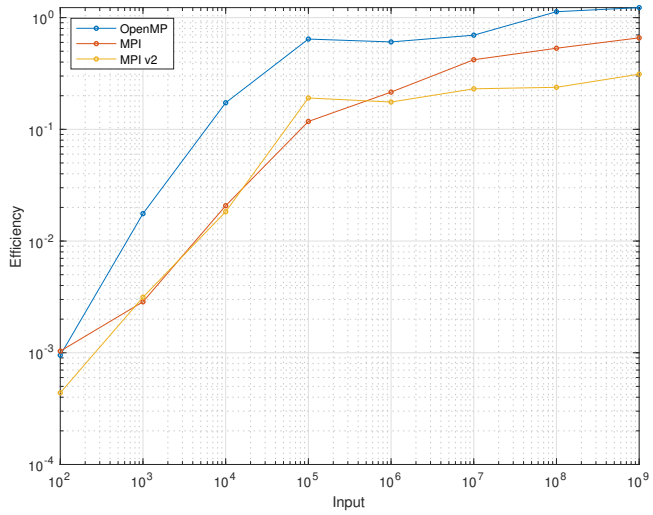

Figura 4: Confronto input - efficienza

Conclusioni

Dalle analisi effettuate e dai risultati ottenuti possiamo osservare che il problema affrontato può essere ben parallelizzato ottenendo percui buone prestazioni. Come si può notare dai risultati ottenuti, la soluzione migliore risulta essere quella scritta utilizzando *OpenMP*, ma nonostante questo anche con la prima versione di *MPI* otteniamo dei risultati apprezzabili.
Al contrario, con la seconda versione di *MPI* non raggiungiamo dei buoni risultati per quanto riguarda l'efficienza e lo speedup.