Documentazione somma di N numeri su architettura MIMD-DM

Parallel and distributed computing 2023-2024

Ferdinando D'Alessandro N86003933 Gian Marco Addati N86003795

Contents

1 Descrizione del problema

Il problema proposto è la somma di N numeri su architettura parallela MIMD-DM.
Per quanto possa sembrare un problema banale da risolvere sequenzialmente, l'operazione di somma è un'operazione di riduzione: la quantità di dati di partenza viene ridotta ad un singolo output, questo significa che avanzando nella soluzione del problema il dataset da gestire si riduce sempre più, rendendo complesso lavorare davvero in parallelo.

L'architettura MIMD-DM (Multiple instructions multiple data, Distributed memory) è una delle architetture della tassonomia di Flynn, in particolare è un'architettura parallela in cui più unità di elaborazione eseguono istruzioni diverse su file diversi, senza condividere la memoria.
Vedremo infatti che la distribuzione e comunicazione dei dati è parte integrante e complessa dell'algoritmo risolutivo, data la memoria distribuita dell'architettura.

2 Descrizione dell'algoritmo

La componente sequenziale dell'algoritmo è ovvia e rigida, infatti per ottenere la somma di N numeri dovremo necessariamente effettuare N-1 somme. La variabilità sta nei compiti assegnati ai diversi processori e nel modo in cui comunicano.
Di conseguenza l'algoritmo risolutivo avrà una generica struttura di:

1. Ottenimento dei dati da parte di P0;
2. Distribuzione dei dati agli altri processori;
3. Calcolo della somma locale da parte di tutti i processori;
4. Comunicazione delle somme parziali;
5. Calcolo della somma totale.

Per quanto riguarda la fase di comunicazione delle somme parziali abbiamo utilizzato 3 strategie diverse, dando la possibilità all'utente di scegliere la strategia desiderata mediante riga di comando.

La prima cosa che viene fatta è il controllo sulla strategia, in quanto le strategie numero 2 e 3 possono essere eseguite solo su un numero di processori potenza di 2. In caso questa condizione non sia rispettata, viene forzata la strategia 1 avvertendo l'utente e si procede con l'esecuzione.

I dati sono generati randomicamente se era richiesta dall'utente la somma di più di 20 valori oppure in caso contrario sono ricavati dagli argomenti dell'eseguibile (Leggendo quindi opportunamente il contenuto da argv). In entrambi i casi sono poi posti in un vettore allocato dinamicamente, mediante malloc.

Vengono distribuiti i dati, avendo cura che tutti i processori ricevano lo stessa quantità di valori su cui lavorare per quanto possibile (Seguiranno più dettagli a riguardo in questo capitolo) ed ogni processo eccetto P0 alloca dinamicamente un vettore della giusta dimensione in cui porre i dati ricevuti.
P0 utilizzerà il vettore di partenza avendo cura di trattare solamente i dati che non sono destinati agli altri processi.

In seguito ogni processore calcola la somma dei dati ricevuti ed infine si procede con la parte di algoritmo dipendente dalla strategia scelta.

Strategia I Ogni processore (eccetto lo 0) invia la propria somma parziale al processore 0, che somma i diversi risultati ottenendo così la somma totale.

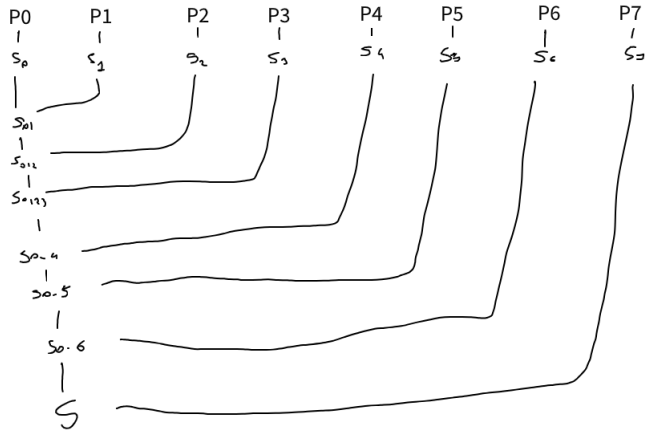


Figure 1: Strategia 1 su 8 processori

E' chiaro che in questa strategia abbiamo un parallelismo minimo, dopo la prima fase di somma locale tutti i processori che non sono P0 si troveranno ad aspettare il lavoro di quest'ultimo. Le comunicazioni inoltre sono tutte sequenziali, quindi abbiamo un enorme impatto del costo della comunicazione sul tempo di esecuzione, che risulterà particolarmente evidente su piccoli input. Cercheremo di rimediare a questi problemi nella seconda strategia, per quanto possibile.

Strategia II Ad ogni passo si formano delle coppie di processori composte da mittente e destinatario, in cui il mittente invierà la propria somma al destinatario, che la unirà alla propria somma parziale. Successivamente il destinatario si ferma e tra i processori rimasti si instaurano nuovamente le coppie, ripetendo il procedimento.
Alla fine avremo i dati localizzati in P0, che potrà calcolare la somma totale.

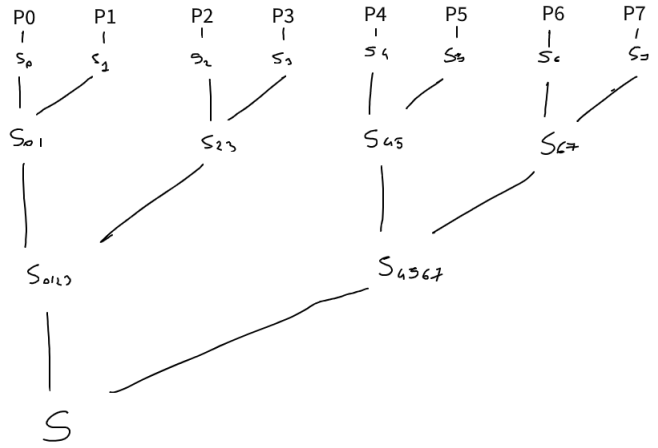


Figure 2: Strategia 2 su 8 processori

Come anticipato, in questa strategia sfruttiamo il parallelismo dei processori il più possibile nei limiti del problema. Questo dovrebbe aumentare l'efficienza dell'algoritmo, come verificheremo sperimentalmente successivamente, avendo anche permesso ad alcune delle comunicazioni di avvenire in parallelo, riducendone l'impatto sul risultato.

Strategia III Seguiamo i passi della strategia II ma invece di portare tutti i dati man mano verso 0, effettuiamo degli scambi di dati tra le varie coppie di algoritmi, portando così tutti i processori ad avere la somma totale in memoria.

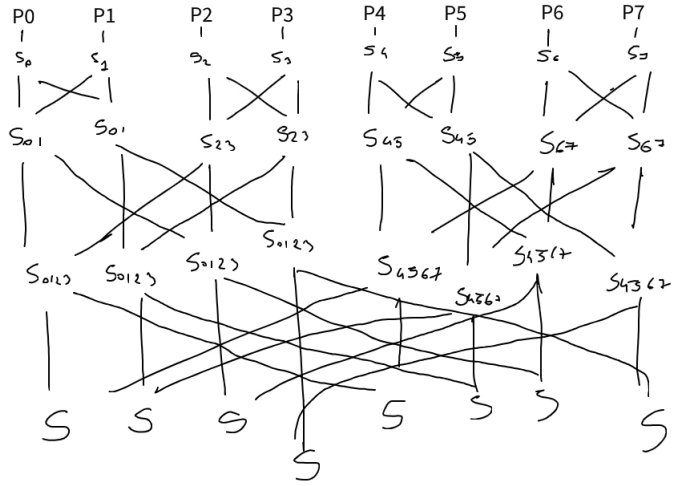


Figure 3: Strategia 3 su 8 processori

Questa strategia sfrutta al massimo il parallelismo, facendo lavorare tutti i processori fino alla soluzione del problema, ma non ne ricava alcun'aumento di velocità a livello teorico.
Il beneficio che porta l'uso di questa strategia è la disponibilità a fine esecuzione del risultato su tutti i processori, che quindi sono pronti per un eventuale altro lavoro in parallelo a partire da questo risultato, senza alcuna comunicazione aggiuntiva.

Le comunicazioni in più che vengono effettuate rispetto alla seconda strategia sono effettuate in parallelo e quindi non dovrebbero inficiare sulle prestazioni della strategia II.

Nell'implementazione dell'algoritmo sono risultati più impegnativi i passaggi di distribuzione iniziale dei dati e di comunicazione tra i processi nel caso delle strategie II e III.

La distribuzione equa dei dati richiede di dividere il numero di dati per il numero di processori utilizzati ed in caso di presenza di resto *rest*, dividere i dati aggiuntivi tra i primi *rest* processori, dandogli quindi un dato in più a testa.

Per quanto riguarda la comunicazione nella seconda strategia, analizzando il procedimento di comunicazione, si può notare che ad ogni passo si accumula la somma di 2^i processori, per cui otterremo la somma finale dopo $log_2(nproc)$ iterazioni.
Notiamo inoltre che all' i-esima iterazione partecipano allo scambio di dati solo i processori con rank divisibile per 2^i ed il ricevente sarà il processore che partecipa anche alla prossima iterazione, cioè il cui rank è divisibile anche per $2^(i+1)$.
In un ciclo che si ripete $log_2(nproc)$ volte quindi, facciamo sì che il processore con rank divisibile per 2^i ma non per $2^i + 1$ invii i dati al processore con rank divisibile anche per $2^(i+1)$, fino a portare i dati in P0.

Per quanto il numero di passi sia lo stesso, la comunicazione procede in modo diverso nella strategia III, perchè ad ogni passo partecipano tutti i processori.

Possiamo notare però che ogni processore scambia i dati con il processore con tag distante 2^i, quindi similmente possiamo semplicemente ad ogni iterazione far scambiare i dati tra i suddetti processori.
Nel codice questo è stato implementato facendo comunicare i processori tali che $rank\%2^(i+1) < 2^(i)$ con i processori che non rispettano questa condizione e con rank distante 2^i, inviando poi reciprocamente i dati.
Utilizzando le funzioni di comunicazione bloccanti, abbiamo dovuto porre invio e ricezione dei dati in ordine inverso nel mittente rispetto al destinatario, per evitare di cadere in un *deadlock*.

In generale come tag per le comunicazioni abbiamo sempre scelto una costante sommata al ramk del processo mittente, così da poter facilmente ricavare il tag giusto da utilizzare ed evitare di avere scambi di messaggi a vuoto, causa di conseguenti *deadlock*.

3 Descrizione delle routine

Per semplicità, essendo l'algoritmo non eccessivamente complesso, il codice è stato scritto interamente nel main, ad eccezione della routine per la generazione dei numeri random,

```
int random_int()
```

funzione che utilizzando rand ed srand con seed ottenuto mediante la funzione time(), genera un numero pseudocasualmente e poi facendo l'operazione modulo 100 ritorna un numero random tra 0 e 99 con distribuzione approssimabilmente uniforme.

Abbiamo inoltre fatto uso delle seguenti funzioni di libreria standard:

- Dalla libreria **stdlib.h**:
  - `int atoi(const char *nptr)`

    La funzione atoi(...) converte una stringa in integer. L'abbiamo utilizzata per convertire i parametri passati dall'utente al main (Come strategia desiderata, numero di input etc.), confidando che l'utente passi dei parametri convertibili ad un intero, poichè la funzione non è robusta, ha comportamento indefinito in caso di stringa non convertibile.
    Abbiamo comunque utilizzato questa funzione per la sua semplicità d'uso, ritenendo che la robustezza non fosse il focus del codice scritto.
  - `void *malloc(size_t size)`

    La procedura malloc(...) alloca un blocco di *size* byte all'interno dell'heap e restituisce un puntatore al primo byte allocato. Questa procedura è stata utilizzata per allocare un vettore dinamicamente in cui porre i dati da sommare.
    Si è preferito allocare il vettore dinamicamente in quanto data la quantità di dati da sommare, un vettore statico si sarebbe potuto rivelare inefficace, data la dimensione ridotta dello stack rispetto all'heap, portando ad eventuali segmentation fault.
  - `void free(void *ptr)`

    free(...) è la procedura opposta alla malloc, dealloca il blocco di memoria puntato da *ptr*.
    E' necessario utilizzare una free per ogni malloc utilizzata per evitare memory leakage, che lavorando con così tanti dati su numerose iterazioni possono rivelarsi incredibilmente dannose.
  - `void srand(unsigned int seed)`
    `int rand(void)`

    La funzione srand(...) utilizza *seed* come seme per la generazione di una sequenza pseudocasuali di valori, che vengono ritornati dalla funzione rand().
- Dalla libreria **math.h**:
  - `double log(double x)`

    La funzione log(...) ritorna il logaritmo naturale di x. E' stata utilizzata per calcolare il logaritmo in base 2 del numero di processori, sfruttando la seguente proprietà dei logaritmi:

$$log_2(n) = \frac{log_e(n)}{log_e(2)} \tag{1}$$

Infine abbiamo fatto uso delle seguenti funzioni della libreria **OpenMPI**:

- `int MPI_Init(int *argc, char ***argv)`

  MPI_Init(...) è la funzione che si occupa di inizializzare l'ambiente di esecuzione parallelo, occupandosi di distribuire gli argomenti ricevuti ai processi creati.
  Deve essere chiamata prima di qualsiasi altra funzione di MPI.

- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`

  Questa funzione imposta in *rank*, passato per riferimento mediante puntatore, il rank del processo chiamante all'interno del communicator *comm*.
  Nel codice è stata utilizzata per inizializzare il valore *menum* facendo uso del communicator globale, *MPI_COMM_WORLD*.

- `int MPI_Comm_size(MPI_Comm comm, int *size)`

  Questa funzione imposta in *size*, passato per riferimento, la taglia del communicator *comm*, ovvero il numero di processori al suo interno.
  Nel codice è stata utilizzata per inizializzare il valore *nproc* facendo uso del communicator globale, *MPI_COMM_WORLD*.

- `int MPI_Bcast(void *buffer, int count,`
  `MPI_Datatype datatype, int root, MPI_Comm comm)`

  Questa funzione invia dal processo con rank *root* a tutti i processi presenti in *comm*, *count* dati di tipo *datatype* dal buffer *buffer*.
  Deve essere chiamata da tutti i processi: nei processi che non hanno rank *root*, questa funzione farà una ricezione dei dati inviati da *root* con un certo tag; in root questa funzione effettuerà l'invio dei dati a tutti i processi in *comm* (incluso se stesso, per cui seguirà una ricezione dei suoi stessi dati per evitare deadlock).
  All'interno del codice è stata utilizzata per inviare i dati uguali per tutti i processi, come strategia scelta dall'utente e numero di dati in input.

- `int MPI_Send(const void *buf, int count,`
  `MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`

  Questa funzione fa inviare al processo chiamante *count* dati di tipo *datatype* dal buffer *buffer* al processo con rank *dest* su communicator *comm*, con tag *tag*.
  La MPI_Send è una funzione bloccante: il flusso d'esecuzione del processo chiamante sarà ripreso solo quando il messaggio sarà ricevuto mediante una MPI_Recv(...) dal processo con rank *dest*, con lo stesso tag della send.
  All'interno del codice è stato utilizzato per la comunicazione delle somme parziali tra i processi.

- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype,`
  `int source, int tag, MPI_Comm comm, MPI_Status *status)`

  Controparte della MPI_Send(...), la MPI_Recv(...) presenta gli stessi parametri, speculari dei parametri della send, ad eccezione di *status*, parametro che viene settato all'interno della funzione per avere informazioni aggiuntive sul risultato dell'operazione di ricezione.
  Come la send, questa è una funzione di ricezione bloccante, il processo chiamante sarà interrotto fin quando non viene ricevuto un messaggio dalla giusta sorgente con il giusto tag.
  All'interno del codice è stato utilizzato per la comunicazione delle somme parziali tra i processi.

- `int MPI_Barrier(MPI_Comm comm)`

  Funzione che interrompe il flusso d'esecuzione di tutti i processi appartenenti al communicator *comm* fin quando non hanno eseguito tutti la funzione stessa. Funge quindi da controllo dell'esecuzione, ci assicura che sincronizzera il flusso dei processi nel punto in cui è utilizzata.
  Nel codice è stata utilizzata per la misurazione dei tempi d'esecuzione dei vari processi, per assicurarci che fossero sincronizzati nella registrazione del tempo d'inizio. (E' stata quindi anteposta alla prima chiamata fatta alla funzione di MPI_Wtimte()).

- `double MPI_Wtime()`

  Questa funzione che ritorna il numero di secondi passati da un certo evento passato, che è garantito non variare durante la vita del processo.
  Facendo la differenza tra due risultati di MPI_Wtime() posti ad inizio e fine procedimento di somma abbiamo calcolato il tempo impiegato dal singolo processo terminare il proprio compito. Tra i vari tempi ottenuti abbiamo poi preso il massimo, utilizzando la funzione di MPI_Reduce(...).

- ```
  int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
         MPI_Datatype datatype, MPI_Op op, int root,
         MPI_Comm comm)
  ```

  Questa funzione permette di eseguire una funzione di riduzione (*op*) tra i dati contenuti nei buffer *sendbuf* dei vari processi appartenenti al communicator *comm*. Il risultato dell'operazione è poi posto nel buffer *recvbuf* del processo con rank *root*.
  Tra le operazioni possibili troviamo tutte le operazioni di riduzione, come somma, prodotto, massimo e minimo.
  Nel codice scritto è stata utilizzata per calcolare il massimo tra i tempi di esecuzione registrati dai vari processi.

Quasi tutte le funzioni di MPI ritornano un valore intero come error value, per notificare l'eventuale fallimento dell'operazione.
  ```

4 Descrizione dei test

4.1 PBS "tipo" utilizzato

Il PBS tipo utilizzato per il test sui tempi di esecuzione per la prima strategia è il seguente:

```
#!/bin/bash

#PBS -q studenti
#PBS -l nodes=8:ppn=8
#PBS -N sommas1p2
#PBS -o sommas1p2.out
#PBS -e sommas1p2.err

sort -u $PBS_NODEFILE > hostlist

PBS_O_WORKDIR=$PBS_O_HOME/sommanumeri

/usr/lib64/openmpi/1.4-gcc/bin/mpicc -o $PBS_O_WORKDIR/sommas1p2
    $PBS_O_WORKDIR/somma.c

for n in 100000 1000000
do
    for NCPU in 1 2 3 4 5 6 7 8
    do
        /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np
            $NCPU $PBS_O_WORKDIR/sommas1p2 $n 1
    done
done

echo ----------------------------------------------------------------------
```

Le righe di codice più significative sono:

- `#PBS -l nodes=8:ppn=8`

  Questa è una direttiva PBS che segnala l'occupazione di 8 nodi e di 8 processori per nodo (ppn). Essendo il cluster composto per l'appunto di 8 nodi da 8 processori l'uno, qui indichiamo che vogliamo bloccare l'intero cluster per eseguire il nostro test, così da non poter avere rallentamenti causati da altri job che vengono runnati contemporaneamente al nostro.
  Tuttavia non è in quella riga che specifichiamo quali processori effettivamente utilizzeremo, che è un processo leggermente più complesso.

- `sort -u $PBS_NODEFILE > hostlist`

  La variabile PBS_NODEFILE contiene il nome del file contenente la lista di tutti i processori che abbiamo occupato nella precedente direttiva.
  Facendone il sort con l'opzione di unique, rimuoviamo tutti i nomi uguali, ottenendo così solamente un processore per nodo (Essendo il nome del processore dato dal nodo di appartenenza). Procediamo in questo modo perchè per ottenere un ambiente distribuito, dato che i processori all'interno di un singolo cluster condividono la memoria. (Questo significa che potremo fare uso al massimo di 8 processori all'interno dei nostri test)
  Questo risultato è posto in un file *hostlist* che poi passeremo come opzione *-machinefile* a *mpiexec*.

```
for n in 100000 1000000
do
for NCPU in 1 2 3 4 5 6 7 8
    do
    /usr/lib64/openmpi/1.4-gcc/bin/mpiexec ...
    done
done
```

In questo doppio ciclo for si esegue il test al variare del numero di input e di cpu utilizzate, bisogna notare che questo è uno spezzone di uno dei diversi file PBS dedicati al testing: si sono suddivisi in più file i test in modo da non mantenere un eseguibile in coda troppo a lungo.
E' naturale che lo stesso ciclo for nei file PBS dedicati alle strategie II e III non utilizzerà un numero di processori che va da 1 a 8 ma utilizzerà solamente numeri potenze di 2 (1, 2, 4, 8).

4.2 Esempi d'uso in casi limite

- /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np 4 $PBS_O_WORKDIR/sommaLIMITE 30 **0** (4 processori, 30 numeri in input, **strategia 0**)

  sommaLIMITE.err:

  ```
  Input della strategia errato: Success
  --------------------------------------------------------------------------
  mpiexec has exited due to process rank 2 with PID 13821 on
  node wn275.scope.unina.it exiting without calling "finalize". This may
  have caused other processes in the application to be
  terminated by signals sent by mpiexec (as reported here).
  --------------------------------------------------------------------------
  ```

  Il programma accetta solamente valori di strategia interi compresi tra 1 e 3, quindi stampa su standard error il precedente messaggio di errore.
  La stringa "Success" è stampata perchè non imposto nel codice manualmente il valore di errno, che quindi rimane il valore standard di successo.

- /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist **-np 7** $PBS_O_WORKDIR/sommaLIMITE 30 **2** (**7 processori**, 30 numeri in input, **strategia 2**)

  sommaLIMITE.out:

  ```
  (...)
  E' stata richiesta la strategia 2 ma il numero di processi richiesta non e' potenza di due,
  si prosegue con la strategia 1
  La somma totale calcolata con la prima strategia e' 330
  (...)
  ```

  La strategia 2 non può essere utilizzata su un numero di processori non potenza di 2, quindi come richiesto da traccia, la somma è eseguita utilizzando la strategia 1.

- /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np 4 $PBS_O_WORKDIR/sommaLIMITE **0** 2 (4 processori, **0 numeri in input**, strategia 2)

```
sommaLIMITE.err:

Numero di input non valido!: No such file or directory
--------------------------------------------------------------------------
mpiexec has exited due to process rank 0 with PID 18709 on
node wn273.scope.unina.it exiting without calling "finalize". This may
have caused other processes in the application to be
terminated by signals sent by mpiexec (as reported here).
--------------------------------------------------------------------------
```

Il numero di input deve essere positivo.
La stringa che segue al messaggio stampato deriva dall'errno, che non abbiamo modificato personalmente.

- /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist **-np 4** $PBS_O_WORKDIR/sommaLIMITE **3** 1 2 3 2 (**4 processori**, **3 numeri in input**, strategia 2, valori passati: 1 2 3)

```
sommaLIMITE.err:

Numero di input minore del numero di processori utilizzati: No such file or directory
--------------------------------------------------------------------------
mpiexec has exited due to process rank 0 with PID 18709 on
node wn273.scope.unina.it exiting without calling "finalize". This may
have caused other processes in the application to be
terminated by signals sent by mpiexec (as reported here).
--------------------------------------------------------------------------
```

Il numero di input deve essere maggiore o uguale al numero di processori utilizzati, affinchè abbia senso il problema della somma.
La stringa che segue al messaggio stampato deriva dall'errno, che non abbiamo modificato personalmente.

- /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist **-np 4** $PBS_O_WORKDIR/sommaLIMITE

```
sommaLIMITE.err:

Numero di argomenti insufficiente: No such file or directory
--------------------------------------------------------------------------
mpiexec has exited due to process rank 0 with PID 18709 on
node wn273.scope.unina.it exiting without calling "finalize". This may
have caused other processes in the application to be
terminated by signals sent by mpiexec (as reported here).
--------------------------------------------------------------------------
```

Il numero di argomenti dev'essere almeno 2 (n e strategia scelta)
La stringa che segue al messaggio stampato deriva dall'errno, che non abbiamo modificato personalmente.

4.3 Esempi d'uso in casi esemplificativi

- /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np 4 $PBS_O_WORKDIR/sommaLIMITE 10 1 1 1 1 1 1 1 1 1 1 (4 processori, 10 volte 1 in input, strategia 1)

```
La somma totale calcolata con la prima strategia e' 10
La somma totale calcolata con la prima strategia e' 10
La somma totale calcolata con la prima strategia e' 10
La somma totale calcolata con la prima strategia e' 10
La somma totale calcolata con la prima strategia e' 10
La somma totale calcolata con la prima strategia e' 10
La somma totale calcolata con la prima strategia e' 10
La somma totale calcolata con la prima strategia e' 10
La somma totale calcolata con la prima strategia e' 10
La somma totale calcolata con la prima strategia e' 10
Strategia usata: 1,
Numero di input: 10,
Numero di processori:4,
Tempo impiegato: 6.291866e-05
```

- /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np 4 $PBS_O_WORKDIR/sommaLIMITE 10 1 2 3 4 5 6 7 8 9 10 2 (4 processori, valori da 1 a 10, strategia 2)

```
La somma totale calcolata con la seconda strategia e' 55
La somma totale calcolata con la seconda strategia e' 55
La somma totale calcolata con la seconda strategia e' 55
La somma totale calcolata con la seconda strategia e' 55
La somma totale calcolata con la seconda strategia e' 55
La somma totale calcolata con la seconda strategia e' 55
La somma totale calcolata con la seconda strategia e' 55
La somma totale calcolata con la seconda strategia e' 55
La somma totale calcolata con la seconda strategia e' 55
La somma totale calcolata con la seconda strategia e' 55
Strategia usata: 2,
Numero di input: 10,
Numero di processori:4,
Tempo impiegato: 7.009506e-05
```

- /usr/lib64/openmpi/1.4-gcc/bin/mpiexec -machinefile hostlist -np 4 $PBS_O_WORKDIR/sommaLIMITE 10 10 9 8 7 6 5 4 3 2 1 3 (4 processori, valori da 10 a 1, strategia 3)

```
Sono il processo 0, la somma totale calcolata con la terza strategia e' 55
Sono il processo 1, la somma totale calcolata con la terza strategia e' 55
Sono il processo 3, la somma totale calcolata con la terza strategia e' 55
Sono il processo 0, la somma totale calcolata con la terza strategia e' 55
Sono il processo 1, la somma totale calcolata con la terza strategia e' 55
Sono il processo 2, la somma totale calcolata con la terza strategia e' 55
Sono il processo 3, la somma totale calcolata con la terza strategia e' 55
Sono il processo 0, la somma totale calcolata con la terza strategia e' 55
Sono il processo 1, la somma totale calcolata con la terza strategia e' 55
Sono il processo 2, la somma totale calcolata con la terza strategia e' 55
Sono il processo 3, la somma totale calcolata con la terza strategia e' 55
```

```
Sono il processo 2, la somma totale calcolata con la terza strategia e' 55
Sono il processo 0, la somma totale calcolata con la terza strategia e' 55
Sono il processo 1, la somma totale calcolata con la terza strategia e' 55
Sono il processo 2, la somma totale calcolata con la terza strategia e' 55
Sono il processo 3, la somma totale calcolata con la terza strategia e' 55
Sono il processo 0, la somma totale calcolata con la terza strategia e' 55
Sono il processo 1, la somma totale calcolata con la terza strategia e' 55
Sono il processo 2, la somma totale calcolata con la terza strategia e' 55
Sono il processo 3, la somma totale calcolata con la terza strategia e' 55
Sono il processo 1, la somma totale calcolata con la terza strategia e' 55
Sono il processo 3, la somma totale calcolata con la terza strategia e' 55
Sono il processo 1, la somma totale calcolata con la terza strategia e' 55
Sono il processo 3, la somma totale calcolata con la terza strategia e' 55
Sono il processo 0, la somma totale calcolata con la terza strategia e' 55
Sono il processo 2, la somma totale calcolata con la terza strategia e' 55
Sono il processo 0, la somma totale calcolata con la terza strategia e' 55
Sono il processo 2, la somma totale calcolata con la terza strategia e' 55
Sono il processo 0, la somma totale calcolata con la terza strategia e' 55
Sono il processo 2, la somma totale calcolata con la terza strategia e' 55
Sono il processo 1, la somma totale calcolata con la terza strategia e' 55
Sono il processo 3, la somma totale calcolata con la terza strategia e' 55
Sono il processo 0, la somma totale calcolata con la terza strategia e' 55
Sono il processo 1, la somma totale calcolata con la terza strategia e' 55
Sono il processo 2, la somma totale calcolata con la terza strategia e' 55
Sono il processo 3, la somma totale calcolata con la terza strategia e' 55
Sono il processo 0, la somma totale calcolata con la terza strategia e' 55
Sono il processo 1, la somma totale calcolata con la terza strategia e' 55
Sono il processo 2, la somma totale calcolata con la terza strategia e' 55
Sono il processo 3, la somma totale calcolata con la terza strategia e' 55
Strategia usata: 3,
Numero di input: 10,
Numero di processori:4,
Tempo impiegato: 5.955696e-05
```

5 Analisi delle performance

In base allo studio teorico che abbiamo fatto del problema e delle considerazioni fatte precedentemente, a grandi linee ci aspettiamo che:

- su n piccoli di input, in tutte le strategie pesi più la comunicazione che il calcolo, portando quindi a risultati pessimi al crescere del numero di processori;
- su n grandi, la comunicazione impatti di meno (soprattutto nel caso delle strategie II e III) sul tempo di esecuzione, permettendo di vedere tempi di molto ridotti al crescere del numero di processori;
- fissato il numero di processori, al crescere di n si abbia un aumento dell'efficienza;
- la strategia I dia risultati peggiori delle altre 2, dato il pessimo utilizzo che si fa dei processori, non sfruttandoli tutti a sufficienza.

5.1 Risultati in forma tabellare e grafici

Riportiamo tutti i risultati ricavati, in forma tabellare, con dei grafici per illustrare i casi notevoli.

5.1.1 Tempi d'esecuzione

Numero di processori	Numero di input				
	100	**1000**	**1E+04**	**1E+05**	**1E+06**
1	1,215935E-06	4,911423E-06	3,964901E-05	3,951311E-04	3,986192E-03
2	1,926422E-05	5,459785E-06	6,105900E-05	2,390385E-04	2,059317E-03
3	4,391670E-05	4,718304E-05	5,915165E-05	1,800776E-04	1,394200E-03
4	4,115105E-05	7,307529E-05	8,084774E-05	1,732826E-04	1,087403E-03
5	5,655289E-05	6,029606E-05	6,799698E-05	1,565456E-04	8,834124E-04
6	5,321503E-05	9,050369E-05	8,254051E-05	1,748800E-04	7,953167E-04
7	1,902676E-03	1,728368E-03	2,007699E-03	2,081037E-03	2,573109E-03
8	7,500648E-05	7,362366E-05	6,217957E-05	1,801729E-04	6,542921E-04

Table 1: Tempi di esecuzione con strategia I

N processori	Numero di input				
	100	**1000**	**1E+04**	**1E+05**	**1E+06**
1	1,215935E-06	5,197525E-06	4,003048E-05	3,949642E-04	3,978157E-03
2	1,950264E-05	2,105236E-05	6,289482E-05	2,390862E-04	2,061296E-03
4	2,090931E-05	7,061958E-05	4,823208E-05	1,917362E-04	1,102209E-03
8	5,559921E-05	7,250309E-05	6,380081E-05	1,394033E-04	5,891800E-04

Table 2: Tempi di esecuzione con strategia II

N processori	Numero di input				
	100	**1000**	**1E+04**	**1E+05**	**1E+06**
1	1,192093E-06	6,103516E-06	4,026890E-05	3,939152E-04	3,972483E-03
2	8,106232E-06	8,726120E-06	4,870892E-05	2,242327E-04	2,063036E-03
4	5,767345E-05	6,039143E-05	6,935596E-05	2,044439E-04	1,100039E-03
8	1,183987E-04	1,140594E-04	1,375437E-04	2,271414E-04	6,751537E-04

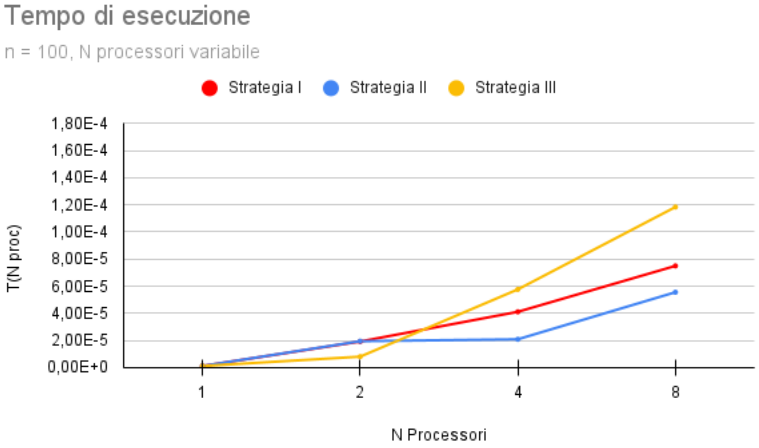
Table 3: Tempi di esecuzione con strategia III

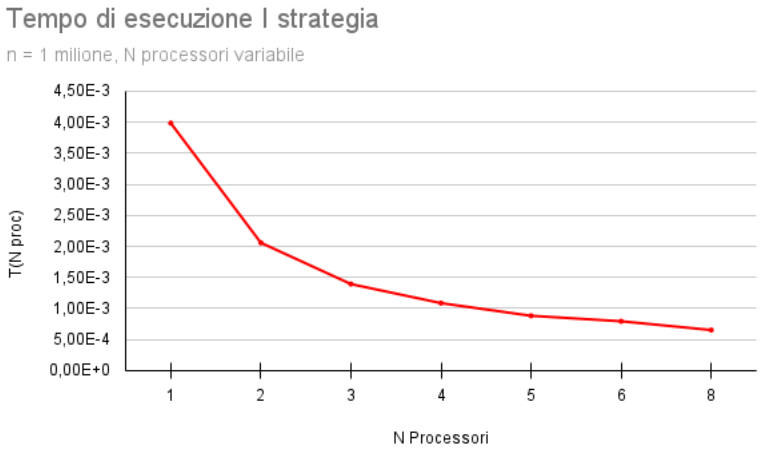
La prima cosa che salta all'occhio sono i tempi di esecuzione ottenuti con la prima strategia utilizzando 7 processori, che risultano 10 o 100 volte più grandi dei tempi ottenuti con un numero diverso di processori.
Purtoppo non c'è alcuna spiegazione teorica per questo imprevisto, ma essendo stato riscontrato da altri studenti in fase di testing, crediamo sia un difetto del cluster, per cui non terremo in considerazione i risultati ottenuti facendo uso di 7 processori.

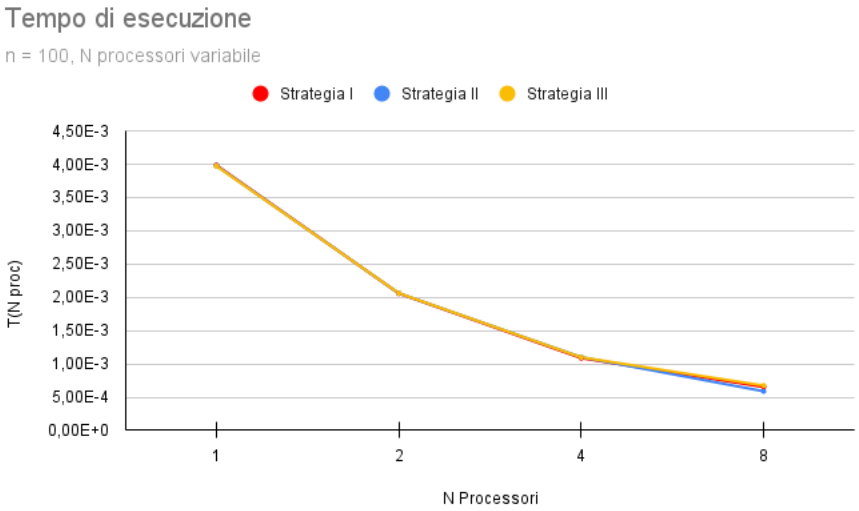
Possiamo inoltre notare che come previsto, su n bassi, come 100, abbiamo un aumento del tempo di esecuzione al crescere del numero di processori. Sono presenti numerosi valori irregolari che non permettono di derivare un trend monotono in tutti i test in cui si fa uso di meno di un milione di numeri, causati probabilmente da irregolarità delle comunicazioni tra i processori all'interno del cluster (Dato che al crescere del numero di input, il tempo di comunicazione ha sempre meno impatto in percentuale).

Come ci aspettavamo, avendo su n=100 un notevole impatto il tempo di comunicazione, al crescere del numero di processori la performance peggiora e la strategia II è quella che si comporta meglio, in quanto riduce al minimo le comunicazioni sequenziali.
Vediamo ora il comportamento con n = 1 milione:


Tempo di esecuzione I strategia
n = 100, N processori variabile
9,00E-5
8,00E-5
7,00E-5
6,00E-5
5,00E-5
4,00E-5
3,00E-5
2,00E-5
1,00E-5
0,00E+0
T(N proc)
1
2
3
4
5
6
8
N Processori

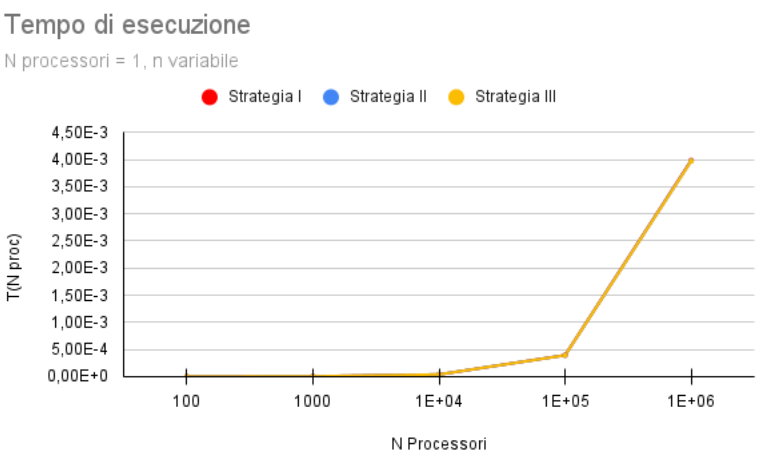

Tempo di esecuzione
n = 100, N processori variabile
Strategia I
Strategia II
Strategia III
1,80E-4
1,60E-4
1,40E-4
1,20E-4
1,00E-4
8,00E-5
6,00E-5
4,00E-5
2,00E-5
0,00E+0
T(N proc)
1
2
4
8
N Processori

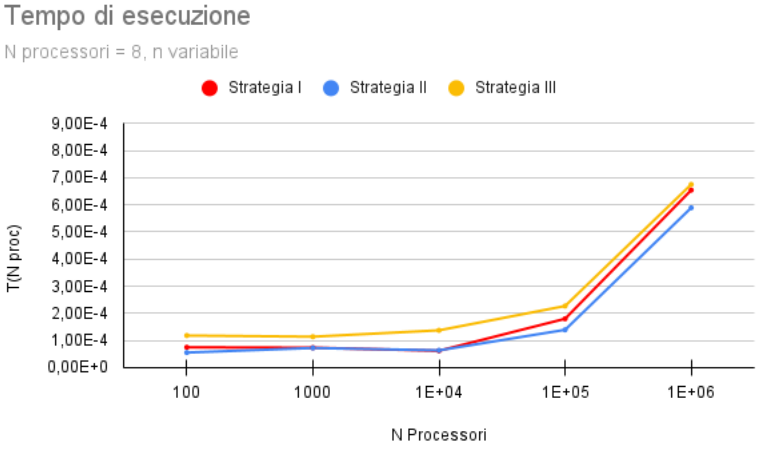




Possiamo notare con piacere che al crescere del numero di processori, il tempo di esecuzione diminuisca sensibilmente. La differenza tra i tempi di esecuzione delle strategie è minima (Distinguibili solo con una precisione di 10^-4), ma possiamo notare che la strategia II è la più performante.
Vediamo ora i risultati al variare del numero di input, fissando il numero di processori, in particolare con i due casi estremi: 1 ed 8 processori.

Nel caso di utilizzo di 1 processore, le strategie lavorano in modo equivalente, come risulta evidente dai dati.
Lavorando su 8 processori saltano all'occhio le performance migliori della strategia II, come ci aspettavamo per via dei presupposti teorici alla base degli algoritmi delle 3 strategie.
E' inaspettato invece il comportamento della strategia III, che risulta addirittura peggiore della strategia I, nonostante il minor numero di comunicazioni sequenziali in linea teorica. Possiamo però spiegarci questo comportamento pensando al numero elevato di comunicazioni che devono essere fatte in questa strategia, che seppur parallele, possono avere una durata variabile e una comunicazione particolarmente lenta potrebbe aver trascinato poi tutte le altre in parallelo (Dato che per procedere allo step successivo devono essere finite tutte le comunicazioni coinvolte del precedente step).
La differenza in performance tra i due grafici la vedremo meglio utilizzando la misura dello Speedup.


Tempo di esecuzione
N processori = 1, n variabile
Strategia I
Strategia II
Strategia III
4,50E-3
4,00E-3
3,50E-3
3,00E-3
2,50E-3
2,00E-3
1,50E-3
1,00E-3
5,00E-4
0,00E+0
T(N proc)
100
1000
1E+04
1E+05
1E+06
N Processori


Tempo di esecuzione
N processori = 8, n variabile
Strategia I
Strategia II
Strategia III
9,00E-4
8,00E-4
7,00E-4
6,00E-4
5,00E-4
4,00E-4
3,00E-4
2,00E-4
1,00E-4
0,00E+0
T(N proc)
100
1000
1E+04
1E+05
1E+06
N Processori

5.1.2 Speedup

Ricordiamo che lo Speedup è calcolabile nel seguente modo:

$$S(N) = \frac{T(1)}{T(N)} \tag{2}$$

esprime quindi la bontà del tempo d'esecuzione su N processori rispetto al tempo ottenuto lavorando sequenzialmente.
In particolare T(1) si può calcolare in due modi:

1. Calcolando il tempo d'esecuzione del migliore algoritmo possibile sequenziale per risolvere il problema d'interesse;
2. Calcolando il tempo d'esecuzione dell'algoritmo parallelo scritto, ponendo il numero di processori da utilizzare ad 1

Nel primo caso lo speedup esprimerà quanto è più efficace la soluzione parallela rispetto a quella sequenziale, ma è più difficile da calcolare, perchè dovremo scrivere effettivamente un algoritmo che sappiamo essere il migliore a risolvere il problema sequenzialmente.
Nel secondo caso invece lo speedup esprimerà quanto si comporta più efficacemente il nostro codice se svolto su N processori piuttosto che su un processore, ed è la misura che più ci interessa, quindi utilizzeremo questa.

Essendo il tempo di esecuzione con N processori ideale

$$T_{ideale}(N) = \frac{T(1)}{N} \tag{3}$$

lo speedup ideale, teoricamente irraggiungibile, è

$$S_{ideale}(N) = \frac{T(1)}{\frac{T(1)}{N}} = N \tag{4}$$

ci interessa quindi anche quanto è vicino S(N) a N stesso, misura che effettueremo più nel dettaglio nel prossimo paragrafo

Numero di input

Numero di processori	**100**	**1000**	**1E+04**	**1E+05**	**1E+06**
1	1	1	1	1	1
2	0,063	0,900	0,649	1,653	1,936
3	0,028	0,104	0,670	2,194	2,859
4	0,030	0,067	0,490	2,280	3,666
5	0,022	0,081	0,583	2,524	4,512
6	0,023	0,054	0,480	2,259	5,012
7	6,390657E-04	2,841654E-03	0,020	0,190	1,549
8	0,016	0,067	0,638	2,193	6,092

Table 4: Speedup con strategia I

Numero di input

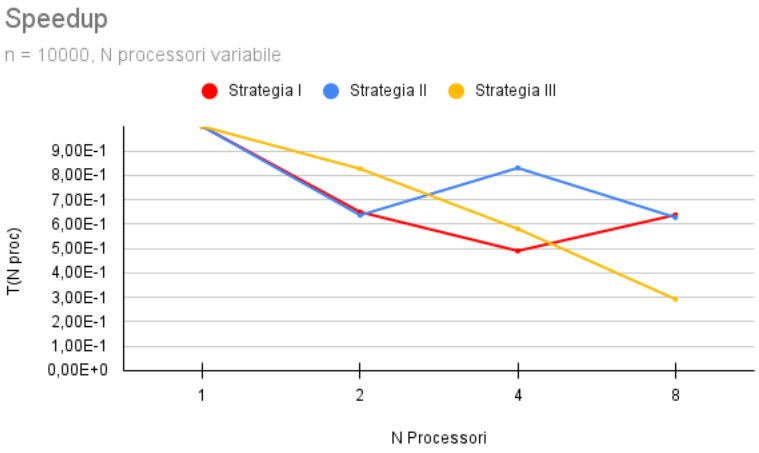
N processori	**100**	**1000**	**1E+04**	**1E+05**	**1E+06**
1	1	1	1	1	1
2	0,062	0,247	0,636	1,652	1,930
4	0,058	0,074	0,830	2,060	3,609
8	0,022	0,072	0,627	2,833	6,752

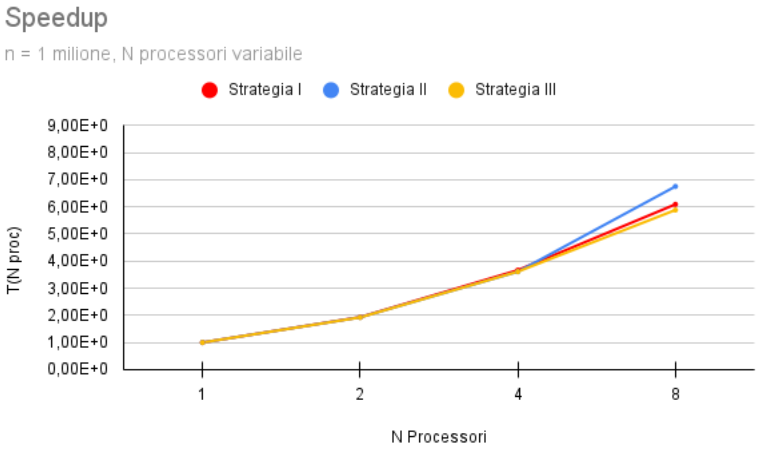
Table 5: Speedup con strategia II

N processori \ Numero di input	100	1000	1E+04	1E+05	1E+06
1	1	1	1	1	1
2	0,147	0,699	0,827	1,757	1,926
4	0,021	0,101	0,581	1,927	3,611
8	0,010	0,054	0,293	1,734	5,884

Table 6: Speedup con strategia III

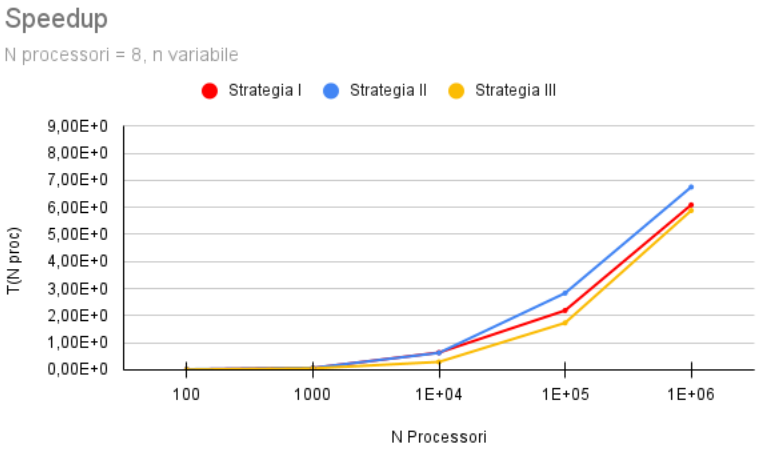
Analizziamo i grafici dei risultati più interessanti:





Possiamo notare come perfino con n = 10000, abbiamo ancora un trend negativo nello speedup: lavorare con più processori è meno conveniente che lavorare sequenzialmente, in particolare con la strategia III.
Sono invece positivi i risultati con n = 1 milione, che riportano un risultato più standard, cioè che lo speedup cresca all'aumentare del numero di processori.
Risulta evidente come su grandi input, non faccia la differenza la strategia scelta se non lavorando con tanti processori (Nel nostro caso la differenza è sostanziale utilizzandone 8, ma possiamo aspettarci un

trend simile utilizzandone ancora di più) Vediamo il comportamento dello speedup su 8 processori, al variare di n:



Possiamo notare uno speedup inferiore a 1 fino a $n = 10^4$, che indica che su quantità piccole di input non ci conviene assolutamente lavorare con più processori, a prescindere dalla strategia adoperata, e invece una crescita elevata al crescere di n per $n > 10^4$, con un notevole distacco tra le strategie. Risulta ancora una volta evidente il beneficio su grandi input e utilizzando molti processori nell'utilizzare la seconda strategia.

5.1.3 Efficienza

Analizziamo ora l'efficienza, ovvero la sopracitata vicinanza dello speedup calcolato allo speedup ideale, formalmente:

$$E(N) = \frac{S(N)}{S_{ideale}(N)} < 1 \tag{5}$$

Numero di input

Numero di processori	100	1000	1E+04	1E+05	1E+06
1	1,000	1,000	1,000	1,000	1,000
2	0,032	0,450	0,325	0,827	0,968
3	0,0092	0,035	0,223	0,731	0,953
4	0,0074	0,017	0,123	0,570	0,916
5	0,0043	0,016	0,117	0,505	0,902
6	0,0038	0,0090	0,0801	0,377	0,835
7	9,13E-05	4,06E-04	0,0028	0,027	0,221
8	0,0020	0,0083	0,0797	0,274	0,762

Table 7: Efficienza con strategia I

Numero di input

N processori	100	1000	1E+04	1E+05	1E+06
1	1,000	1,000	1,000	1,000	1,000
2	0,031	0,123	0,318	0,826	0,965
4	0,015	0,018	0,207	0,515	0,902
8	0,0027	0,0090	0,078	0,354	0,844

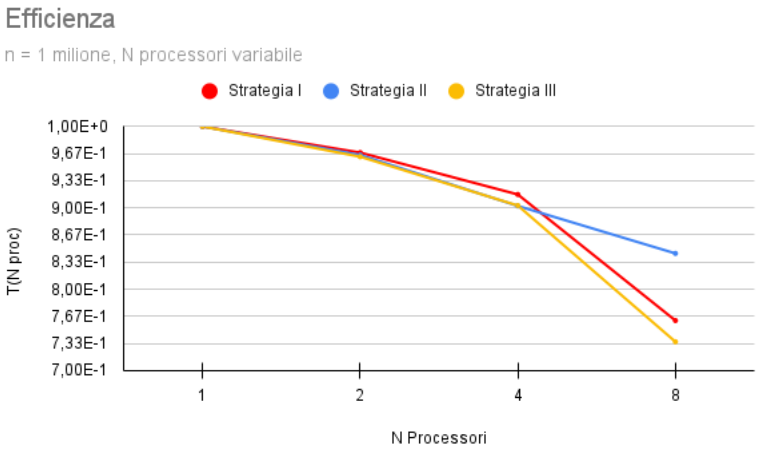
Table 8: Efficienza con strategia II

Numero di input

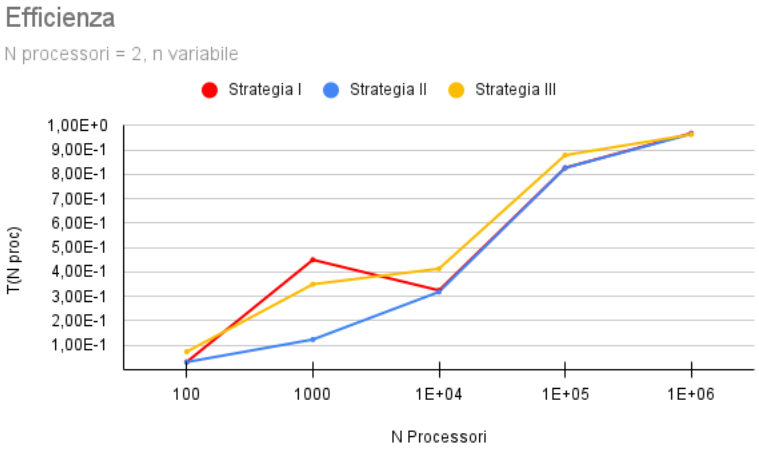
N processori	100	1000	1E+04	1E+05	1E+06
1	1	1	1	1	1
2	0,074	0,350	0,413	0,878	0,963
4	0,0052	0,025	0,145	0,482	0,903
8	0,0013	0,0067	0,037	0,217	0,735

Table 9: Efficienza con strategia III

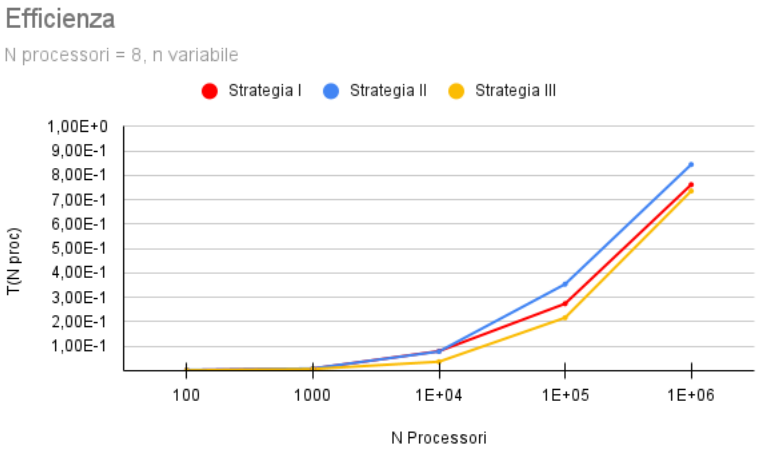
Vediamo ora nel dettaglio i grafici raffiguranti i dati più importanti:



Possiamo notare da questo grafico come solo utilizzando 8 processori ci sia una differenza netta nell'efficienza delle strategie e che utilizzando 2 processori, su input così grandi, l'efficienza sia molto vicina all'ideale.
Per questo esaminiamo meglio i risultati al variare di n, fissando il numero di processori a 2 ed 8.



Notiamo come su input grandi, in particolare per $n >= 10^5$ l'efficienza sia elevatissima nell'uso di due processori, con dei risultati leggermente migliori per la terza strategia.
Le strategie I e II su 2 processori si comportano sostanzialmente allo stesso modo, eseguendo una singola iterazione che porta la somma locale di P1 a P0. Il risultato notevolmente differente con $n = 1000$ è probabilmente legato ad una stranezza tecnica del cluster ed è inaspettato.



In questo grafico possiamo notare un notevole incremento dell'efficienza al passare da $n = 10^5$ a $n = 10^6$, con performance leggermente migliori della strategia II.
Purtroppo non siamo riusciti ad effettuare delle prove con n ancora maggiori ma possiamo aspettarci un trend simile, con un avvicinamento rapido all'efficienza ideale.

Dai risultati analizzati e commentati possiamo quindi concludere che ad eccezione di pochi casi isolati, ciò che ci aspettavamo per le nostre analisi teoriche si è effettivamente verificato e che a grandi linee, aumentare parallelamente il numero di input ed il numero di processori utilizzati ci porta un'elevata efficienza.

6 Codice

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "mpi.h"

int random_int();

int main(int argc, char *argv[]) {
    int menum, nproc, tag, i, start_pos, rest, strat;
    long n, nloc, sum, parz_sum, tmp;
    int *x, *xloc;
    int iter;
    double log_nproc;
    double start_time, end_time, elapsed_time, total_time;
    double time_mean = 0.0;
    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &menum);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);

    for (iter = 0; iter < 10; iter++) { //Eseguiamo tutto 10 volte per
        avere una media dei tempi

        //Acquisizione dell'input
        if (menum==0){
            if (argc < 3) {
                perror("Numero di argomenti insufficiente");
                return 1;
            }
            n = atoi(argv[1]);
            x = (int*) malloc(n*(sizeof(int)));
            if (n <= 20) {
                if (argc != n+3) {
                    perror("Numero di argomenti errato!");
                    return 1;
                }
                for (i = 0 ; i < n; i++)
                    x[i] = atoi(argv[2+i]);


                strat = atoi(argv[n+2]);
            }
            else {
                for (i = 0; i < n; i++)
                    x[i] = random_int();

                strat = atoi(argv[2]);
```

```
    }

    if (n <= 0) {
        perror("Numero di input non valido!");
        return 1;
    }

    if (n < nproc) {
        perror("Numero di input minore del numero di processori
            utilizzati");
        return 1;
    }
}

//Comunicazione dei dati
MPI_Bcast(&n, 1, MPI_LONG, 0, MPI_COMM_WORLD);
MPI_Bcast(&strat, 1, MPI_INT, 0, MPI_COMM_WORLD);

//Eventuale cambio di strategia
if (strat == 2 || strat == 3) {
    if ((nproc & (nproc-1)) != 0) { //numero di processi non
        potenza di 2, sfruttiamo il codice binario particolare
        delle potenze di 2, composto da tutti 0 eccetto per un 1
    //Se nroc    potenza di 2, nproc-1 sar  quindi composto da
        tutti 1 fino ad uno 0 laddove nproc aveva il suo unico 1.
        L'and bit a bit quindi ci restituir  0.
        if (menum == 0)
            printf("E' stata richiesta la strategia %d ma il
                numero di processi richiesta non e' potenza di due
                , si prosegue con la strategia 1\n", strat);
        strat = 1;
    }
}

//Comunicazione dei dati

nloc=n/nproc; //Dividiamo il numero di input per il numero di
    processori per distribuirli equamente, dobbiamo per  stare
    attenti al resto, come vediamo in seguito
rest=n%nproc;
if (menum<rest) nloc=nloc+1; //Fino al rest_esimo processore
    dobbiamo dare un numero in input in pi

if (menum == 0) {
    xloc = x;
    tmp = nloc;
    start_pos = 0;
    for (i = 1; i < nproc ; i++) {
        start_pos += tmp;
        if (i == rest) //Arrivati al rest_esimo processore
            rimuoviamo il numero extra
            tmp--;
```

```
            tag = 22+i;
            MPI_Send(&x[start_pos], tmp, MPI_INT, i, tag,
                MPI_COMM_WORLD);
        }
    }
    else {
        xloc = (int*)malloc(nloc*sizeof(int));
        tag = 22 + menum;
        MPI_Recv(xloc, nloc, MPI_INT, 0, tag, MPI_COMM_WORLD, &status
            );
    }

    //Calcolo di potenze e logaritmi, per efficienza negli algoritmi
        delle strategie seguenti
    int pow[nproc];
    int curr_pow = 1;

    for (i = 0; i < nproc; i++) {
        pow[i] = curr_pow;
        curr_pow*=2;
    }

    log_nproc = log(nproc)/log(2);

    MPI_Barrier(MPI_COMM_WORLD);
    start_time = MPI_Wtime(); //Prendiamo il tempo di inizio, dopo
        aver sincronizzato i vari processi con la precedente Barrier

    //Calcolo locale
    sum = xloc[0];
    for (i = 1; i < nloc; i++)
        sum += xloc[i];

    //Comunicazione
    switch(strat) {
        case 1: //I strategia
            if (menum == 0) {
                for (i = 1; i < nproc; i++) {
                    tag = 80+i;
                    MPI_Recv(&parz_sum, 1, MPI_LONG, i, tag,
                        MPI_COMM_WORLD, &status);
                    sum+=parz_sum;
                }
                end_time = MPI_Wtime();
                printf("La somma totale calcolata con la prima
                    strategia e' %ld\n", sum);
            }
            else {
                tag = 80+menum;
                MPI_Send(&sum, 1, MPI_LONG, 0, tag, MPI_COMM_WORLD);
                end_time = MPI_Wtime(); //Otteniamo il tempo di fine
            }
```

```
        break;

    case 2: //II strategia
        for (i = 0; i < log_nproc; i++) { //Avremo log2(nproc)
            passi, dato che ad ogni passo accumuliamo la somma di
            2^i processi
            if (menum % pow[i] == 0) { //All'i-esimo passo
                parteciperanno solo i processi il cui rank
                divisibile per 2^i
                if (menum % pow[i+1] == 0) {//I processi che
                    riceveranno i dati sono quelli che
                    parteciperanno alla prossima iterazione, cio
                     con rank divisibile per 2^(i+1)
                    tag = 120 + menum + pow[i]; //menum + 2^i ci
                        dar  il rank del processo accoppiato che
                        invier  i dati
                    MPI_Recv(&parz_sum, 1, MPI_LONG, menum+pow[i
                        ], tag, MPI_COMM_WORLD, &status);
                    sum+=parz_sum;
                }
                else {
                    tag = 120 + menum;
                    MPI_Send(&sum, 1, MPI_LONG, menum-pow[i], tag
                        , MPI_COMM_WORLD);
                    break;
                }
            }
        }
        end_time = MPI_Wtime();//Otteniamo il tempo di fine
        if (menum == 0)
            printf("La somma totale calcolata con la seconda
                strategia e' %ld\n", sum);
        break;

    case 3: //III strategia
        for (i = 0; i < log_nproc; i++) {
            if (menum % pow[i+1] < pow[i]) { //Distinguiamo i
                processi per formare le coppie, ogni processo
                comunicher  con quello distante 2^i
                tag = 200 + menum + pow[i]; //Invia prima il
                    secondo processo, con rank menum+2^i per il
                    ricevente
                MPI_Recv(&parz_sum, 1, MPI_LONG, menum+pow[i],
                    tag, MPI_COMM_WORLD, &status);
                tag = 200 + menum;
                MPI_Send(&sum, 1, MPI_LONG, menum+pow[i], tag,
                    MPI_COMM_WORLD);
                sum+=parz_sum;
            }
            else {
                tag = 200 + menum;
                MPI_Send(&sum, 1, MPI_LONG, menum-pow[i], tag,
```

```
                        MPI_COMM_WORLD);
                    tag = 200 + menum - pow[i];//Invia adesso il
                        secondo processo, con rank menum-2^i per il
                        ricevente
                    MPI_Recv(&parz_sum, 1, MPI_LONG, menum-pow[i],
                        tag, MPI_COMM_WORLD, &status);
                    sum+=parz_sum;
                }
            }
            end_time = MPI_Wtime(); //Otteniamo il tempo di fine
            printf("Sono il processo %d, la somma totale calcolata
                con la terza strategia e' %ld\n", menum, sum);
            break;

        default:
            if (menum == 0)
                perror("Input della strategia errato");
            return 1;

    }

    if (menum != 0)
        free(xloc);
    else
        free(x);

    elapsed_time = end_time - start_time;
    MPI_Reduce(&elapsed_time, &total_time, 1, MPI_DOUBLE, MPI_MAX, 0,
        MPI_COMM_WORLD); //Prendiamo il massimo dei tempi calcolati
        dai vari processi

    if (menum == 0) {
        time_mean+=total_time;
    }

}

if (menum == 0) {
    time_mean/=10; //Facciamo la media dei risultati delle 10
        iterazioni del processo
    printf("Strategia usata: %d,\nNumero di input: %ld,\nNumero di
        processori:%d,\nTempo impiegato: %e\n\n\n\n", strat, n, nproc,
        time_mean);
}

MPI_Finalize();

return 0;

}

int random_int() {
```

```
    srand(time(NULL));
    return rand()%100;
}
```