ELABORATO I

Somma di N numeri in ambiente parallelo

UNIVERSITÀ DEGLI STUDI DI NAPOLI "FEDERICO II"
ROBERTO BASILE GIANNINI
N97000340

Indice

Descrizione del problema

Si vuole implementare un algoritmo per il calcolo della somma di numeri reali, in ambiente di calcolo parallelo su architettura MIMD a memoria distribuita, utilizzando la libreria MPI. Gli input del problema, oltre al numero N di addendi e agli addendi stessi, riguardano la strategia di calcolo da adottare (scelta tra la I, II e la III) e il/i processore/i destinati alla stampa del risultato. Gli addendi destinata alla somma dovranno essere generati random qualora il numero N dovesse risultare strettamente maggiore di 20. La stampa da parte di tutti i processori avviene quando il rispettivo parametro ID è pari a -1, ma in particolare viene richiesto che ciò sia possibile solo quando la strategia adottata sia la III. Qualora venisse richiesta la strategia II o III, è indispensabile constatarne l'applicabilità: qualora non fosse applicabile, la strategia da applicare sarà la I.

Input e Output

Il software richiede in ingresso i seguenti parametri:

- *int N*, il numero degli addendi da sommare, necessariamente strettamente maggiore di 0;
- *int N_strat*, il numero della strategia scelta, tra 1, 2 e 3;
- *int ID_stamp*, l'identificativo del processore che dovrà stampare a video (-1 se dovranno stampare tutti), dunque contenuto nell'intervallo *[-1, numero_proc – 1*];
- *float* $a_1, \ldots, a_N$, gli *N* addendi di tipo float *eventualmente* riportati in input.

Gli input vengono specificati tramite file *.pbs*, indicandoli con lo stesso ordine col quale sono stati presentati:

$$N\ N\_strat\ ID\_stamp\ a_1, \ldots, a_N$$

L'eventuale gestione degli errori legati alla coerenza dei parametri inseriti (come l'esistenza o meno della strategia scelta, Il numero N effettivamente pari al numero di addendi inseriti, e così via) viene garantita da specifici controlli effettuati a monte delle operazioni che li coinvolgono: nella sezione dedicata agli Indicatori di errore se ne parlerà in maggiore dettaglio. Gli elementi di tipo float da sommare possono non essere inseriti, a patto di accettare che la loro generazione venga effettuata in maniera randomica: oltre alla politica di generare i numeri random per N > 20, la loro generazione randomica avviene anche nel caso in cui N dovesse risultare più grande della quantità di elementi riportati.

Gli output a video sono i seguenti:

- *float sum*, la somma totale calcolata secondo la strategia richiesta e stampata a video;

- *double time*, ovvero il tempo impiegato da ogni singolo processore per il proprio coinvolgimento nel calcolo della somma (compatibilmente alla strategia adottata);

- *double timemax*, il massimo tempo impiegato tra i time prodotti dai processori;

- eventuali messaggi di errore.

L'interpretazione degli output è abbastanza immediata dal momento che ogni parametro di uscita viene accompagnato da un messaggio che ne esplicita la natura. In più, per ogni messaggio di errore è previsto uno specifico codice, da *E1* ad *E7*.

Descrizione dell’algoritmo

L’idea di base è quella di sfruttare gli strumenti messi a disposizione dalla libreria MPI al fine di realizzare un algoritmo parallelo tale da garantire alte prestazioni al crescere della dimensione del problema. La comunicazione tra i processori, fondamentale in fase di distribuzione dei dati e delle misure, avviene sfruttando le funzioni di comunicazione (collettiva ed uno a uno) della libreria MPI. L’algoritmo implementato è diviso in 9 blocchi complessivi, ognuno avente uno specifico compito. Se si volesse rappresentare in maniera estremamente astratta la soluzione, la rispettiva descrizione in pseudocodice sarebbe la seguente:

```
INIZIO

//0.
    Inizializzazione dell'ambiente

//1.
    Lettura e distribuzione addendi

//2.
    Verifica consistenza degli input di controllo

//3.
    Calcolo strutture dati di supporto

//4.
    Calcolo somme parziali

//5.
    Calcolo somma totale

//6.
    Stampa del risultato

//7.
    Stampa dei tempi
```

```
26.
27. //8.
28.     Finalizzazione
29.
30. FINE
```

Nel *passo 0*, oltre alla dichiarazione di tutte le variabili che verranno utilizzate, avviene l'inizializzazione dell'ambiente MPI per mezzo delle rispettive funzioni di ambiente: a tutti i processori del communicator verrà assegnato un identificativo di cui verranno a conoscenza, insieme al numero di processori stessi. Questi parametri risulteranno indispensabili per la corretta individuazione dei processori demandati a specifici compiti (per esempio, il processore P0 si occuperà della lettura e distribuzione degli addendi), per la comunicazioni, per la corretta allocazione delle strutture dati che ospiteranno gli elementi da sommare e così via.

Nel *passo 1*, da parte di P0, avviene la lettura e la distribuzione degli addendi, a partire dal parametro N contenente il numero dichiarato di addendi disponibili. Considerando che potrebbe esserci inconsistenza tra il valore di N e l'effettivo numero di addendi posti in input, P0 deve verificarne la coerenza ed eventualmente gestire l'errore. Inoltre, P0 dovrà verificare l'effettiva validità del parametro (per maggiori dettagli sulla gestione degli errori si rimanda alla sezione Indicatori di errore). Letto e gestito N, P0 provvederà alla lettura e distribuzione degli addendi tra i diversi processori per mezzo di una operazione di send bloccante. Analogamente, tutti i processori diversi da P0 riceveranno gli addendi per mezzo di una recieve anch'essa bloccante. La giustificazione di avere operazioni di comunicazione bloccanti sta nel fatto che per procedere con l'esecuzione dell'algoritmo è necessario che tutti i processori siano muniti dei propri addendi. Un occhio di riguardo merita il caso in cui dovessimo ottenere del resto, ovvero quando si ha:

```
1.  ...
2.  ( N % Numero_proc ) ≠ 0
3.  ...
```

Questa condizione è rappresentante della dinamica nella quale il numero di addendi è tale da non poterli equamente distribuire in maniera esatta tra tutti i processori, creando uno sbilanciamento

che diventa lieve se opportunamente gestito. Per esempio, potremmo trovarci al cospetto di 10 addendi e 8 processori: certamente ne verrebbe distribuito almeno 1 ad ogni processore, ma otterremmo un resto di 2 addendi che dovranno in qualche modo essere distribuiti. Tenere traccia del resto ci torna utile per minimizzare lo sbilanciamento. Consideriamo lo scenario appena mostrato e la seguente soluzione:

```
1.  ...
2.  resto = ( N % Numero_proc ) ≠ 0
3.  temp = (N / Numero_proc)
4.  start = 0
5.
6.  Per ogni i da 1 a Numero_proc-1
7.      start = start + temp
8.
9.      Se i è uguale al resto
10.         temp = temp - 1
11.
12.     Spedisci addendi[start], ..., addendi[start+temp-1] a proc i
13.
14. ...
```

Nel nostro caso, il resto è uguale a 2 e *temp = 2*:

- Per *i = 1*, abbiamo *start = 2*. Osserviamo subito che gli addendi *addendi[0]* e *addendi[1]* non verranno distribuiti da P0, dal momento che faranno parte del suo insieme di addendi da sommare. Il controllo ci dice che *i* ≠ *resto* e quindi che temp rimarrà invariato. Verranno spediti a P1 gli elementi: *addendi[2]* e *addendi[3]*;

- Per *i = 2*, *start* sarà pari a *start* + *temp*, ovvero 4 (l'elemento successivo all'ultimo spedito). 10 In questo caso *i = resto* e dunque *temp* verrà decrementato: abbiamo *temp = 1*. Dunque, a P2 verrà spedito un solo elemento, ovvero *addendi[start]*, essendo *addendi[start+temp-1]* pari propri ad *addendi[start]* (basta sostituire temp con 1). Tutti i processori successivi fino a P7 riceveranno un solo addendo, trovandoci nella situazione in cui solo P0 e P1 avranno gli elementi "in eccesso", equamente distribuiti tra loro. Se il resto fosse stato 3, avremmo avuto che anche a P2 sarebbero stati recapitati 2 elementi, per poi spedirne solo uno ai successivi. È banale osservare che in assenza di resto, ovvero con *resto = 0*, la condizione *i = resto* non si verifica mai, non comportando mai un

decremento di *temp*: ciò è coerente con la soluzione, dal momento che in assenza di resto non è necessario prevedere alcuno sbilanciamento.

Nel *passo 2* avviene il controllo degli input di controllo. Per input di controllo ci si riferisce al numero di strategia N_strat richiesta e all'identificato ID_stamp del processore che si vuole stampi il risultato. Il controllo viene fatto sulla base dei seguenti criteri:

- *N_strat* dovrà essere 1, 2 oppure 3, non essendo previste altre strategie;
- *N_strat* potrà essere 2 o 3 solo quando il numero di addendi è potenza di 2;
- *ID_stamp* dovrà essere compreso in *[-1, Numero_proc -1],* in particolare quando è -1 viene richiesto che tutti i processori stampino la somma totale;
- *ID_stamp* può essere -1 solo se la strategia adottata è la III.

Nel *passo 3* vengono calcolate le variabili di supporto, ovvero l'indice *index* pari al $log_2(Numero\_proc)$, il vettore contenente gli elementi 2^i e il vettore contenente gli elementi 2^{i+1}, con $i = 0$ *a index-1*. Questi parametri risultano funzionali all'applicazione delle strategie II e III. Si preferisce calcolarli a monte delle somme (parziali e totale) per evitare di condizionare le prestazioni in fase di calcolo.

Nel *passo 4* vengono effettuate le somme locali da parte di tutti i processori.

Nel *passo 5* viene effettuata la somma totale. In particolare, sulla base del parametro *N_strat* viene scelta la strategia di calcolo da seguire. Nel caso della strategia I, i processori che hanno calcolato le somme parziali spediscono il proprio risultato ad un processore designato, in particolare *PID_stamp*. Il numero di passi sarà pari al numero di processori – 1. Nel caso della strategia II, ad ogni passo di comunicazione vengono stabiliti i partecipanti al passo e discriminati tra *sender* e *reciever* delle somme parziali. Le somme parziali non vengono tutte spedite al processore designato per il calcolo totale, ma man mano calcolate tra coppie di processori, riducendo il numero di passi nel $log_2(Numero\_proc)$. In questo caso, così come in quello precedente, solo un processore avrà a disposizione la somma totale. Nel caso della

strategia III, tutti i processori si scambiano le somme parziali tra loro, potendo dunque ognuno di loro calcolare la somma totale: ad ogni passo, ogni processore effettuerà un'operazione di *send* seguita da un'operazione di *recieve*. L'ordine non è casuale, ma tale da evitare condizioni di deadlock.

Nel *passo 6* viene effettuata la stampa della somma totale (distinguendo opportunamente i casi in base ad *ID_stamp*), mentre nel passo 7 vengono stampati i tempi. Infine, nel passo 8 avvengono le operazioni di finalizzazione, come la deallocazione delle strutture dati dinamicamente allocate e la terminazione dell'ambiente *MPI*.

Di seguito viene descritta in maggior dettaglio la soluzione implementata, mantenendo comunque una forte astrazione:

```
1.  //I codici di errore sono indicati nella sezione relativa agli Indicatori di errore.
2.  //La stringa di input si presenta come N, N_strat, ID_stamp, A1, ..., AN
3.
4.  INIZIO
5.  //0.
6.      Inizializzazione ambiente
7.
8.  //1.
9.      Se sei P0
10.         Leggi e controlla N
11.
12.         Se N non è valido
13.             Stampa errore E1, E2, E3
14.             Gestisci rispettivo errore
15.
16.         Leggi/Genera addendi
17.
18.     Distribuzione in BCast di N
19.     Inizializzazione strutture dati e dimensioni
20.
21.     Distribuzione addendi con gestione resto
22. //
23.
24. //2.
25.     Lettura e controllo ID_stamp
26.
```

```
    Se ID_stamp non è consistente
        Se sei P0
            Stampa errore E4
        ID_stamp = 0

    Lettura e controllo N_strat

    Se N_strat non è consistente
        Se sei P0
            Stampa errore E5
        N_strat = 1

    Altrimenti
        Se N_strat è II o III
            Se N_strat non è potenza di 2
                Se sei P0
                    Stampa errore E6
                N_strat = 1

        Se ID_stamp = -1 e N_strat non è III
            Se sei P0
                Stampa errore E7
            N_strat = 3
//

//3.
    Se N_strat è II o III
        Calcolo variabili di supporto
//

    Sincronizza processori

    Inizio misura tempo (t0)
//4.
    Calcolo somma parziale sum_parz
//

//5.
    Se N_strat è 1
        Somma totale con strategia 1

    Se N_strat è 2
        Somma totale con strategia 2

    Se N_strat è 3
        Somma totale con strategia 3
//
    Fine misura tempo (t1)

    Calcolo time = t1-t0

    Se ID_stamp è -1
        ID_stamp_temp = 0
    Altrimenti
        ID_stamp_temp = ID_stamp

```

```
    Pnproc-1 inviano time in Reduce a PID_stamp_temp

//6.
    Se ID_stamp è -1
        P0, ..., Pnproc-1 stampa somma totale

    Altrimenti
        Se sei PID_stamp
            Stampa somma totale
//

//7.
    Se sei PID_stamp_temp
        Stampa timemax


//8.
     Finalizza e dealloca risorse
     FINE
```

Indicatori di errore

Gli errori previsti vanno da dati di input non validi fino a richieste di stampa non compatibili con la strategia voluta. Si è deciso, indipendentemente dall'errore, di procedere sempre con l'esecuzione del programma, sostituendo gli input non validi con valori compatibili. Non è previsto alcun meccanismo di interruzione forzata dell'applicativo. Il processore identificato con l'ID 0 è stato scelto come segnalatore di errori qualora dovessero presentarsi.

Di seguito vengono riportate le diverse circostanze di errore ed indicata la rispettiva gestione:

- *Numero di addendi N ≤ 20, ma superiore agli effettivi addendi posti in input*

  In questo caso, si è deciso di considerare solo gli addendi effettivamente indicati. Al verificarsi della circostanza, il processore P0 provvede alla segnalazione dell'errore ed alla sua gestione.

```
1.  //ERR: addendi non sufficiente
2.  if ((n > argc-4) && (n <= 20)) {//Numero addenti non sufficienti
3.      printf ("[ERROR E1] Addendi non sufficienti. Devono essere almeno %d. Verranno considerati solo quelli presenti.\n", n);
4.      n = argc-4;
5.  }
```

- *Numero di addendi N non valido*

  Rappresentante il caso in cui N dovesse essere minore strettamente di 1. Si è deciso di porre *N* a 21 e di segnalare a video la circostanza di errore e la relativa gestione. Si pone *N* a 21 per generare automaticamente gli addendi.

```
1.if (n < 1) {//Se N non è valido
2.    printf ("[ERROR
   E2] Numero %d di addendi non valido. Valore default: 21\n", n);
3.    n = 21;
4.}
```

- *Numero N non sufficientemente grande*

  Il processore P0 si limita a segnalare l'errore a video e la decisione di considerare solo i primi *N* addendi posti in input.

```
1. //ERR: N non sufficiente
2. if ((n < argc-4) && (n <= 20)) {
3.     printf ("[ERROR
   E3] N non sufficientemente grande. Verranno considerati solo i primi %d elementi.\n
   ", n);
4. }
```

- *Identificatore del processore di stampa non valido*

  Qualora dovesse essere indicato un valore non compreso nell'intervallo *[-1, numero_processori-1]*. P0 provvede alla segnalazione dell'errore e sceglie se stesso come processore di stampa.

```
1. //ERR: controllo ID_stamp
```

```
2. if (ID_stamp < -1 || ID_stamp >= nproc) {//In caso di ID_stamp non valido
3.     if (menum == 0) {//P0 demandato alla segnalazione di errori
4.         printf ("[ERROR      E4] ID_stamp %d non valido, deve essere compreso tra [-
   1, %d]. Valore default: 0\n", ID_stamp, nproc-1);
5.     }
6.     ID_stamp = 0;
7. }
```

- *Numero di strategia non valido*

  Se la strategia richiesta non rientra tra la I, II e III, P0 segnala l'errore e lascia la strategia di default: 1.

```
1. //ERR: numero di strategia non valido
2. if ((N_strat_temp < 1) || (N_strat_temp > 3)) {
3.     if (menum == 0) {//P0 demandato alla segnalazione di errori
4.         printf("[ERROR
   E5] Strategia %d non valida, possibili solo: 1, 2 o 3. Valore default: 1\n", N_stra
   t_temp);
5.     }
6. }  ...
```

- *Strategia II o III richiesta non applicabile*

  Le strategia II e III sono applicabili solo quando il numero di addendi è potenza di 2. Per questo motivo, se il relativo controllo dovesse fallire, il processore P0 provvederà a segnalare l'errore e lasciare la strategia di default: 1.

```
1. ...
2. else if (N_strat_temp != 1){//Numero strategia valido
3.
4.         //Verifica applicabilità strategia II o III
5.         if((n != 0) && ((n &(n -
    1)) == 0)){ //Se la strategia scelta è la II o III, verifica se N è potenza di 2.

6.                 N_strat = N_strat_temp;
7.         }
8.
```

```
        else {//ERR: non applicabilità strategia 2 o 3
            if(menum == 0) //P0 demandato alla segnalazione di errori
                printf("[ERROR E6] Strategia %d applicabile solo se N e' potenza di 2. Strategia di default: 1\n", N_strat_temp);
        }

    }   //Altrimenti N_strat = N_strat_temp che è uguale a 1, ma 1 è già il valore di default di N_strat
```

- *Viene richiesta la stampa di tutti i processori, ma la strategia richiesta non è la III*

  Si è deciso di rendere possibile la stampa da parte di tutti i processori (indicato mediante l'input *-1* per la variabile *ID_stamp*) solo quando tutti i processori effettuano la somma totale. Ciò si verifica quando viene adottata la strategia III, per questa ragione si segnala errore qualora venisse richiesta la stampa da parte di tutti i processori ma una strategia diversa dalla III.

```
//ERR: se la strategia non è la III e viene richiesta la stampa di tutti
if ((N_strat != 3) && (ID_stamp == -1)) {
    if(menum == 0) {//P0 demandato alla segnalazione degli errori
        printf("[ERROR E7] Strategia %d incompatibile con la stampa di tutti. Strategia imposta: 3\n", N_strat);
    }
    N_strat = 3;
}
```

Sono presenti dei file .PBS simulanti input tali da generare le condizioni di errore appena descritte. Per l'esattezza, il file *somma_E1_E2_E4_E5.pbs* simula gli errori E1, E2, E4, E5, mentre il file *somma_E3_E6_E7.pbs* simula i rimanenti. Sono presenti anche i rispettivi file .out con i risultati ottenuti.

Subroutine

Sono state utilizzate diverse funzioni non appartenenti alla libreria standard del C, alcune delle quali appartenenti alla libreria MPI, altre scritte personalmente. Di seguito verranno trattate individualmente, specificando testata, valori di input e output, e funzionamento.

MPI

Tra le funzioni di MPI coinvolte, cominciamo con la descrizioni delle funzioni di ambiente:

- *int MPI_Init(int *argc, char **argv[])*, si occupa di inizializzare l'ambiente a partire dagli argomenti di input del main: *int argc*, un intero che indica il numero di parole che indichiamo sulla riga di comando e *char *argv[]*, un doppio puntatore a carattere, ovvero un vettore di stringhe che sono esattamente le parole scritte sulla riga di comando. *argv[0]* è il nome del programma, mentre i restanti saranno gli argomenti con i quali (nel nostro esempio) facciamo la somma. Con essi, la routine crea i vari processi che abbiamo

lanciato e crea il common world. Si tratta sostanzialmente della prima necessaria operazione che deve essere eseguita, al fine di inizializzare l'ambiente MPI;

- *int MPI_Finalize()*, si occupa di chiudere i processi, l'ambiente, cestina i communicator liberando dunque gli identificativi e quindi si può terminare il programma. È l'operazione che deve sempre giungere alla fine del codice;

- *int MPI_Comm_rank(MPI_COMM_WOLRD, &menum)*, permette al processo chiamante di conoscere (specificando in ingresso il communicator di appartenenza) il proprio identificativo e lo salva nella variabile *menum* (il cui riferimento viene posto in input). Ovviamente, ogni processo avrà un suo identificativo univoco;

- *int MPI_Comm_size(MPI_COMM_WOLRD, &nproc)*, restituisce il numero di processi totale (relativo al communicator specificato in ingresso) ed immagazzina questa informazione in *nproc*.

Procediamo adesso con la descrizione delle funzioni di comunicazione:

- *int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*, viene utilizzata dal processo mittente per inviare un dato indirizzato con **buf*, avente count elementi, di tipo *datatype*, al processo destinatario identificato con *dest*, associando al messaggio l'identificativo univoco *tag* e tramite il communicator *comm* (che dovrà corrispondere a quello del ricevitore). Quando il processore esegue questa primitiva, rimane in attesa di risposta essendo un'operazione bloccante;

- *int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status * status)*, utilizzata dal processo destinatario che attende un messaggio da immagazzinare in **buf*, avente al massimo count elementi, di tipo *datatype*, da parte del processo con identificativo *source*, messaggio identificato univocamente con *tag*, tramite il communicator *comm* e avente lo stato **status*. Lo *status* contiene per l'appunto lo stato del messaggio, incluse le dimensioni e le informazioni relative alla ricezione. Il destinatario, dunque, può aggiornare lo stato di ricezione del

messaggio operando sulla variabile *status*. Anche in questo caso si tratta di un'operazione bloccante;

- *int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm),* a differenza delle prime due, questa operazione di comunicazione è di natura collettiva. L'istruzione deve essere eseguita da tutti i processi coinvolti, senza una diretta discriminazione tra chi invia e chi riceve (cosa che avviene nel caso della comunicazione uno a uno). *int root* indica l'identificativo del processore che spedirà il messaggio (di tipo *datatype* e nel communicator *comm*). Per il *root* il buffer indicherà la struttura (di *count* elementi) dalla quale prelevare i dati da inviare, mentre per i processori riceventi indicherà la struttura (di count elementi) nella quale immagazzinare il dato spedito dal *root*. Anche in questo caso abbiamo un comportamento bloccante.

In occasione della misura dei tempi, abbiamo utilizzato anche delle operazioni collettive:

- *int MPI_Barrier(MPI_Comm comm)* che permette di sincronizzare tutti i processi appartenenti ad un communicator *comm*, nel senso che quando i processi trovano questa primitiva si fermano e riprendono l'esecuzione tutti insieme solo quando è arrivato anche l'ultimo processo: viene annullata qualsiasi differenza di tempo tra i processi.

- *int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)*, abbiamo un processo designato con *root* che dovrà effettuare una operazione *op* su una serie di dati. Questi dati vengono inviati dai processi partecipanti alla riduzione, dunque spediranno un *sendbuf* di *count* elementi e di tipo *datatype*, dati che giungeranno quindi al processo *root*, appartenente al medesimo communicator *comm*. Il *recvbuf* riguarda esclusivamente il *root* e conterrà il risultato dell'operazione *op* sugli elementi inviati a quest'ultimo.

Tutte queste funzioni ritornano un valore intero, rappresentante un avvenuto successo oppure un codice d'errore.

Per la misura dei tempi è stata utilizzata la funzione *double MPI_Wtime()* che restituisce una misura del tempo in secondi.

Funzioni di supporto

Sono state personalmente scritte cinque funzioni di supporto, riguardanti la distribuzione, le tre strategie e la stampa del risultato. Di seguito:

- *void distribuzione_dati(int me, float *data, float *dataloc, int nl, int np, int res, MPI_Status *stat),* funzione demandata al compito della distribuzione dei dati, sia in uscita che in entrata. Il processore incaricato alla distribuzione dei dati, quale P0, provvede a distribuire per mezzo di una *send* bloccante porzioni di nl elementi contenuti in *data*, agli *np* processori, tenendo conto di un eventuale resto *res*. Il resto, inteso come un numero di addendi maggiore al numero di processori e per cui non è possibile una distribuzione perfettamente equilibrata degli elementi (per esempio 9 elementi per 8 processori), viene gestito dalla routine in questione, cercando di impattare l'inevitabile squilibrio di distribuzione. I processori *me* che non corrispondono al processore 0, ricevono per mezzo di una *recv* bloccante gli elementi che verranno caricati nel vettore *dataloc*. Le informazioni in merito allo stato della spedizione vengono memorizzate nella struttura dati *stat*.

- *void I_strategia(int me, int ID_s, float *sum_p, float *sum_tot, int np, MPI_Status *stat)*, il processore *ID_s* riceve, da parte degli *np* processori e per mezzo di una *recv* bloccante le somme parziali *sum_p* che a sua volta provvederà a sommare nella variabile *sum_tot*. Anche in questo caso, lo stato della spedizione viene indicato nella struttura *stat*.

- *void II_strategia(int me, float *sum_p, float *sum_tot, int idx, int *p_2, int *p_2_plus_1, MPI_Status *stat),* aldilà dei parametri in comune, in questo caso sono previste degli input precedentemente calcolati per cercare di ridurre l'impatto prestazionale durante l'operazione di somma. L'intero *idx* rappresenta il numero di passi di comunicazione che

vengono eseguiti, ed è pari al $log_2(nproc)$. I vettori *p_2* e *p_2_plus_1* contengono, rispettivamente, 2^i e 2^{i+1} calcolati (precedentemente) per $i = 0, \ldots, idx-1$. Anche in questo caso, lo stato della spedizione viene indicato nella struttura *stat*.

- *void III_strategia(int me, float *sum_p, float *sum_tot, int idx, int *p_2, int *p_2_plus_1, MPI_Status *stat),* il ruolo degli input è praticamente lo stesso degli input della seconda strategia, con le dovute differenze che riguardano le due politiche distinte. Si osserva che in questo caso, dal momento che tutti i processori effettuano operazioni di *send* e *recv* ad ogni passo di comunicazione, si è posta l'operazione di *send* precedentemente a quella di *recv*, per evitare condizioni di deadlock.

- *void stampa_ris(int me, float sum_tot, int ID_s, int N_str, MPI_Status *stat),* funzione di stampa del risultato, prende in ingresso una copia *sum_tot* della somma totale e l'*ID_s* relativo al processore che deve stampare. Viene considerato anche il numero *N_str* di strategia. Se *ID_s* dovesse essere pari a *-1*, al cospetto della terza strategia tutti i processori stamperanno direttamente, altrimenti verrà segnalato da P0 l'impossibilità di stampare (ricordo, tuttavia, che è prevista una relativa gestione dell'errore, al cospetto di questa circostanza). Qualora *ID_s* dovesse essere strettamente maggiore di *-1*, al cospetto della II strategia, essendo P0 l'unico processore ad avere *sum_tot* provvederà a stamparlo se $ID\_s = 0$, altrimenti lo spedirà ad $ID\_s \neq 0$ che lo stamperà. Se invece la strategia è la I o la III, *ID_s* può procedere direttamente alla stampa.

Analisi dei tempi

La misura dei tempi è stata effettuata per mezzo della funzione *double MPI_Wtime(),* assicurandoci che tutti i processori fossero sincronizzati al momento della misura. Gli istanti di tempo misurati sono stati due, rispettivamente memorizzati nelle variabili *t0* e *t1*:

- *t0* è stato memorizzato esattamente prima della prima istruzione di somma locale, che dà inizio all'algoritmo di somma;
- *t1* è stato memorizzato esattamente un'istruzione dopo l'ultima operazione di somma totale.

Nella misura non ricade né il prelievo e controllo dei dati (con relativa distribuzione), né la stampa dei risultati. Le operazioni di calcolo delle potenze di 2 e del logaritmo, necessarie per implementare le strategie II e III, sono state effettuate a monte della misurazione, memorizzando i risultati in specifiche strutture dati di supporto: questa scelta è stata fatta per evitare di alterare le prestazioni relative alla somma. Successivamente, ogni processore stampa la differenza tra *t1* e *t0*: sarà il processore *ID_stamp* a stampare il massimo tra i valori registrati dai singoli processori. Nei valori misurati, sono state considerate cinque dimensioni del problema, tutte potenze di 2 per assicurare l'applicabilità delle strategie II e III. Ogni caso di misura è strutturato nel seguente modo:

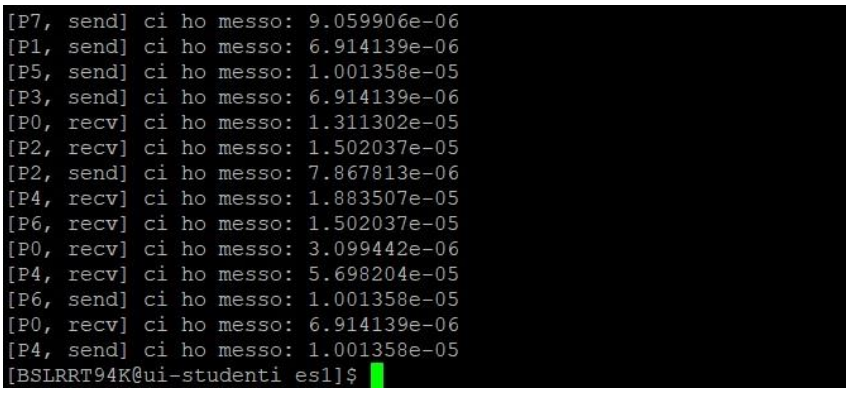
nproc processori, dimensione n del problema, strategia N_strat, timemax misurato

Ogni caso è stato misurato tre volte e la misura riportata è frutto di una media aritmetica delle tre misure. Il numero di processori coinvolti è stato: 1, 2, 4, 8; nel caso di *nproc = 1* si è scelto di eseguire lo stesso algoritmo, non prevedendo una soluzione a parte puramente sequenziale. Il numero di casi totali previsti è stato di 60, per un complessivo di 180 misurazioni (considerando le ripetizioni della misura).

Dimensione N = 1024

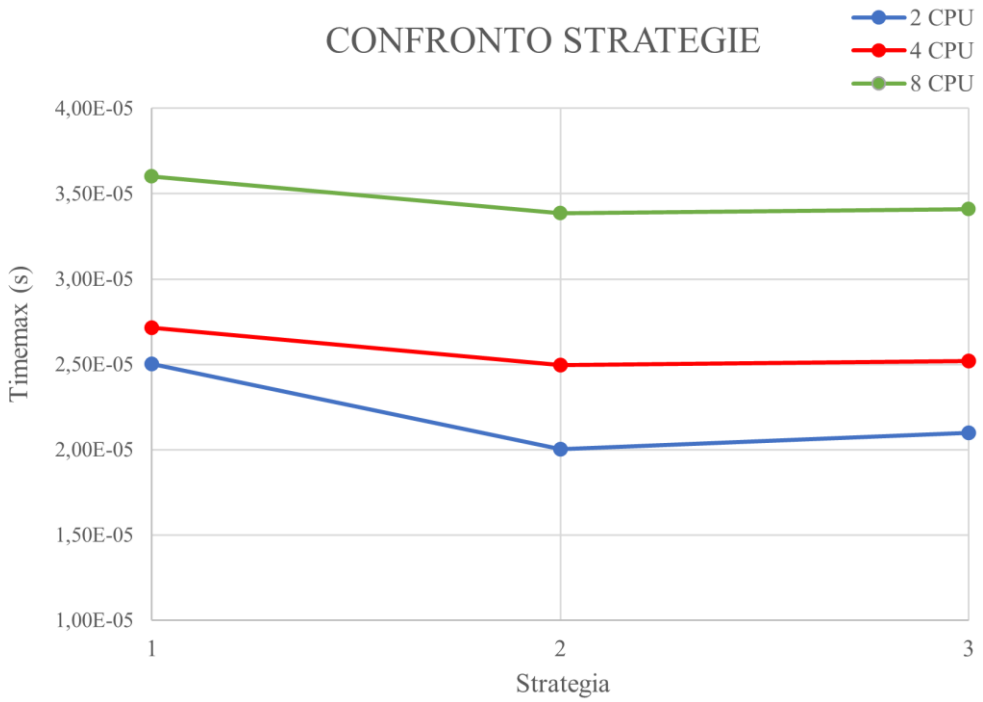
#CPU	Strategia	Timemax (s)
1	1	5,50204E-06
1	2	7,15256E-06
1	3	6,91414E-06
2	1	2,50340E-05
2	2	2,00272E-05
2	3	2,09808E-05
4	1	2,71345E-05
4	2	2,49413E-05
4	3	2,51871E-05
8	1	3,60012E-05
8	2	3,38554E-05
8	3	4,41074E-05

Nel caso di un solo processore, si registrano tempi più bassi rispetto alle soluzioni multiprocessore. Questa anomalia è probabilmente legata ai tempi funzionali della comunicazione tra processori (trattandosi di architettura MIMD a memoria distribuita), comunicazione che non avviene nel caso in cui il processore è unico. La presenza di overhead di comunicazione è stata registrata da questa misurazione concernente esclusivamente i tempi di send e recv, non considerando alcuna operazione di calcolo o di controllo e relativa al caso della strategia II:



I processori P0 e P2, per esempio, durante la ricezione impiegano un tempo nell'ordine di 10^{-5}, comportando un timemax complessivo del medesimo ordine. Per una dimensione del problema così ridotta la scelta di avere un numero maggiore di processori non è giustificata.

Osserviamo adesso i grafici dei tempi per i casi di 2, 4 e 8 CPU mettendo a confronto le tre diverse strategie:

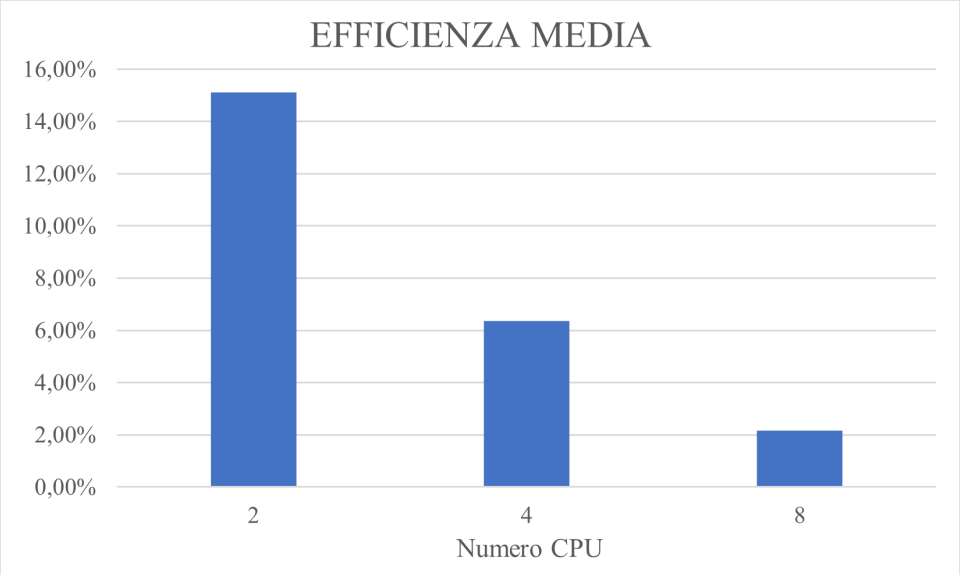


Notiamo come il passaggio dalla prima strategia alla seconda o alla terza comporta un lieve miglioramento dei tempi. Come precedentemente osservato, compatibilmente all'overhead di comunicazione, i tempi nel caso di 4 e 8 CPU risultano peggiori rispetto al caso di 2 CPU.

Di seguito la misura dello speed-up effettivo e dell'efficienza:

SPEED UP			
Strategia	S(2)	S(4)	S(8)
1	0,219783	0,202769	0,152829
2	0,357143	0,286776	0,211268
3	0,329546	0,274511	0,156757
EFFICIENZA			
1	10,99%	5,07%	1,91%
2	17,86%	7,17%	2,64%
3	16,48%	6,86%	1,96%
MEDIA	**15,11%**	**6,37%**	**2,17%**

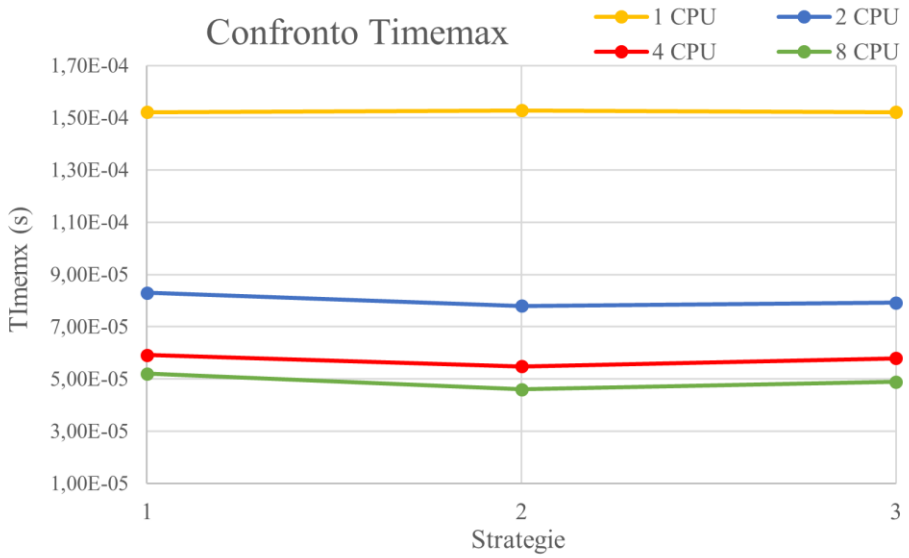
Ci rendiamo immediatamente conto che, indipendentemente dalla strategia, i valori di efficienza sono estremamente bassi, e peggiorano all'aumentare del numero di processori.


EFFICIENZA MEDIA
16,00%
14,00%
12,00%
10,00%
8,00%
6,00%
4,00%
2,00%
0,00%
2
4
8
Numero CPU

Dimensione N = 32768

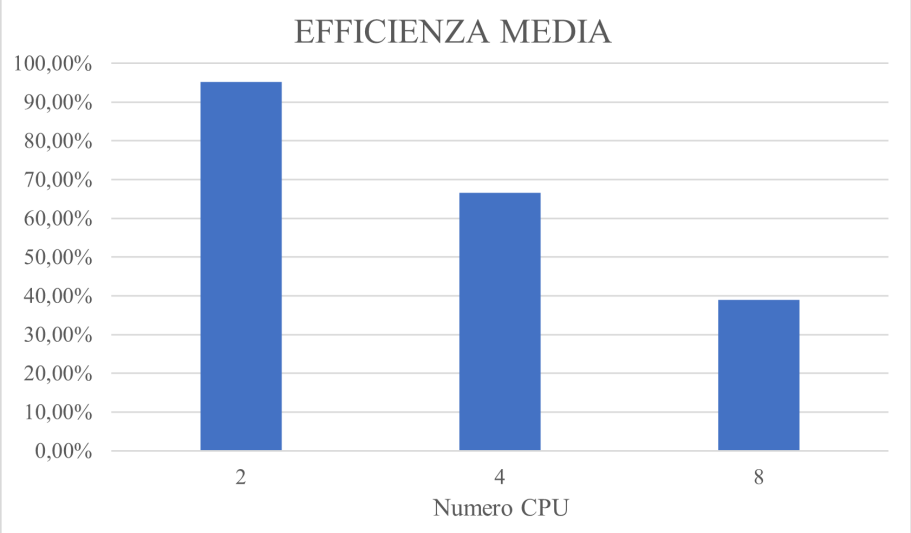
#CPU	Strategia	Timemax (s)
1	1	1,52111E-04
1	2	1,52826E-04
1	3	1,52111E-04
2	1	8,29697E-05
2	2	7,79629E-05
2	3	7,91550E-05
4	1	5,91278E-05
4	2	5,48363E-05
4	3	5,79357E-05
8	1	5,19753E-05
8	2	4,60148E-05
8	3	4,88894E-05

Ci rendiamo immediatamente conto che, aumentando la dimensione del problema, le migliorie rispetto al caso di una sola CPU sono apprezzabili di almeno un ordine di grandezza. Graficamente, mettendo a confronto i *Timemax* delle diverse soluzioni, abbiamo:



In termini di speed-up ed efficienza notiamo un lieve miglioramento rispetto al caso precedente. Nel caso di 2 CPU, l'efficienza media supera il 95%, per poi scendere al 66% nel caso di 4 CPU e ulteriormente (39%) nel caso di 8 CPU. Osserviamo come l'aver aumentato la dimensione del problema abbia portato ad una crescita notevole delle prestazioni (seppur ancora basse nel caso di 8 CPU).

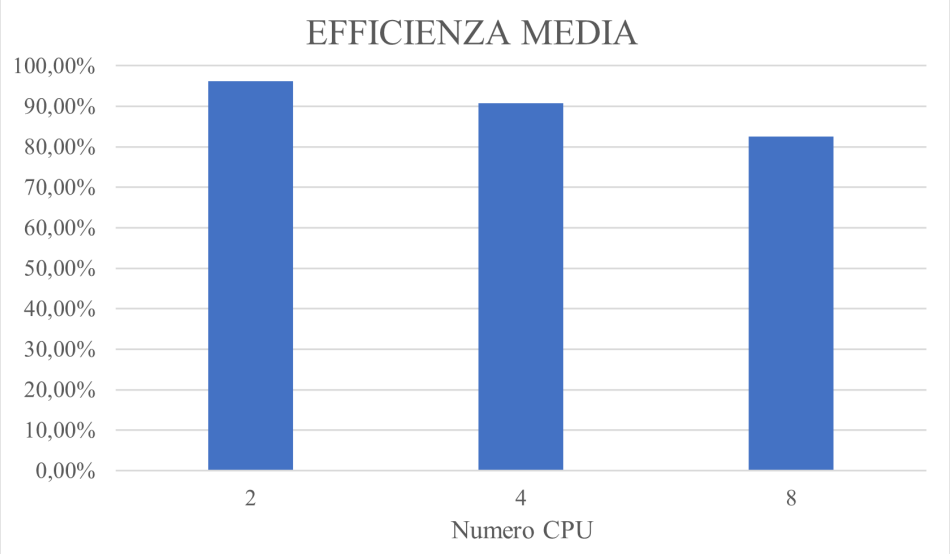
SPEED UP			
Strategia	S(2)	S(4)	S(8)
1	1,83333	2,57258	2,92661
2	1,96024	2,78696	3,32124
3	1,92168	2,62552	3,11133
EFFICIENZA			
1	91,67%	64,31%	36,58%
2	98,01%	69,67%	41,52%
3	96,08%	65,64%	38,89%
MEDIA	**95,25%**	**66,54%**	**39,00%**



Dimensione N = 524288

#CPU	Strategia	Timemax (s)
1	1	2,06494E-03
1	2	2,05708E-03
1	3	2,06399E-03
2	1	1,07193E-03
2	2	1,07002E-03
2	3	1,07193E-03
4	1	5,86033E-04
4	2	5,60999E-04
4	3	5,58853E-04
8	1	3,24965E-04
8	2	3,07798E-04
8	3	3,04937E-04

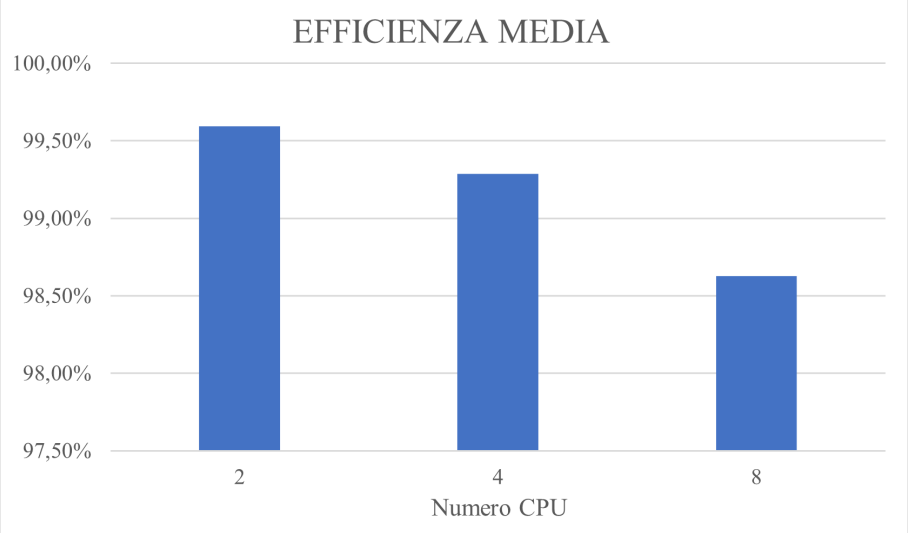
SPEED UP			
Strategia	S(2)	S(4)	S(8)
1	1,92638	3,52360	6,35436
2	1,92246	3,66681	6,68319
3	1,92549	3,69326	6,76857
EFFICIENZA			
1	96,32%	88,09%	79,43%
2	96,12%	91,67%	83,54%
3	96,27%	92,33%	84,61%
MEDIA	**96,24%**	**90,70%**	**82,53%**



Dimensione N = 16777216

#CPU	Strategia	Timemax (s)
1	1	6,76730E-02
1	2	6,75449E-02
1	3	6,75421E-02
2	1	3,39731E-02
2	2	3,39260E-02
2	3	3,38941E-02
4	1	1,70381E-02
4	2	1,70159E-02
4	3	1,70009E-02
8	1	8,52990E-03
8	2	8,56805E-03
8	3	8,50200E-03

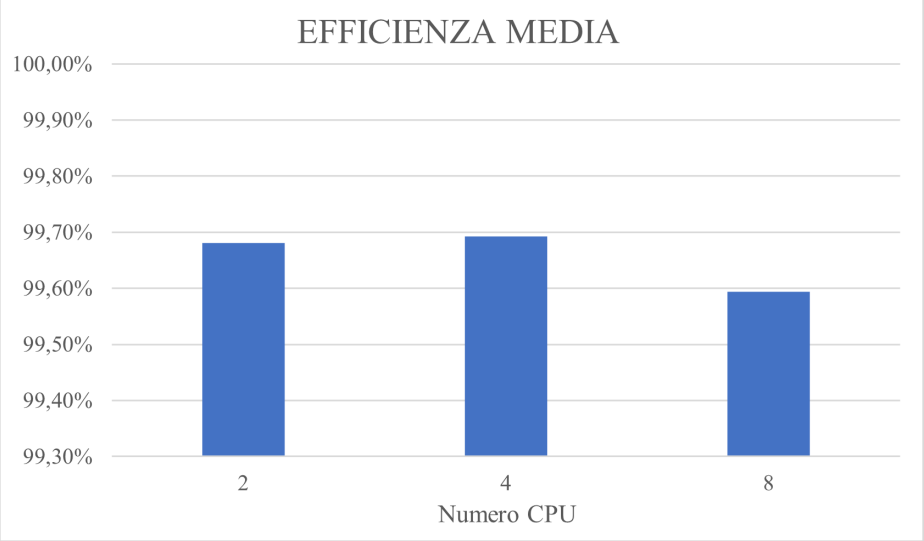
SPEED UP			
Strategia	S(2)	S(4)	S(8)
1	1,99196	3,97186	7,93362
2	1,99095	3,97285	7,88335
3	1,99274	3,69326	7,85373
EFFICIENZA			
1	99,60%	99,30%	99,17%
2	99,55%	99,24%	98,17%
3	99,64%	99,32%	84,61%
MEDIA	**99,59%**	**99,29%**	**98,63%**



Dimensione N = 536870912

#CPU	Strategia	Timemax (s)
1	1	2,16063E+00
1	2	2,15507E+00
1	3	2,16051E+00
2	1	1,08373E+00
2	2	1,08134E+00
2	3	1,08341E+00
4	1	5,41459E-01
4	2	5,40923E-01
4	3	5,40660E-01
8	1	2,70980E-01
8	2	2,70142E-01
8	3	2,70807E-01

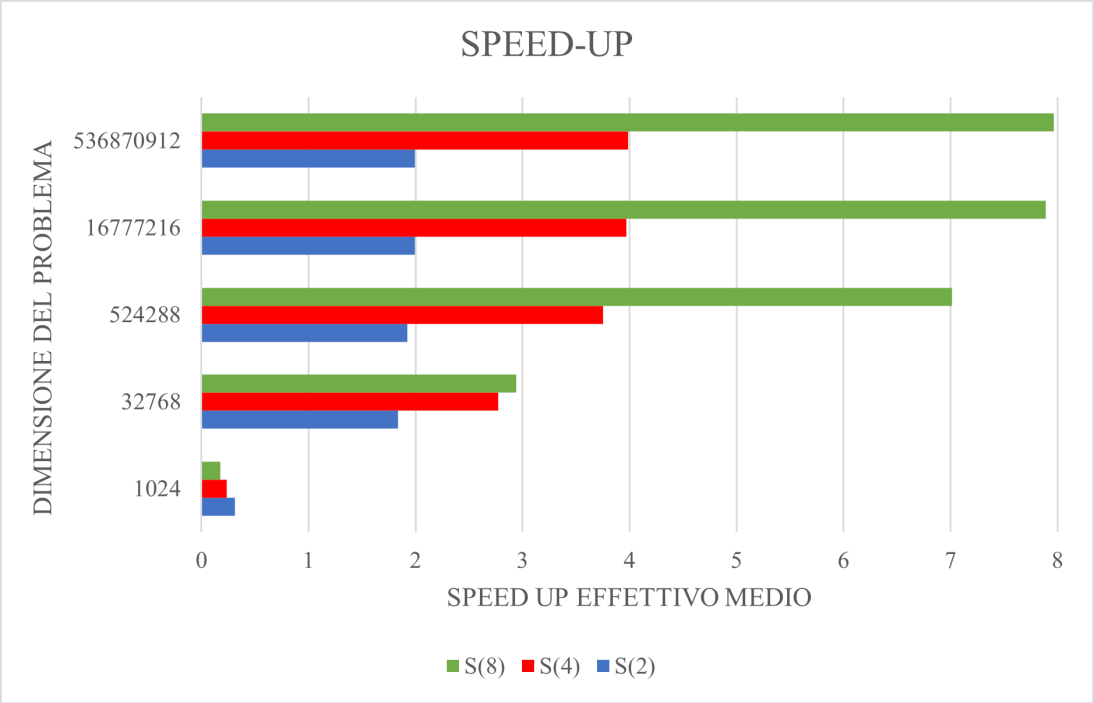
SPEED UP			
Strategia	S(2)	S(4)	S(8)
1	1,99370	3,99434	7,97339
2	1,99296	3,98011	7,95104
3	1,99417	3,98868	7,97804
EFFICIENZA			
1	99,68%	99,86%	99,67%
2	99,65%	99,50%	99,39%
3	99,71%	99,72%	99,73%
MEDIA	**99,68%**	**99,69%**	**99,59%**

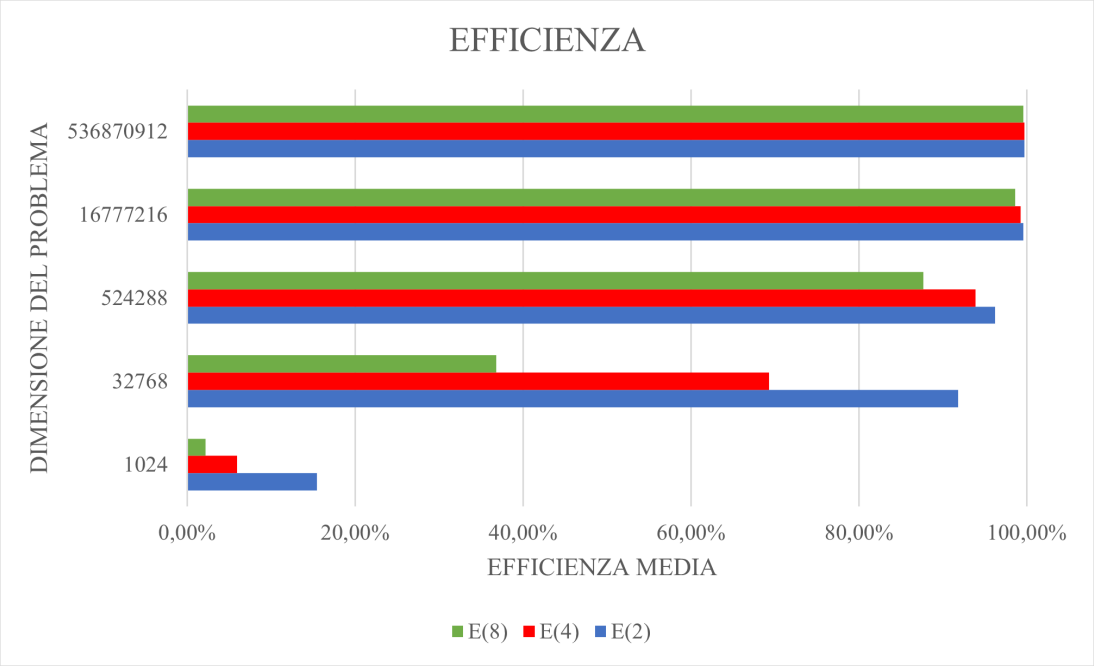


Conclusioni

Partendo da una dimensione particolarmente ridotta, ovvero 2^{10}, ci siamo resi conto che l'overhead generato dalla presenza di più processori non è giustificato, ed anzi le prestazioni risultano estremamente degradate, al punto di raggiungere nel caso di 8 processori un'efficienza media del 2.17% ed uno speed-up effettivo medio di circa 0.173, ben distante da quello ideale pari ad 8. Di contro, aumentando la dimensione del problema arrivando fino a 2^{29} osserviamo un incremento prestazionale notevole in termini di efficienza e speed-up medi: 99.59% e 7.967 nel caso di 8 processori. A partire dalla dimensione 2^{19}, e così come per tutte le dimensioni superiori del problema, i valori di speed-up ed efficienza ottenuti, per tutte le soluzioni multiprocessore considerate, sono stati rappresentativi di un utilizzo efficiente dell'ambiente di calcolo parallelo. Ciò non sorprende alla luce della legge di Ware.

Vediamo il confronto dei valori medi al variare della dimensione del problema:





Non ci sorprende osservare che, nonostante una dimensione del problema molto elevata, l'efficienza ideale e lo speed-up ideale non vengano mai raggiunti: che sia per le comunicazioni o comunque per la gestione della strategia, un certo overhead è sempre presente.

Isoefficienza

A partire dalle misure registrate, si è voluto effettuare un esperimento sul concetto di isoefficienza, considerando il caso in cui $n_0 = 524.288$, $p_0 = 2$ e valore di efficienza pari a *96.32%*. Il numero di processori p_1 è pari a 8: vogliamo conoscere a quanto ammonta la dimensione n_1 del problema al fine di avere il medesimo valore di efficienza. Nel caso dell'algoritmo della somma (che abbiamo visto essere un algoritmo scalabile, ovvero che l'efficienza non degrada all'aumentare del numero dei processori e della dimensione del problema), il calcolo dell'isoefficienza è dato da

$$n_1 = n_0 \frac{p_1 log_2(p_1)}{p_0 log_2(p_0)} = I(n_0, p_0, p_1)$$

In questo caso, $n_1 = 6291456$. Dalle misure effettuate, seguendo gli stessi criteri adottati fino a questo momento, i risultati registrati sono stati i seguenti:

#CPU	Timemax (s)
1	2,520313E-02
8	3,259155E-03

L'efficienza calcolata, per 8 CPU, è pari a $E_{n1}(8) = 96.66\% \simeq E_{n0}(2)$.

Esempi d’uso

Sono stati effettuati quattro esempi d’uso, variando la dimensione del problema, la strategia adottata, il numero di processori e il metodo di stampa. Esempi relativi alla gestione degli errori e alle relative stampe sono presenti nella cartella “Errori”, contenente i file *.PBS* e *.out*. Di seguito verranno riportati sia i dati di input che i risultati ottenuti da terminale.

1. Input: *Somma di 2^{18} elementi, 8 CPU, Strategia III e stampa di tutti.*

   Output:

```
.../mpiexec -machinefile E -np 8
/homes/DMA/PDC/2020/BSLRRT94K/es1/somma 262144 3 -1
[P7] Somma: 13234635.000000, strategia: 3.
[P0] Somma: 13234635.000000, strategia: 3.
[P1] Somma: 13234635.000000, strategia: 3.
[P2] Somma: 13234635.000000, strategia: 3.
[P3] Somma: 13234635.000000, strategia: 3.
[P5] Somma: 13234635.000000, strategia: 3.
[P6] Somma: 13234635.000000, strategia: 3.
[P4] Somma: 13234635.000000, strategia: 3.
```

2. Input: *Somma di 16 numeri (10, 10, …, 10), 4 CPU, Strategia II, stampa ID: 3*.

   Output:

```
.../mpiexec -machinefile E -np 4
/homes/DMA/PDC/2020/BSLRRT94K/es1/somma 16 2 3 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10
[P3] Somma: 160.000000, strategia: 2.
```

3. Input: *Somma di 2^{20} elementi, 2 CPU, Strategia III e stampa di tutti.*

   Output:

```
.../mpiexec -machinefile E -np 2
/homes/DMA/PDC/2020/BSLRRT94K/es1/somma 1048576 3 -1
[P0] Somma: 52953632.000000, strategia: 3.
[P1] Somma: 52953632.000000, strategia: 3.
```

4. Input: *Somma di 2^{10} elementi, 1 CPU, Strategia I e stampa di 0.*

   Output:

```
.../mpiexec -machinefile E -np 1
/homes/DMA/PDC/2020/BSLRRT94K/es1/somma 1024 1 0
[P0] Somma: 52557.000000, strategia: 1.
```