UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
SCUOLA POLITECNICA E DELLE SCIENZE DI BASE
DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

CORSO DI LAUREA IN INFORMATICA

Anno Accademico 2021/2022

Algoritmo per la somma di N numeri in parallelo

Autore

Carmine Grimaldi

Matr. N97000394

Sommario

1. Definizione ed analisi del problema

Il software presentato in questa relazione, si occupa di sommare N numeri attraverso un'architettura **MIMD**, ossia un'architettura parallela in cui unità di elaborazione distinte eseguono simultaneamente elaborazioni su flussi di dati diversi.
Riguardo le risorse hardware, il codice viene eseguito sul cluster SCoPE.
Il risultato a cui ambiamo è che all'aumentare del numero dei processori, aumenta l'efficienza con cui il risultato viene restituito, ma ciò in realtà non è possibile, principalmente per due motivi. Il primo motivo, è che per eseguire dei programmi paralleli, per quanto si possa strutturare l'algoritmo per massimizzare il parallelismo, ad un certo punto i processi si troveranno in una situazione di dipendenza l'uno dall'altro, cioè per poter proseguire c'è bisogno che i processi comunichino i propri risultati agli altri processi, di conseguenza un processo si mette in attesa fin quando il mittente non inoltra il proprio risultato calcolato, ciò introdurrà inevitabilmente un overhead. Un altro motivo è che nel corso dell'esecuzione di un algoritmo parallelo, i processori coinvolti non saranno mai tutti attivi contemporaneamente, ciò è anche legato ai problemi di comunicazione.

2. Descrizione dell'algoritmo

La somma di N numeri in parallelo può essere calcolata con diverse strategie, lo scenario ideale sarebbe ottenere il massimo parallelismo, effettuando il maggior numero di somme in parallelo, ma ciò non è possibile, poiché i processi, quando calcolano un proprio risultato parziale, per proseguire con la composizione del risultato finale hanno bisogno di comunicare i loro risultati ad altri processi.

In particolare, sono state implementate tre strategie differenti:

- Nella **prima** strategia si stabilisce un processo principale, che viene scelto dall'utente in input, a cui verranno comunicati tutti i risultati parziali, dopodiché tale processo si occuperà di comporre il risultato finale.

- Nella **seconda** strategia i processori lavorano a coppie. Inizialmente tutti i processi sono attivi e in questa fase il processo P_i invia il proprio risultato parziale al processo P_{i-1}. Proseguendo con la costruzione del risultato ad ogni iterazione, il numero di processi attivi si dimezza. La comunicazione dovrebbe essere vista come un "albero capovolto" dove le foglie dell'albero sono i processi, i nodi interni rappresenteranno il destinatario della coppia da cui parte il calcolo della somma parziale, per poi

andare a formare una nuova coppia con un processo che era anche lui destinatario al passo precedente, questo processo si ripete fino a quando non resta un solo processo attivo che possiederà la somma totale.

- La **terza** strategia è sostanzialmente un'estensione della seconda, ma si aggiungono scambi di informazioni simmetrici; ciò si traduce nel fatto che, nello stesso passo, una coppia di processori si scambia i propri risultati parziali in entrambe le direzioni, mentre nella seconda strategia la comunicazione avviene in un solo verso.

Differenze principali:

- La seconda strategia, rispetto alla prima, ci aspettiamo che impieghi meno tempo; infatti, qui il grado di parallelismo aumenta, visto che le coppie di processori iniziano a costruire parti del risultato finale in modo "indipendente", mentre nella prima strategia, c'è un solo processo che resta in attesa che tutti gli altri producano il proprio risultato e glielo comunichino. Ottenuto un risultato parziale, ogni processo resta inattivo.
  Uno svantaggio della seconda strategia rispetto alla prima, è che per il suo modo di effettuare le comunicazioni ad albero binario, richiede necessariamente un numero di processi che sia una potenza di due, di conseguenza riduce la sua applicabilità ad una classe ben definita di casi (ad esempio, nel caso in cui si hanno a disposizione 25 processori, con tale strategia se ne potranno utilizzare solo 16, lasciandone 9 inattivi, mentre la prima strategia non presenta questa limitazione).

- La terza strategia, a differenza della seconda, è utile nel caso in cui al termine del calcolo della somma tale valore viene preso in input per effettuare altre tipologie di operazioni, ed in particolare se questo valore e necessario a tutti i processi; mentre nella seconda strategia, un solo processo al termine possiede il risultato finale. Uno svantaggio rispetto alla seconda strategia, è che per effettuare questa comunicazione in entrambi i versi si aggiunge un overhead.

- Tra la prima e la terza strategia valgono le stesse differenze che si presentano tra la prima e la seconda strategia.

Pseudo-codice strategia 1

```
int strategia1 (int cpuPrint, int numCpu, int addendi:in_rif, int numAddendi)
begin
        forall Pi , 0 ≤ i ≤ numCpu - 1 do
                sommaParziale := 0
                for i:=0 to numAddendi-1 do
                        sommaParziale := sommaParziale + addendi[i]
                endfor

                if(P_i ≠ P_cpuPrint) then
                        "P_i invia sommaParziale a P_cpuPrint"
                else
                        sommaTotale := sommaParziale
                        for j:=1 to numCpu-1 do
                                idSrc := (numCpu + j + i) % numCpu
                                "P_cpuPrint riceve la sommaParziale da P_idSrc"
                                sommaTotale := sommaTotale + sommaParziale
                        endfor
                        return sommaTotale
                endif
        end forall
end
```

Nella prima strategia, ogni processo calcola la propria somma parziale, al termine di ciò, il processo principale attende che tutti gli altri gli inoltrino la loro somma parziale. Gli inoltri vengono fatti partire dal processo $(P_{cpuPrint+1})$ successivo al processo principale, ma per conoscere tale processo bisogna calcolarlo tramite il modulo del numero dei processi, poiché se il processo principale è l'ultimo, il prossimo processo sarà P_0. Al termine, il processo principale restituisce in output la somma totale.

Pseudo-codice strategia 2

```
int strategia2 (int cpuPrint, int numCpu, int addendi:in_rif, int numAddendi)
begin
        forall Pi , 0 ≤ i ≤ numCpu - 1 do
                if(P_i = P_cpuPrint)then
                        log2n := log_2 numCpu
                        "P_i alloca un array e lo popola con le prime log2n potenze di 2"
                endif
                "P_cpuPrint comunica il valore log2n a tutti gli altri processi"
                if(P_i ≠ P_cpuPrint)then
                        "P_i alloca un array per ricevere le potenze di 2 da P_cpuPrint"
                endif
                "P_cpuPrint comunica le potenze di due precedentemente calcolate a tutti gli altri processi"
                for j:=0 to log2n do
                        shiftId := ((j - cpuPrint + numCpu) % numCpu);
                        if(shiftId % 2^j = 0) then
                                if(shiftId % 2^(j+1) ≠ 0) then
                                        dst := ((numCpu + (i - 2^j)) % numCpu)
                                        "P_i invia sommaParziale a P_dst"
                                else
                                        sommaTmp := sommaParziale
                                        src := ((numCpu + (i + 2^j)) % numCpu)
                                        "P_i riceve sommaParziale a P_src"
                                        sommaParziale := sommaParziale + sommaTmp
                                endif
                        endif
                endfor
                "Si deallocano gli array delle potenze di 2"
                if(P_i = P_cpuPrint) then
                        return sommaParziale
                endif
        end forall
end
```

In questa seconda strategia, inizialmente, come per la prima, tutti I processi calcolano la propria somma parziale, dopodiché, tramite il controllo
if(shiftId % 2^j = 0) si filtrano i processi che alla j-esima iterazione partecipano ancora alle comunicazioni.

Sulla seconda condizione if(shiftId % 2^{j+1} ≠ 0) si controlla quali sono i processi che alla prossima iterazione non parteciperanno a comunicazioni, ciò ci fa capire che essi nell'iterazione attuale devono inviare i loro risultati parziali; nel caso opposto, invece, essi riceveranno i risultati parziali dal processo $P_i + 2^j$.
Gli id dei processi vengolo calcolati sempre in modulo, per far in modo che il processo principale sia quello scelto dall'utente. Inoltre, in tale strategia, quando ci riferiamo alle potenze di due, ci stiamo riferendo ad una cella di un array opportunamente inizializzato. Il for più esterno che itera sulle comunicazioni e si occupa del calcolo della somma totale varia da 0 a $\log_2 numCpu$, poichè ad ogni iterazione il numero di processi attivi viene dimezzato, fino a che non resta solo il processo principale con la somma totale.

Pseudo-codice strategia 3

```
int strategia3 (int cpuPrint, int numCpu, int addendi:in_rif, int numAddendi)
begin
        forall Pi , 0 ≤ i ≤ numCpu - 1 do
                if(P_i = P_cpuPrint)then
                        log2n := log_2 numCpu
                        "P_i alloca un array e lo popola con le prime log2n potenze di 2"
                endif
                "P_cpuPrint comunica il valore log2n a tutti gli altri processi"
                if(P_i ≠ P_cpuPrint)then
                        "P_i alloca un array per ricevere le potenze di 2 da P_cpuPrint"
                endif
                "P_cpuPrint comunica le potenze di due precedentemente calcolate a tutti gli altri processi"
                for j:=0 to log2n do
                        shiftId := ((j - cpuPrint + numCpu) % numCpu);
                        if(shiftId % 2^(j+1) < 2^j) then
                                dstRcv := ((numCpu + (i + 2^j)) % numCpu)
                                "P_i invia sommaParziale a P_dstRcv"
                                "P_i riceve sommaTemp da P_dstRcv"
                                sommaParziale := sommaParziale + sommaTemp
                        else
                                dstRcv := ((numCpu + (i - 2^j)) % numCpu)
                                "P_i riceve sommaTemp da P_dstRcv"
                                "P_i invia sommaParziale a P_dstRcv"
                                sommaParziale := sommaParziale + sommaTemp
                        endif
                endfor
                "Si deallocano gli array delle potenze di 2"
                if(P_i = P_cpuPrint)then
                        return sommaParziale
                endif
        end forall
end
```

Il principio di funzionamento è lo stesso della seconda strategia, ma in questo caso i processi che alla prossima iterazione non dovrebbero partecipare, in realtà entreranno a far parte di un secondo gruppo, dopodiché i processi di due gruppi distinti si scambieranno le proprie somme parziali andando a costruire la somma totale, la quale sarà posseduta da tutti i processi, a differenza della prima e della seconda strategia.

3. Guida alla compilazione e manuale d'uso

Di seguito è illustrato il modo con cui è possibile invocare l'eseguibile allegato a questa relazione.

```
qsub Somma.pbs -v ID=[cpuID],N=[size],STR=[strategy]
```

All'interno dello script "Somma.pbs" è presente l'invocazione vera e propria dell'algoritmo (scritto in linguaggio C), ciò avviene attraverso l'ausilio della libreria MPI (Message Passing Interface):

```
mpiexec Somma ${ID} ${STR} ${N} [x1 x2 x3 …]
```

Passaggio dei parametri

cpuID: indica l'id del processore principale che si occuperà di allocare memoria, distribuire dati agli altri processori e fornire in output i risultati; tale valore deve essere compreso tra 0 e il numero di processori - 1, o nel caso in cui si vuole che tutti i processi diano in output il risultato, si può dare in input il valore "-1".

size: la quantità di valori che si desidera sommare; opzionalmente si può fornire una lista di tali valori $[x_1\ x_2\ \dots\ x_N]$ se la quantità di numeri da sommare è minore o uguale a 20.
In caso contrario, verrà utilizzata una lista di numeri generati in modo casuale.

strategy: identificativo della strategia che si vuole utilizzare, tale valore deve essere compreso tra "1" e "3".

In output sarà fornito un messaggio che indicherà la somma totale e quanto tempo è stato impiegato per ottenerla.

Attenzione: al fine di mostrare delle tempistiche simili a quelle che sono illustrate in questa relazione, il programma allegato a questa documentazione è eseguito attraverso la direttiva "#PBS -l nodes=8:ppn=8" all'interno del file "Somma.pbs", ciò si traduce nell'utilizzo totale delle risorse del cluster dell'infrastruttura "SCoPE" per l'esecuzione dell'algoritmo di somma.

In allegato a questa documentazione è presente la cartella "Output", in cui è contenuto un file di output di esempio per ogni strategia, utilizzando 8 processori e N=10.000.000.
Per semplicità di verifica del risultato, i numeri da sommare sono tutti pari a "1".

Indicatori di errore

Di seguito sono illustrati gli errori segnalati a causa di un input errato da parte dell'utente.

Si ricorda che l'ordine con il quale vengono passati i parametri è il seguente:

mpiexec Somma **${ID} ${STR} ${N}**

mpiexec Somma 0 1 **-5**: viene segnalato che non è possibile sommare una quantità negativa o nulla di valori.

mpiexec –np **16** Somma 0 3 **10**: viene segnalato che non è possibile sommare una quantità di valori inferiore alla quantità di processi che si sta tentando di lanciare.

mpiexec –np **3** Somma 0 **2** 100 / mpiexec –np **3** Somma 0 **3** 100: nel caso in cui si desidera utilizzare la seconda o la terza strategia con un numero di processi che non è una potenza di due, si segnala che non è possibile farlo e si procede con la prima strategia.

mpiexec Somma 0 **0** 100 / mpiexec Somma 0 **4** 100: nel caso in cui l'identificativo della strategia da utilizzare sia minore di 1 o maggiore di 3, viene segnalato che tale strategia non esiste e si termina l'esecuzione.

Funzioni

void start(int argc, char *argv[], int idCpu, int numCpu):

funzione invocata dal main dopo aver effettuato dei controlli sulla correttezza dell'input; prende in input gli argomenti ricevuti dal main, il numero di processori e l'id del processo che lo sta invocando. Mediante l'invocazione di altre funzioni interpreta l'input in modo da capire quale sia effettivamente la strategia da utilizzare, dopodiché invoca tale strategia per produrre il risultato di somma.

int strategyChoice(int chiamante, int cpuPrint, int strat, int nCpu):

prende in input l'id del processo che la invoca, l'id del processo principale, rappresentato da "cpuPrint", il numero di strategia che si vuole utilizzare ($1 \leq strat \leq 3$) ed infine il numero di processi totale. In base al numero di processi la funzione verifica che la scelta dell'utente sia possibile, in caso contrario si applica la prima strategia. Restituisce in output la strategia da applicare, oppure restituisce 0 in caso di incorrettezza dell'input.

int * addendsDistribution(int idCpu, char * argv[], int * dim2, int * add):

prende in input l'id del processo che lo invoca, l'array di argomenti che è stato fornito in input al main, un puntatore ad intero su cui si memorizzerà un'informazione da dare in uscita e il puntatore all'array che contiene i valori da sommare (la dimensione di tale array è già inclusa in argv[3]).
Tale funzione calcola innanzitutto quanti elementi da sommare saranno assegnati ad ogni processo, dopodiché il risultato viene memorizzato nella locazione di memoria puntata da dim2.

A questo punto il processo principale invia gli addendi ai processi, i quali memorizzeranno una loro copia di questi ultimi in un loro array personale. Al termine viene restituito in output l'array con i valori che dovranno essere sommati, in particolare a ogni processo verrà restituito il puntatore al proprio array.

int * sumArrayInit(int taglia, int argc, char *argv[]):

prende in input la dimensione dell'array che dovrà allocare e i parametri presi in input dal main. Alloca un array d'interi della dimensione indicata in "taglia", se "taglia" <= 20 allora l'array sarà popolato con valori dati in input dall'utente, altrimenti l'array verrà popolato con valori generati in maniera casuale.

int localSum(int * addendi, int numAddendi):

prende in input un array di interi e la sua dimensione, effettua la somma di tutti i valori contenuti in esso e la restituisce in output.

int equivalentID(int idCpu, int idCpuPrint, int numCpu):

prende in input l'identificativo del processo che la invoca, l'identificativo del processo principale e il numero di processi totale. Attraverso l'operazione di modulo, tale funzione comunica in output l'identificativo del processo idCpu che tale processo avrebbe avuto nel caso in cui quello principale fosse stato zero.

int equivalentSrcDst(int idCpu, int numCpu):

prende in input l'id del processo che la invoca, l'id del processo principale e il numero di processi totale, tramite l'operazione di modulo calcola l'id della sorgente o del destinatario di un messaggio che effettivamente dovrà effettuare tale operazione. Tale calcolo è necessario in quanto non sempre il processo principale è quello zero.

void printResults(int idCpu, int cpuPrint, int somma, double tempo):

prende in input l'id del processo che la invoca, l'id del processo principale, la somma totale e il tempo che è stato impiegato per calcolarla, per poi procedere alla loro stampa.
Se $\mathrm{cpuPrint} \leq -1$ stampano tutti i processori, in caso contrario un solo processore, identificato dall'id "$\mathrm{cpuPrint}$", stampa in output il risultato.

int strategy1(int chiamante, int cpuPrint, int numCpu, int * addendi, int numAddendi, double * t),
int strategy2(int chiamante, int cpuPrint, int numCpu, int * addendi, int numAddendi, double * t),
int strategy3(int chiamante, int cpuPrint, int numCpu, int * addendi, int numAddendi, double * t):
prendono in input l'id del processo chiamante, l'id del processo principale, il numero dei processi totale e il puntatore all'array degli addendi che dovranno essere sommati dal processo chiamante, il numero di addendi da sommare, e un puntatore a una locazione di memoria in cui verrà memorizzato il tempo impiegato ad effettuare tale somma.

int generateRandom(int min, int max):

funzione che genera un valore intero compreso tra "min" e "max".

double logarithm(double base, double argomento):

calcola il $\log_{base}(argomento)$.

int * calculatePowers(int base, int numPotenze):

prende in input la base delle potenze da calcolare e il numero di potenze da calcolare, alloca un array con una dimensione sufficiente a contenere il numero di potenze richieste dall'utente dopodiché popola tale array nel seguente questo modo:

$$\forall 0 \leq i \leq numPotenze : potenze[i] = base^i$$

infine viene restituito in output il puntatore a tale array.

int testPowerOfTwo(double x):

verifica che x sia una potenza di 2, tale verifica viene eseguita mediante la seguente espressione condizionale: $\log_2 x - \lfloor \log_2 x \rfloor = 0$, poiché se x è una potenza di 2 allora $\log_2 x$ sarà proprio uguale a un valore intero quindi la differenza precedentemente espressa sarà sicuramente uguale a 0, visto che la base di un valore intero è uguale a sé stessa; quindi, in output si restituisce "true" (1).
In caso contrario, se x non è una potenza di 2, $\log_2 x$ non sarà un valore intero e di conseguenza $0 < \log_2 x - \lfloor \log_2 x \rfloor \leq 1$, per cui si restituisce in output "false" (0).

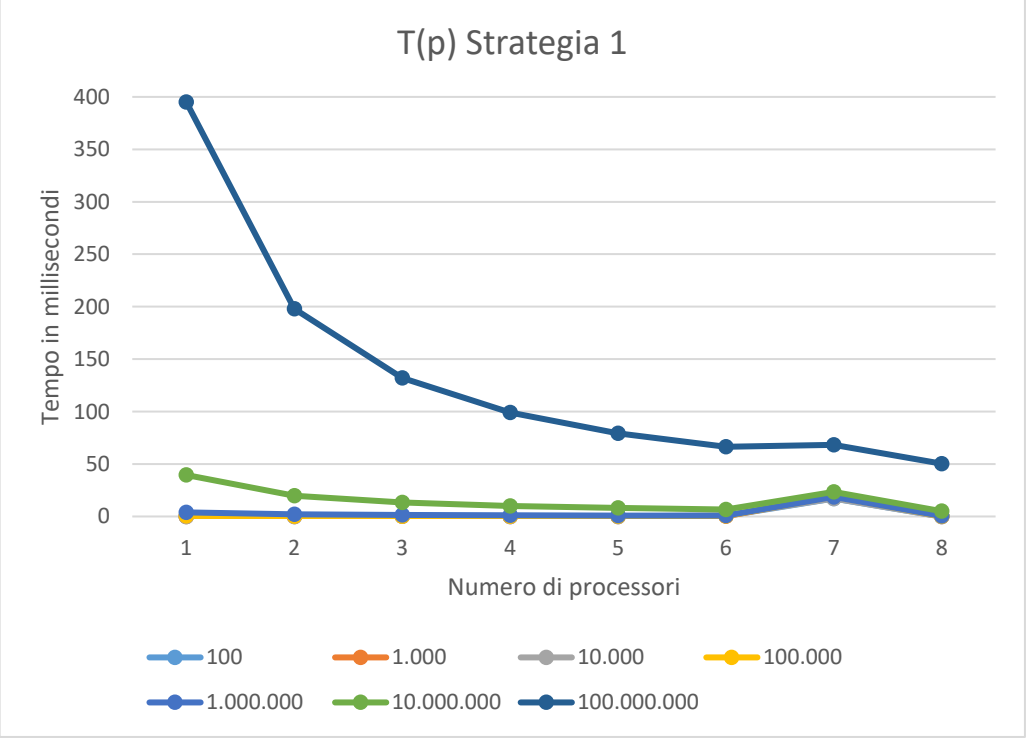
4. Analisi dei tempi

Tutti i tempi[1] di seguito riportati sono la media di 15 esecuzioni differenti, per ogni tipologia di strategia e per diversi tipi di input. I tempi sono espressi in millisecondi.

Strategia 1

Tabella dei tempi di esecuzione:

Numero CPU	Dimensione input						
	100	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000
1	0,002800	0,006866	0,047200	0,451600	3,906066	39,546200	395,132533
2	0,015266	0,016733	0,038933	0,212733	2,004666	19,865533	197,903733
3	0,090333	0,091533	0,106133	0,229866	1,414933	13,281933	132,024866
4	0,044400	0,039933	0,033200	0,124866	1,010666	9,970533	99,095333
5	0,021400	0,021333	0,028466	0,097200	0,817466	8,026333	79,414533
6	0,411133	0,376333	1,141400	0,485866	0,862600	6,715200	66,347733
7	17,063266	18,961466	17,169066	19,148333	18,676866	23,511333	68,212266
8	0,063866	0,065733	0,070133	0,106866	0,553200	5,043733	50,358266

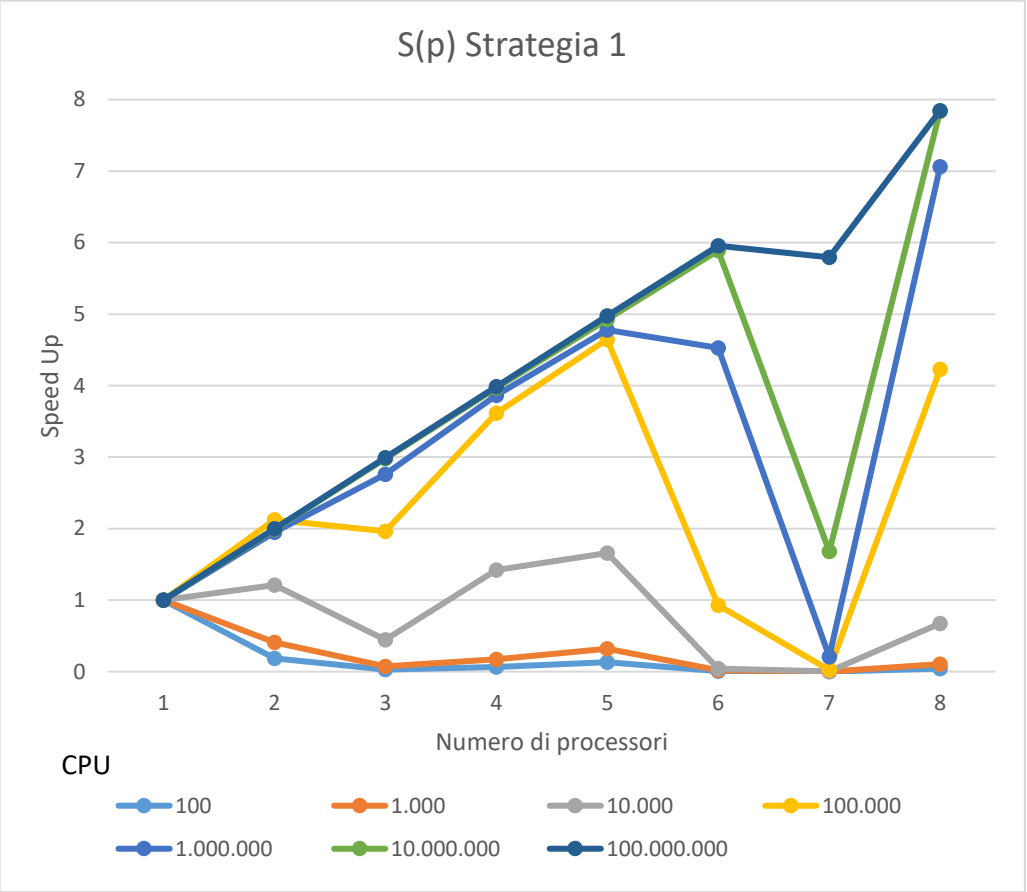


Come si può notare dal grafico, al crescere dalla dimensione dell'input si producono risultati in tempo minore se aumentiamo il grado di parallelismo, mentre per input piccoli, al variare del numero di processori, l'algoritmo si comporta quasi ugualmente, se non peggio, ad esempio dal grafico si può vedere che con 7 processori, per input di media o piccola dimensione, si impiega molto più tempo rispetto agli altri casi.

[1] In tutti i casi, i tempi indicati sono stati calcolati facendo stampare la somma totale a un solo processo.

Tabella speed-up:

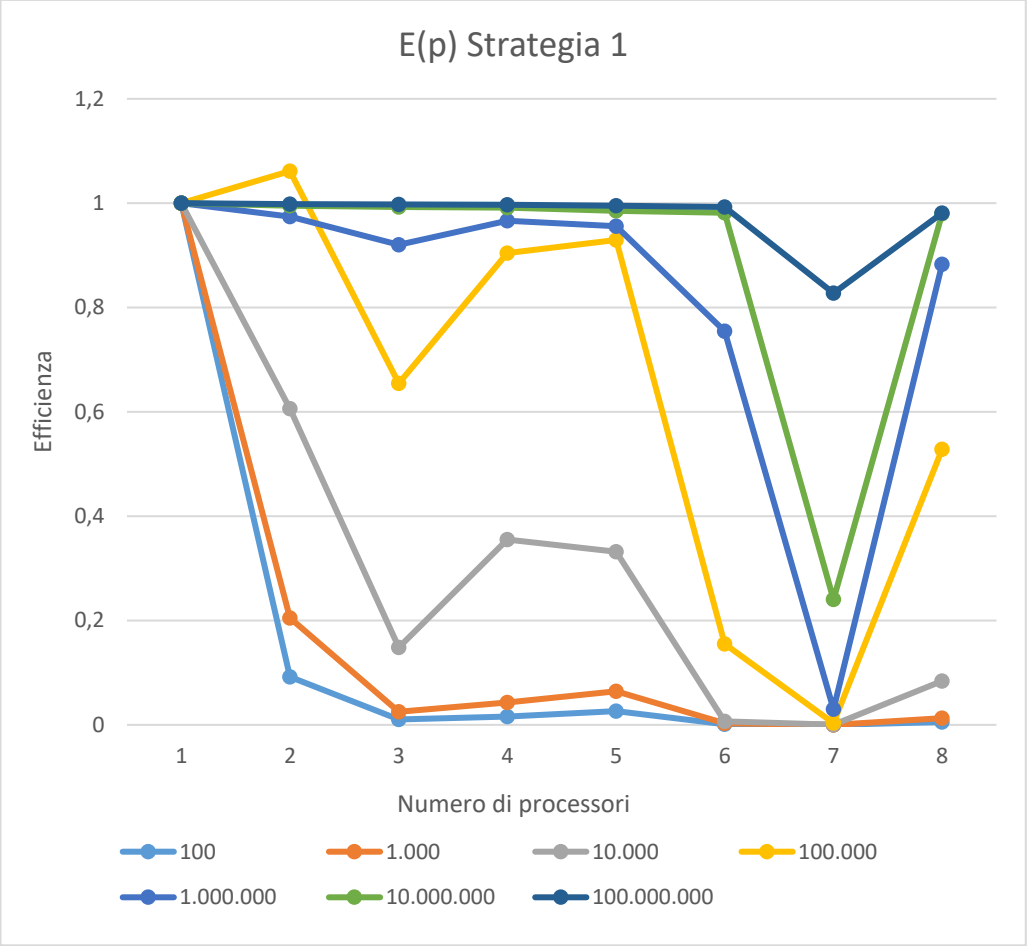
Numero CPU	Dimensione input						
	100	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000
1	1,000000	1,000000	1,000000	1,000000	1,000000	1,000000	1,000000
2	0,183414	0,410327	1,212339	2,122849	1,948487	1,990694	1,996590
3	0,030996	0,075011	0,444725	1,964623	2,760601	2,977443	2,992864
4	0,063063	0,171938	1,421687	3,616677	3,864844	3,966308	3,987398
5	0,130841	0,321849	1,658118	4,646091	4,778261	4,927057	4,975570
6	0,006810	0,018244	0,041353	0,929474	4,528247	5,889058	5,955479
7	0,000164	0,000362	0,002749	0,023584	0,209139	1,682006	5,792690
8	0,043842	0,104453	0,673007	4,225853	7,060857	7,840661	7,846428



Dal grafico dello speed-up si può osservare che la maggior riduzione del tempo di esecuzione si ha con un numero di processori compreso tra 2 e 5; infatti, nel grafico sopra riportato, le curve che mostrano il loro comportamento mettono in evidenza le differenze più grandi in termini di speed-up rispetto a un utilizzo con minor processi. A un certo punto lo speed-up cresce sempre più lentamente all'aumentare del numero di processori, raggiungendo il picco massimo con 8 processi che lavorano su una quantità di numeri da sommare sufficientemente grande, in particolare nel grafico di cui sopra quando si sommano 100 milioni di numeri. Escludendo l'anomalia del caso di 2 processi, dove lo speed-up supera quello ideale, per input molto grandi quasi tutti i processi si avvicinano di molto allo speed-up ideale, fatta eccezione del caso in cui si utilizzano 7 processi.

Tabella efficienza:

Numero CPU	Dimensione input						
	100	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000
1	1,000000	1,000000	1,000000	1,000000	1,000000	1,000000	1,000000
2	0,091707	0,205163	0,606170	1,061424	0,974244	0,995347	0,998295
3	0,010332	0,025004	0,148242	0,654874	0,920200	0,992481	0,997621
4	0,015766	0,042984	0,355422	0,904169	0,966211	0,991577	0,996850
5	0,026168	0,064370	0,331624	0,929218	0,955652	0,985411	0,995114
6	0,001135	0,003041	0,006892	0,154912	0,754708	0,981510	0,992580
7	0,000023	0,000052	0,000393	0,003369	0,029877	0,240287	0,827527
8	0,005480	0,013057	0,084126	0,528232	0,882607	0,980083	0,980804

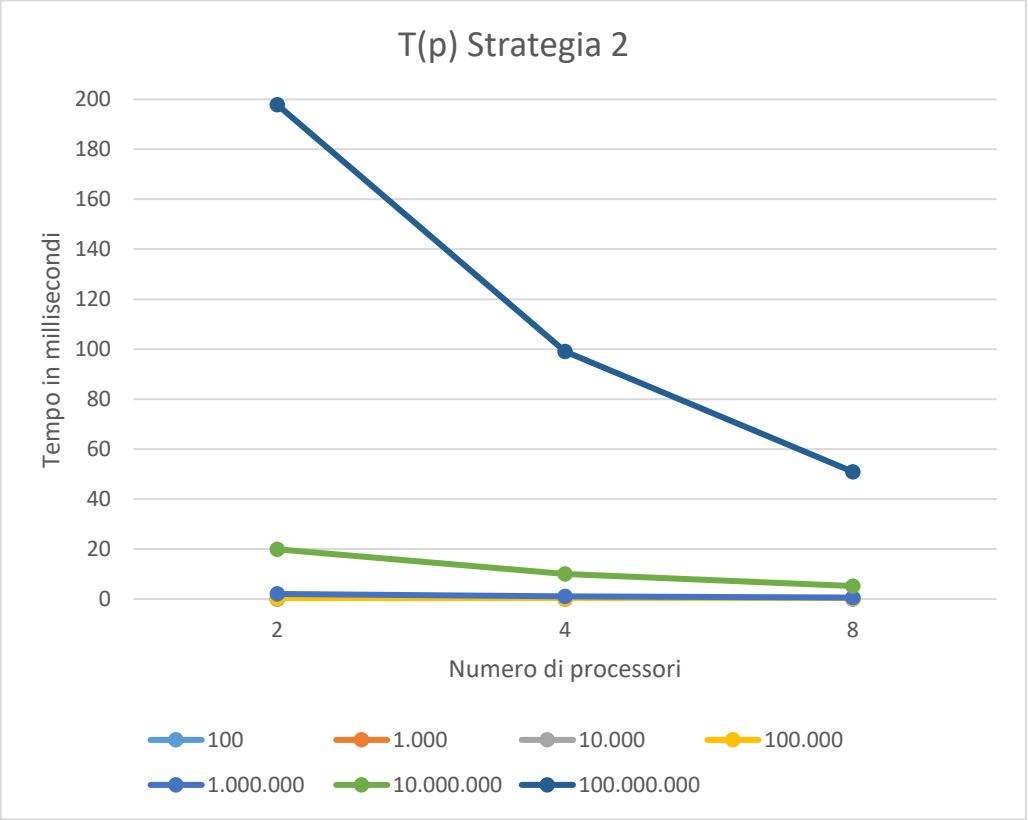


Dal grafico dell'efficienza si può osservare che il miglior utilizzo del parallelismo si manifesta in particolare con 2, 3, 4 e 5 processori; è bene notare tuttavia che nel caso di 2 processi si hanno alcuni comportamenti anomali sul grafico dell'efficienza, poiché essa supera la quantità massima che è pari a 1, tale anomalia potrebbe essere dovuta al fatto che come T(1) di riferimento si è scelto di prendere il caso in cui si usa l'algoritmo parallelo con un singolo processore. Notiamo che con l'aumentare dei processi, le curve iniziano a crescere sempre più lentamente, con un corrispondente calo di efficienza e quindi un minor sfruttamento delle risorse.

Strategia 2

Tabella dei tempi di esecuzione:

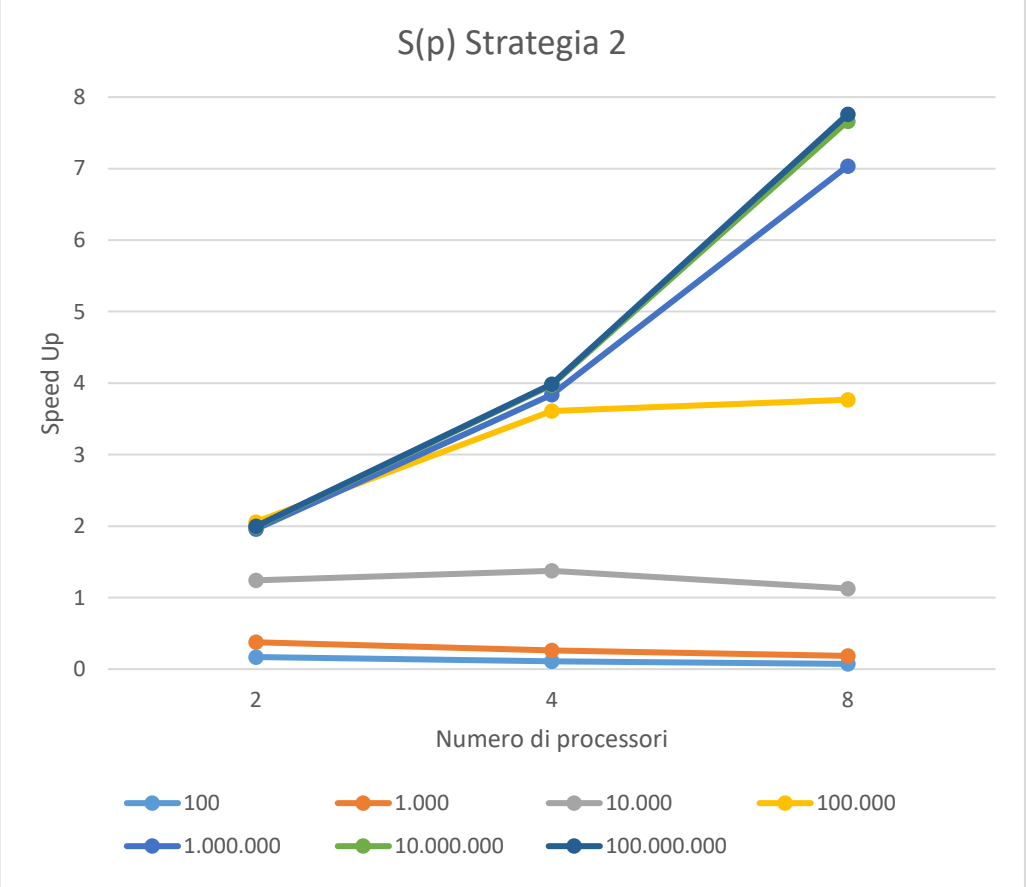
Numero CPU	Dimensione input						
	100	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000
2	0,016533	0,018266	0,038000	0,219400	1,995133	19,863333	197,842066
4	0,025933	0,026200	0,034333	0,125066	1,017866	9,955266	99,079666
8	0,038266	0,037066	0,041866	0,119866	0,555133	5,160733	50,923666



Come nella strategia 1, al crescere dalla dimensione dell'input si producono risultati in tempo minore se aumentiamo il grado di parallelismo, mentre per input piccoli (nel grafico di cui sopra è il caso di input fino a 1 milione di numeri) al variare del numero di processori l'algoritmo si comporta quasi ugualmente. La maggior differenza la si ottiene dunque per input molto grandi.

Tabella speed-up:

Numero CPU	Dimensione input						
	100	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000
2	0,169358	0,375890	1,242105	2,058341	1,957797	1,990915	1,997212
4	0,107971	0,262061	1,374771	3,610893	3,837505	3,972390	3,988029
8	0,073172	0,185237	1,127406	3,767540	7,036271	7,662904	7,759310

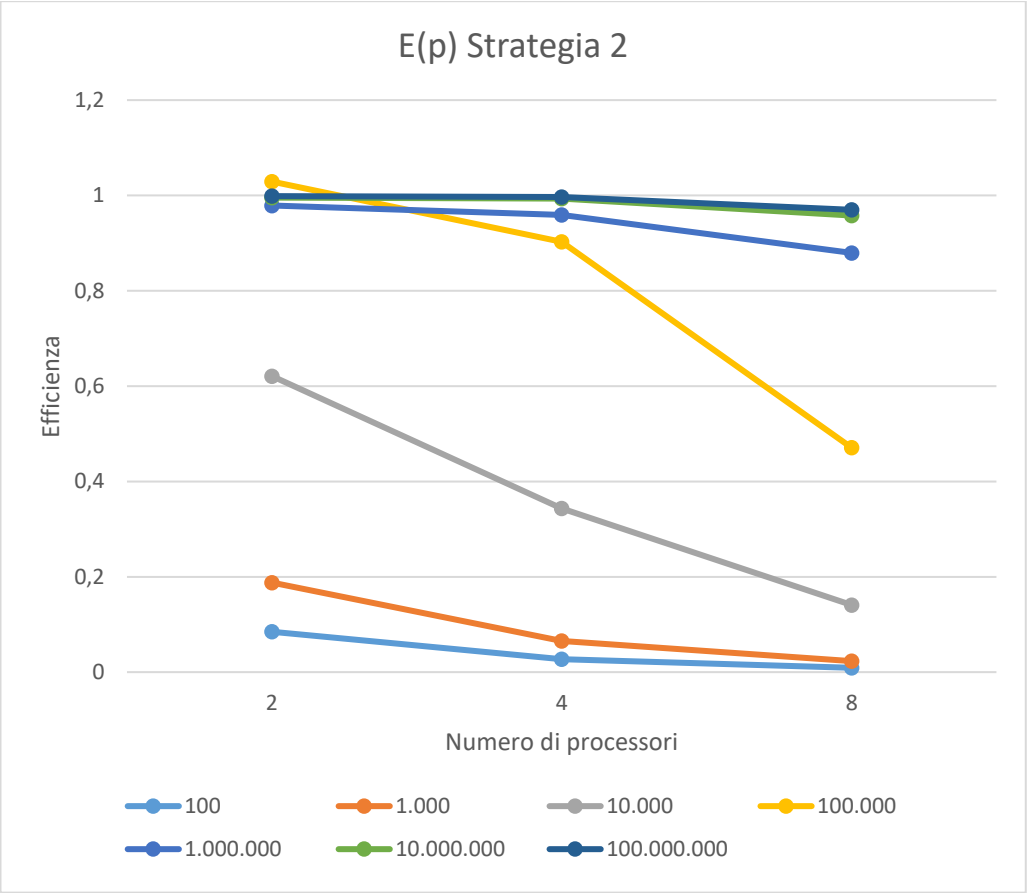


Per il calcolo dello speed-up sono stati utilizzati i valori T(1) della strategia 1.

Dal grafico dello speed-up si può osservare che rispetto alla strategia 1 c'è un leggero calo dello speed-up nel caso di 8 processi, soprattutto per input di taglia superiore o uguale a 1.000.000, questo perché seppur abbiamo aumentato il grado di parallelismo, e ridotto così il numero di passi totale necessario, dall'altro lato abbiamo introdotto un overhead dovuto all'aumento del numero di comunicazioni, che passa da P-1 nella strategia 1 a circa 2P-1 nella strategia 2.

Tabella efficienza:

Numero CPU	Dimensione input						
	100	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000
2	0,084679	0,187945	0,621053	1,029170	0,978899	0,995457	0,998606
4	0,026993	0,065515	0,343693	0,902723	0,959376	0,993098	0,997007
8	0,009147	0,023155	0,140926	0,470943	0,879534	0,957863	0,969914

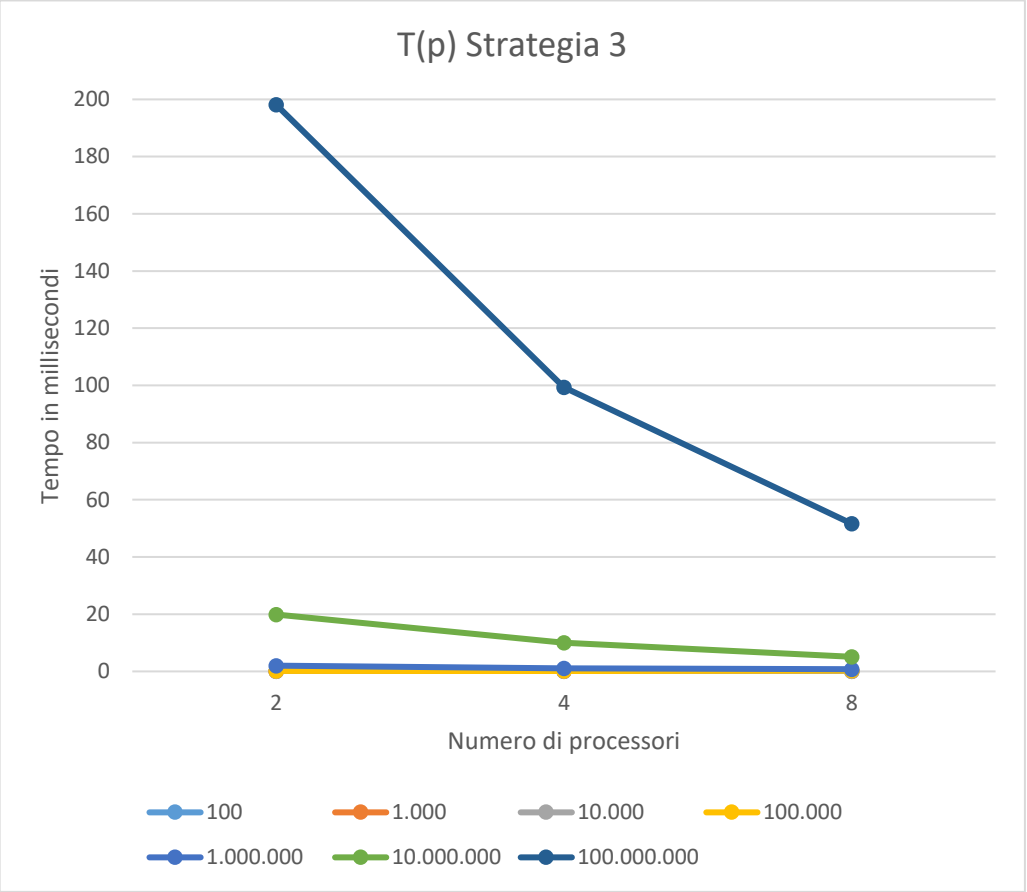


Dal grafico dell'efficienza si può osservare che rispetto alla strategia 1, per input di dimensione superiore o uguale a 1.000.000, abbiamo un'efficienza leggermente più bassa, in particolare nel caso di 8 processi; tuttavia, superata la soglia della dimensione di 1.000.000, l'efficienza rimane stabile e molto vicina a quella ideale, come nel caso della strategia 1. Notiamo come la curva dell'efficienza cresca molto lentamente a partire dalla soglia di 10.000.000 nel caso di 8 processi.

Strategia 3

Tabella dei tempi di esecuzione:

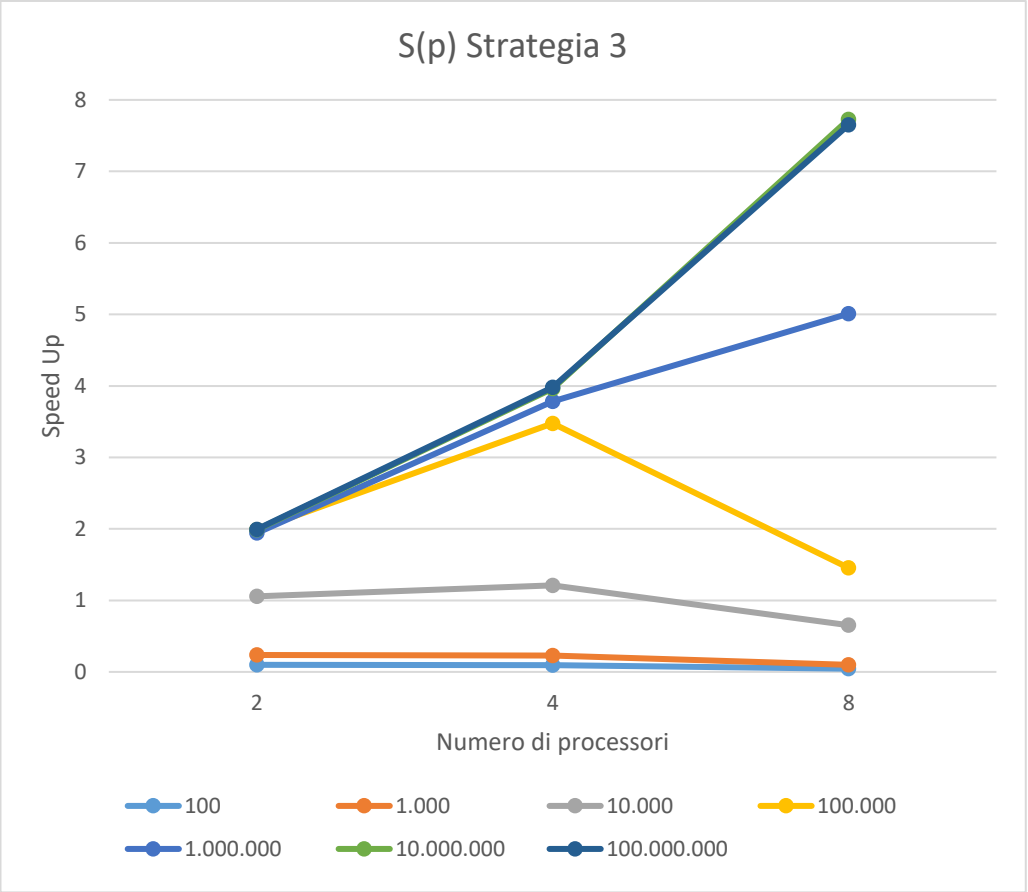
Numero CPU	Dimensione input						
	100	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000
2	0,028333	0,029000	0,044666	0,226933	2,011466	19,889266	198,114933
4	0,029933	0,029800	0,039000	0,129933	1,032200	9,980333	99,333200
8	0,062200	0,069333	0,072266	0,310333	0,779733	5,118666	51,64700



Come nella strategia 1 e 2, al crescere dalla dimensione dell'input si producono risultati in tempo minore aumentando il grado di parallelismo, mentre per input piccoli al variare del numero di processori l'algoritmo si comporta quasi ugualmente. In particolare, i tempi sopra illustrati non differiscono di molto da quelli riportati per la strategia 2, in particolare la strategia 3 produce risultati in leggermente più tempo rispetto alla strategia 2, ricordiamo che infatti la strategia 3 permette di ottenere il risultato finale in tutti i processori, a differenza della prima e della seconda strategia, ciò si traduce inevitabilmente in un maggior overhead dovuto al numero di comunicazioni che è chiaramente maggiore rispetto alle altre strategie.

Tabella speed-up:

Numero CPU	Dimensione input						
	100	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000
2	0,098825	0,236759	1,056732	1,990015	1,941900	1,988319	1,994461
4	0,093542	0,230403	1,210256	3,475637	3,784214	3,962413	3,977850
8	0,045016	0,099029	0,653143	1,455211	5,009492	7,725880	7,650639



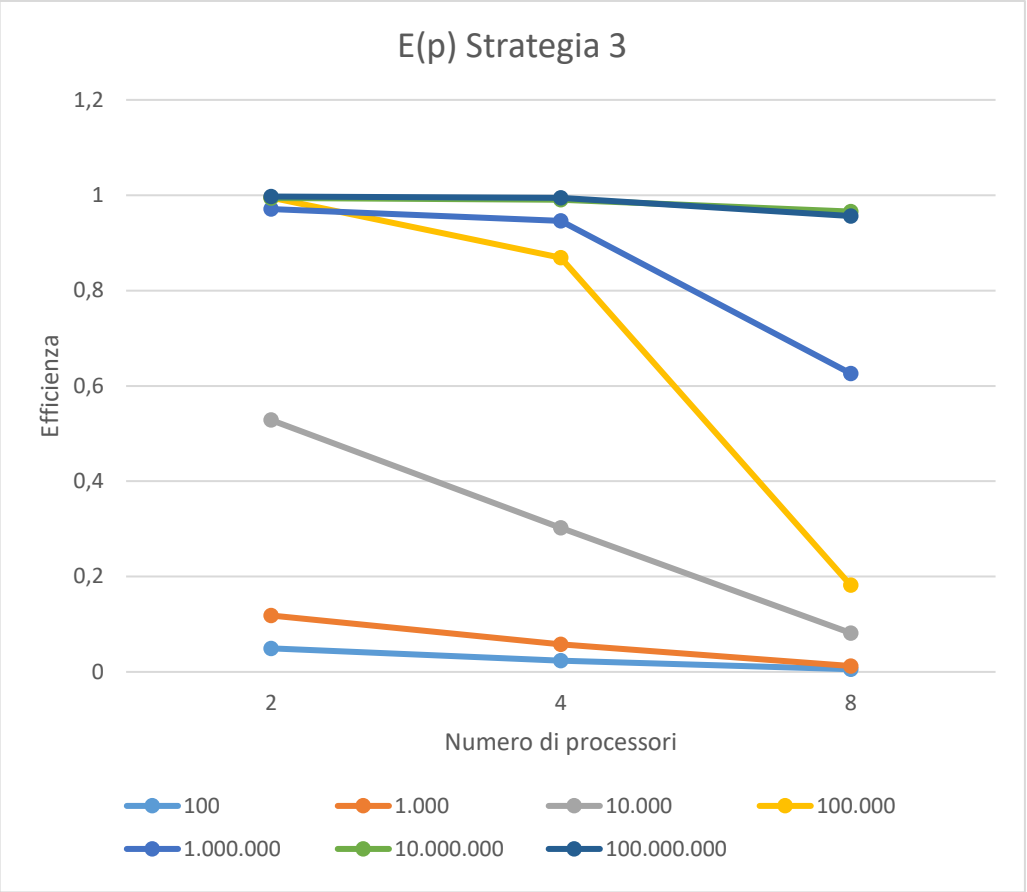
Come per la strategia 2, anche qui per il calcolo dello speed-up sono stati utilizzati i valori T(1) della strategia 1.

Valgono le stesse considerazioni fatte in merito allo speed-up ottenuto attraverso la strategia 2, ossia l'overhead maggiore dovuto a un numero di comunicazioni più alto rispetto alla strategia 1, e un grado di parallelismo maggiore.

La differenza rispetto allo speed-up ottenuto attraverso la strategia 2 è ben evidente, per i motivi riportati in precedenza attraverso il grafico dei tempi della strategia 3, i quali risultano leggermente maggiori rispetto a quelli ottenuti nella strategia 2, da cui derivano quindi speed-up minori in quei casi.

Tabella efficienza:

Numero CPU	Dimensione input						
	100	1.000	10.000	100.000	1.000.000	10.000.000	100.000.000
2	0,049412	0,118379	0,528366	0,995007	0,970950	0,994159	0,997231
4	0,023386	0,057601	0,302564	0,868909	0,946054	0,990603	0,994462
8	0,005627	0,012379	0,081643	0,181901	0,626186	0,965735	0,956330



Dal grafico dell'efficienza si può osservare che per input di taglia inferiore a 1.000.000, l'efficienza nel caso di 8 processori risulta molto inferiore rispetto a quella riportata per 2 e 4 processi; superata la soglia della dimensione di 1.000.000, invece, l'efficienza rimane stabile nel caso di 2 e 4 processi, e molto vicina a quella ideale, tuttavia, a differenza della strategia 1 e 2, nel caso di 8 processi si inizia a notare un calo dell'efficienza per input di taglia maggiore o uguale a 100.000.000; ricordiamo che infatti in questa strategia c'è un overhead dovuto alle comunicazioni necessarie all'ottenimento del risultato finale in tutti i processi.

5. Codice sorgente

Il codice sorgente di seguito illustrato implementa le tre strategie descritte nel capitolo 2, implementate attraverso l'ausilio del linguaggio di programmazione C.

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


/*********************************Costanti*********************************/

//Quantita' oltre la quale i valori in input vengono generati casualmente
#define MAX_INPUT 20

//Posizioni dei parametri presi in input dal main
#define ID 1
#define STRATEGIA 2
#define DIM 3
#define PRIMO_VALORE_INPUT 4

#define MAX_RAND 150
#define MIN_RAND -150


/*************************Prototipi funzioni*************************/

//Funzioni di gestione
void start(int argc, char *argv[], int idCpu, int numCpu);
int strategyChoice(int chiamante, int cpuPrint, int strat, int nCpu);
int * addendsDistribution(int idCpu, char * argv[], int * dim2, int * add);
int * sumArrayInit(int taglia, int argc, char *argv[]);
int localSum(int * addendi, int numAddendi);
int equivalentID(int idCpu, int idCpuPrint, int numCpu);
int equivalentSrcDst(int idCpu, int numCpu);
void printResults(int idCpu, int cpuPrint, int somma, double tempo);

//Strategie per sommare N numeri (interi) in parallelo
int strategy1(int chiamante, int cpuPrint, int numCpu, int * addendi, int numAddendi, double * t);
int strategy2(int chiamante, int cpuPrint, int numCpu, int * addendi, int numAddendi, double * t);
int strategy3(int chiamante, int cpuPrint, int numCpu, int * addendi, int numAddendi, double * t);

//Funzioni di utility
int generateRandom(int min, int max);
double logarithm(double base, double argomento);
int * calculatePowers(int base, int numPotenze);
int testPowerOfTwo(double x);

/*********************************************************************/
```

```c
int main(int argc, char *argv[]) {
    int menum, numCpu;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&menum);
    MPI_Comm_size(MPI_COMM_WORLD,&numCpu);

    /* Controllo che il numero di elementi da sommare sia valido,
     * in caso contrario viene segnalato l'errore e si termina il programma */
    if(atoi(argv[DIM]) < 1) {
        if(menum == atoi(argv[ID]))
            printf("La quantita' di valori da sommare deve essere positiva!\n\n");
        MPI_Finalize();
        return 1;
    }
    else if(atoi(argv[DIM]) < numCpu) {
        if(menum == atoi(argv[ID]))
            printf("Attenzione!\nLa quantita' di valori da sommare deve essere"
            " almeno pari al numero di CPU! (Numero di CPU: %d)\n", numCpu);
        MPI_Finalize();
        return 1;
    }
    else if(atoi(argv[ID]) < -1 || atoi(argv[ID]) > (numCpu-1)) {
        if(menum == 0)
            printf("Attenzione!\nL'ID del processore che stampa la somma deve "
            "essere compreso tra -1 e %d (numero CPU - 1)!\n", numCpu-1);
        MPI_Finalize();
        return 1;
    }
    else {
        srand(time(NULL));
        //Se il numero di elementi e' corretto, si prosegue
        start(argc, argv, menum, numCpu);
    }

    MPI_Finalize();

    return 0;
}


/*************************Funzioni di Gestione*************************/

void start(int argc, char *argv[], int idCpu, int numCpu) {
    int idCpuPrint = atoi(argv[ID]);
    int idCpuMaster;
    int strategia = atoi(argv[STRATEGIA]);
    int dim, dimPersonale, risultato;

    int * addendi;
    int * addendiPersonali;

    double tempoImpiegato;

    /* Se in input ci viene richiesto che tutti i processori stampino
     * la somma totale, impostiamo P0 come il processore
     * incaricato dello scambio dei messaggi */
    if(idCpuPrint == -1) {
        idCpuMaster = 0;
    }
```

```
109     else {
110         idCpuMaster = idCpuPrint;
111     }
112
113     if(idCpu == idCpuMaster) {
114         dim = atoi(argv[DIM]);
115         //Si popola l'array degli addendi
116         addendi = sumArrayInit(dim, argc, argv);
117     }
118
119     /* Comunichiamo agli altri processi (tramite una comunicazione broadcast)
120      * il numero di elementi da sommare */
121     MPI_Bcast(&dim, 1, MPI_INT, idCpuMaster, MPI_COMM_WORLD);
122
123     //Si distribuiscono i vari elementi da sommare ad ogni processo
124     addendiPersonali = addendsDistribution(idCpu, argv, &dimPersonale, addendi);
125
126     int strategia_Scelta = strategyChoice(idCpu, idCpuMaster, strategia, numCpu);
127
128     if(strategia_Scelta != 0) {
129         //Si chiama la funzione che sommera' con una determinata politica
130         if(strategia_Scelta == 1) {
131             risultato =
132                 strategy1
133                     (idCpu, idCpuMaster, numCpu, addendiPersonali, dimPersonale, &tempoImpiegato);
134         }
135         else if(strategia_Scelta == 2) {
136             risultato =
137                 strategy2
138                     (idCpu, idCpuMaster, numCpu, addendiPersonali, dimPersonale, &tempoImpiegato);
139         }
140         else { //strategia_Scelta == 3
141             risultato =
142                 strategy3
143                     (idCpu, idCpuMaster, numCpu, addendiPersonali, dimPersonale, &tempoImpiegato);
144         }
145
146         //Se tutti i processi stampano il risultato, inviamo a essi il tempo totale impiegato
147         if(idCpuPrint == -1) {
148             MPI_Bcast(&tempoImpiegato, 1, MPI_DOUBLE, idCpuMaster, MPI_COMM_WORLD);
149         }
150
151         /* Se ci viene richiesto che tutti i processori stampino il risultato e la
152          * strategia richiesta è stata la prima o la seconda, facciamo recuperare
153          * il risultato completo a tutti i processori diversi da idCpuMaster (= 0) */
154         if(idCpuPrint == -1 && strategia_Scelta != 3) {
155             MPI_Bcast(&risultato, 1, MPI_INT, 0, MPI_COMM_WORLD);
156         }
157
158         printResults(idCpu, idCpuPrint, risultato, tempoImpiegato);
159     }
160     else {
161         //La scelta effettuata non e' corretta
162         MPI_Finalize();
163         exit(1);
164     }
165 }
```

```c
/* Funzione che restituisce in output la strategia da applicare,
 * oppure restituisce 0 in caso di incorrettezza dell'input */
int strategyChoice(int chiamante, int cpuPrint, int strat, int nCpu) {
    if(strat == 1 || (!testPowerOfTwo(nCpu) && (strat == 2 || strat == 3))) {
        if(strat != 1 && chiamante == cpuPrint) {
            printf("Attenzione!\nCon %d processi non e' possibile applicare "
            "la strategia %d!\nSara' applicata la strategia 1.\n", nCpu, strat);
        }
        return 1;
    }
    else if(strat == 2) {
        return 2;
    }
    else if(strat == 3) {
        return 3;
    }
    else {
        if(chiamante == cpuPrint) {
            printf("Attenzione!\nStrategia di somma non esistente!\n\n");
        }
        return 0;
    }
}

/* Distribuisce ad ogni processo il sottoinsieme di valori da sommare,
 * prima di fare cio' viene allocato l'array che conterra' tali valori,
 * che viene poi restituito in output */
int * addendsDistribution(int idCpu, char * argv[], int * dim2, int * add) {
    int idCpuPrint = atoi(argv[ID]);
    int dim = atoi(argv[DIM]);

    int numCpu, resto, start, offset, i, tag, idSrc;
    int * v;

    MPI_Status status;

    MPI_Comm_size(MPI_COMM_WORLD, &numCpu);

    if(idCpuPrint == -1) {
        idCpuPrint = 0;
    }

    //Comunico "esternamente" quanti valori verranno inseriti nell'array
    *(dim2) = (dim / numCpu);
    resto = (dim % numCpu);

    /* Alcuni processi conterranno un elemento in meno nell'array,
     * quindi aggiorniamo la loro dimensione.
     * Cio' puo' accadere nel caso in cui il numero di elementi da sommare non
     * e' un multiplo del numero dei processi*/
    if(equivalentID(idCpu, idCpuPrint, numCpu) < resto) {
        (*(dim2))++;
    }

    //Tutti i processi, tranne quello "principale", allocano il proprio array
    if(idCpu != idCpuPrint) {
        v = (int *)malloc(sizeof(int) * (*(dim2)));
    }
```

```
226     if(idCpu == idCpuPrint) {
227         /* Il processo principale leggera' i valori da sommare
228          * dall'array principale, poiche' e' una sua copia personale */
229         v = add;
230 
231         offset = *(dim2);
232         start = 0;
233 
234         /* Il processo principale invia i valori da sommare ad ogni processo
235          * (escluso se stesso) ad intervalli che iniziano in start e terminano in offset - 1 */
236         for(i = 1; i < numCpu; i++) {
237             /* Per calcolare l'id del destinatario della somma parziale si passa idCpu + i,
238              * poiche' alla prima iterazione si inoltra la somma parziale a idCpuPrint + 1,
239              * alla seconda iterazione a idCpuPrint + 2 e cosi via */
240             idSrc = equivalentSrcDst(i+idCpu, numCpu);
241             start += offset;
242             tag = idSrc + 100;
243 
244             if(i == resto) {
245                 offset--;
246             }
247 
248             MPI_Send(&add[start], offset, MPI_INT, idSrc, tag, MPI_COMM_WORLD);
249         }
250     }
251     else {
252         /* Tutti i processi, tranne quello principale, ricevono i valori
253          * da sommare e li memorizzano in un array personale */
254         tag = 100 + idCpu;
255         MPI_Recv(v, (*(dim2)), MPI_INT, idCpuPrint, tag, MPI_COMM_WORLD, &status);
256     }
257 
258     return v;
259 }
260 
261 /* Questa funzione alloca un array per poi riempirlo.
262  * Se si devono sommare piu' di 20 valori si riempie con valori casuali (oppure tutti 1),
263  * altrimenti viene riempito con valori presi in input dal main */
264 int * sumArrayInit(int taglia, int argc, char * argv[]) {
265     int * v = (int *)malloc(sizeof(int) * taglia);
266     int i;
267 
268     if(taglia <= MAX_INPUT) {
269         printf("\nValori da sommare: \n");
270         for(i = PRIMO_VALORE_INPUT; i < argc; i++) {
271             v[(i - PRIMO_VALORE_INPUT)] = atoi(argv[i]);
272             printf("%d\t", v[(i - PRIMO_VALORE_INPUT)]);
273         }
274     }
275     else {
276         //printf("\nValori da sommare: \n");
277         for(i = 0; i < taglia; i++) {
278             //v[i] = generateRandom(MIN_RAND, MAX_RAND);
279             v[i] = 1;
280             //printf("%d\t", v[i]);
281         }
282     }
283 
284     return v;
285 }
```

```c
//Funzione che effettua la somma di N valori interi
int localSum(int * addendi, int numAddendi) {
    int sommaParziale = 0, i;

    for(i = 0; i < numAddendi; i++) {
        sommaParziale += addendi[i];
    }

    return sommaParziale;
}

/* Presi in input il numero del processo, il numero del processo che deve
 * "stampare" e la quantita' di processi in esecuzione, rende idCpuPrint
 * equivalente all'id che avrebbe tale processo se il processo principale fosse zero. */
int equivalentID(int idCpu, int idCpuPrint, int numCpu) {
    return ((idCpu - idCpuPrint + numCpu) % numCpu);
}

/* Preso in input l'id di un processo che dovrebbe inviare o ricevere un messaggio,
 * produce in output l'id del processo che effettivamente dovra' effettuare tale operazione */
int equivalentSrcDst(int idCpu, int numCpu) {
    return ((numCpu + idCpu) % numCpu);
}

void printResults(int idCpu, int cpuPrint, int somma, double tempo) {
    if(idCpu == cpuPrint || cpuPrint <= -1) {
        if(cpuPrint <= -1 && idCpu != 0) {
            printf("\n\n\tIl processo P%d ha calcolato la somma %d\n", idCpu, somma);
        }
        else {
            printf("\n\n\tIl processo P%d ha calcolato la somma %d in %f secondi\n", idCpu, somma, tempo);
        }
    }
}


/*********************Funzioni di somma parallela**********************/

int strategy1(int chiamante, int cpuPrint, int numCpu, int * addendi, int numAddendi, double * t) {
    int sommaParziale;
    int sommaTotale, tag, idSrc;

    double t0, t1, time;

    MPI_Status status;

    MPI_Barrier(MPI_COMM_WORLD);
    t0 = MPI_Wtime();

    sommaParziale = localSum(addendi, numAddendi);

    /* Nel primo ramo dell'if ci entreranno tutti i processi tranne quello principale,
     * essi comunicheranno la loro somma parziale al processo principale, che la sommera' alla propria */
    if(chiamante != cpuPrint) {
        tag = 200 + chiamante;
        MPI_Send(&sommaParziale, 1, MPI_INT, cpuPrint, tag, MPI_COMM_WORLD);
    }
    else {
        int i;
        sommaTotale = sommaParziale;
        for(i = 1; i < numCpu; i++) {
            idSrc = equivalentSrcDst(i + chiamante, numCpu);
            tag = 200 + idSrc;
            MPI_Recv(&sommaParziale, 1, MPI_INT, idSrc, tag, MPI_COMM_WORLD, &status);
            sommaTotale += sommaParziale;
        }
    }

    t1=MPI_Wtime();
    time = t1 - t0;
    MPI_Reduce(&time, t, 1, MPI_DOUBLE, MPI_MAX, cpuPrint, MPI_COMM_WORLD);

    if(chiamante == cpuPrint) {
        return sommaTotale;
    }
    else {
        return sommaParziale;
    }
}
```

```c
int strategy2(int chiamante, int cpuPrint, int numCpu, int * addendi, int numAddendi, double * t) {
    int sommaParziale;
    int sommaTmp = 0, tag, i, shiftId;
    double log2nCpu;

    int * potenzeDi2 = NULL;

    double t0, t1, time;

    MPI_Status status;

    /* Un solo processo calcola i dati che verrano utilizzati per la somma parallela,
     * questi dati vengono calcolati in anticipo per non aggiungere overhead alla somma. */
    if(chiamante == cpuPrint) {
        log2nCpu = logarithm(2, numCpu);
        potenzeDi2 = calculatePowers(2, (log2nCpu + 1));
    }

    //Al termine il processore che li ha calcolati li spedisce agli altri processori
    MPI_Bcast(&log2nCpu, 1, MPI_INT, cpuPrint, MPI_COMM_WORLD);

    /* Prima di ricevere le potenze di due dal processo principale,
     * i processi devono allocare il proprio array */
    if(chiamante != cpuPrint) {
        potenzeDi2 = (int *) malloc(sizeof(int) * (log2nCpu + 1));
    }
    MPI_Bcast(potenzeDi2, (log2nCpu + 1), MPI_INT, cpuPrint, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);
    t0 = MPI_Wtime();

    sommaParziale = localSum(addendi, numAddendi);

    /* I processi a coppie calcolano una somma parziale, ad ogni iterazione il
     * numero di coppie si dimezza, costruendo un "albero capovolto" dove le
     * foglie sono i valori in input e la somma e' la radice dell'albero,
     * mentre i nodi interni sono le somme parziali calcolate per
     * arrivare alla somma finale */
    for(i = 0; i < log2nCpu; i++) {
        shiftId = equivalentID(chiamante, cpuPrint, numCpu);
        if(shiftId % potenzeDi2[i] == 0) {
            if(shiftId % potenzeDi2[i + 1] != 0) {
                tag = 300 + chiamante;
                MPI_Send(&sommaParziale, 1, MPI_INT,
                    equivalentSrcDst((chiamante - potenzeDi2[i]), numCpu), tag, MPI_COMM_WORLD);
            }
            else {
                sommaTmp = sommaParziale;
                tag = 300 + equivalentSrcDst((chiamante + potenzeDi2[i]), numCpu);
                MPI_Recv(&sommaParziale , 1, MPI_INT,
                    equivalentSrcDst(chiamante + potenzeDi2[i], numCpu), tag, MPI_COMM_WORLD, &status);
                sommaParziale += sommaTmp;
            }
        }
    }

    t1=MPI_Wtime();
    time = t1 - t0;
    MPI_Reduce(&time, t, 1, MPI_DOUBLE, MPI_MAX, cpuPrint, MPI_COMM_WORLD);

    if(potenzeDi2 != NULL) {
        free(potenzeDi2);
    }

    return sommaParziale;
}
```

```c
int strategy3(int chiamante, int cpuPrint, int numCpu, int * addendi, int numAddendi, double * t) {
    int sommaParziale;
    int sommaTmp = 0, tag, i;
    double log2nCpu;

    double t0, t1, time;

    int * potenzeDi2 = NULL;

    MPI_Status status;

    /* Un solo processo calcola i dati che verrano utilizzati per la somma parallela,
     * questi dati vengono calcolati in anticipo per non aggiungere overhead alla somma. */
    if(chiamante == cpuPrint) {
        log2nCpu = logarithm(2, numCpu);
        potenzeDi2 = calculatePowers(2, (log2nCpu + 1));
    }

    //Al termine, il processore che ha calcolato i dati, li spedisce agli altri processori
    MPI_Bcast(&log2nCpu, 1, MPI_INT, cpuPrint, MPI_COMM_WORLD);

    if(chiamante != cpuPrint) {
        potenzeDi2 = (int *) malloc(sizeof(int) * (log2nCpu + 1));
    }
    MPI_Bcast(potenzeDi2, (log2nCpu + 1), MPI_INT, cpuPrint, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);
    t0 = MPI_Wtime();

    sommaParziale = localSum(addendi, numAddendi);

    /* Come per la strategia 2 ma in questo caso c'e' un doppio scambio di messaggi in ingresso e uscita,
     * al termine tutti i processi avranno una propria copia della somma, al contrario della strategia 2 */
    for(i = 0; i < log2nCpu; i++) {
        if((equivalentID(chiamante, cpuPrint, numCpu) % potenzeDi2[i + 1]) < potenzeDi2[i]) {
            tag = ((i*1000) + (400 + chiamante));
            MPI_Send(&sommaParziale, 1, MPI_INT,
                equivalentSrcDst((chiamante + potenzeDi2[i]), numCpu), tag, MPI_COMM_WORLD);

            tag = ((i*1000) + (500 + equivalentSrcDst((chiamante + potenzeDi2[i]), numCpu)));
            MPI_Recv(&sommaTmp , 1, MPI_INT,
                equivalentSrcDst((chiamante + potenzeDi2[i]), numCpu), tag, MPI_COMM_WORLD, &status);

            sommaParziale += sommaTmp;
        }
        else {
            tag = ((i*1000) + (400 + equivalentSrcDst((chiamante - potenzeDi2[i]), numCpu)));
            MPI_Recv(&sommaTmp , 1, MPI_INT,
                equivalentSrcDst((chiamante - potenzeDi2[i]), numCpu), tag, MPI_COMM_WORLD, &status);

            tag = ((i*1000 +(500 + chiamante)));
            MPI_Send(&sommaParziale, 1, MPI_INT,
                equivalentSrcDst((chiamante - potenzeDi2[i]), numCpu), tag, MPI_COMM_WORLD);

            sommaParziale += sommaTmp;
        }
    }

    t1=MPI_Wtime();
    time = t1 - t0;
    MPI_Reduce(&time, t, 1, MPI_DOUBLE, MPI_MAX, cpuPrint, MPI_COMM_WORLD);

    if(potenzeDi2 != NULL)  {
        free(potenzeDi2);
    }

    return sommaParziale;
}
```

```c


/*************************Funzioni di utility ************************/

//Funzione che alloca un array e lo riempie con potenze contigue partendo da base^0
int * calculatePowers(int base, int numPotenze) {
    int * potenze = (int *)malloc(sizeof(int) * numPotenze);
    int i;

    potenze[0] = 1;
    for(i = 1; i < numPotenze; i++) {
        potenze[i] = potenze[i - 1] * base;
    }

    return potenze;
}

//Funzione che calcola il logarithm in una qualsiasi base positiva
double logarithm(double base, double argomento) {
    return (log10(argomento)/log10(base));
}

//Funzione che controlla se un valore e' una potenza di due
int testPowerOfTwo(double x) {
    return ((logarithm(2, x) - (floor(logarithm(2, x)))) == 0);
}

//Generazione di valori casuali compresi nell'intervallo [min, max]
int generateRandom(int min, int max) {
    return ((rand() % (max - min + 1)) + min);
}

/*********************************************************************/
```