Implementazione operazione "saxpy" in ambiente MPI-Docker ($c = \alpha \cdot a + b$)

Denny Caruso[1]

1: denny.caruso001@studenti.uniparthenope.it

1. Definizione e analisi del problema

Si vuole realizzare un prodotto software che implementi in *parallelo* l'operazione nota come *saxpy* in ambiente MIMD-DM mediante l'utilizzo della libreria MPI. In particolare, si vuole determinare il vettore risultato c ottenuto come la somma fra il vettore b e il vettore a che viene moltiplicato per uno scalare α. Le operazioni sono elementari dal punto di vista dell'algebra lineare e fondamentalmente consistono nell'effettuare il prodotto di un vettore per uno scalare e successivamente, il vettore risultante da questa operazione, viene sommato a un altro vettore. Ovviamente i due vettori di input e il vettore di output sono delle stesse dimensioni.

L'algoritmo sequenziale può essere banalmente realizzato con un singolo *loop* che effettua tante iterazioni quanti sono gli elementi in input in uno dei due vettori in input. All'interno di ogni iterazione si eseguirà la somma di un vettore con l'altro vettore che viene anticipatamente moltiplicato per uno scalare.

Keywords: MPI, *saxpy*

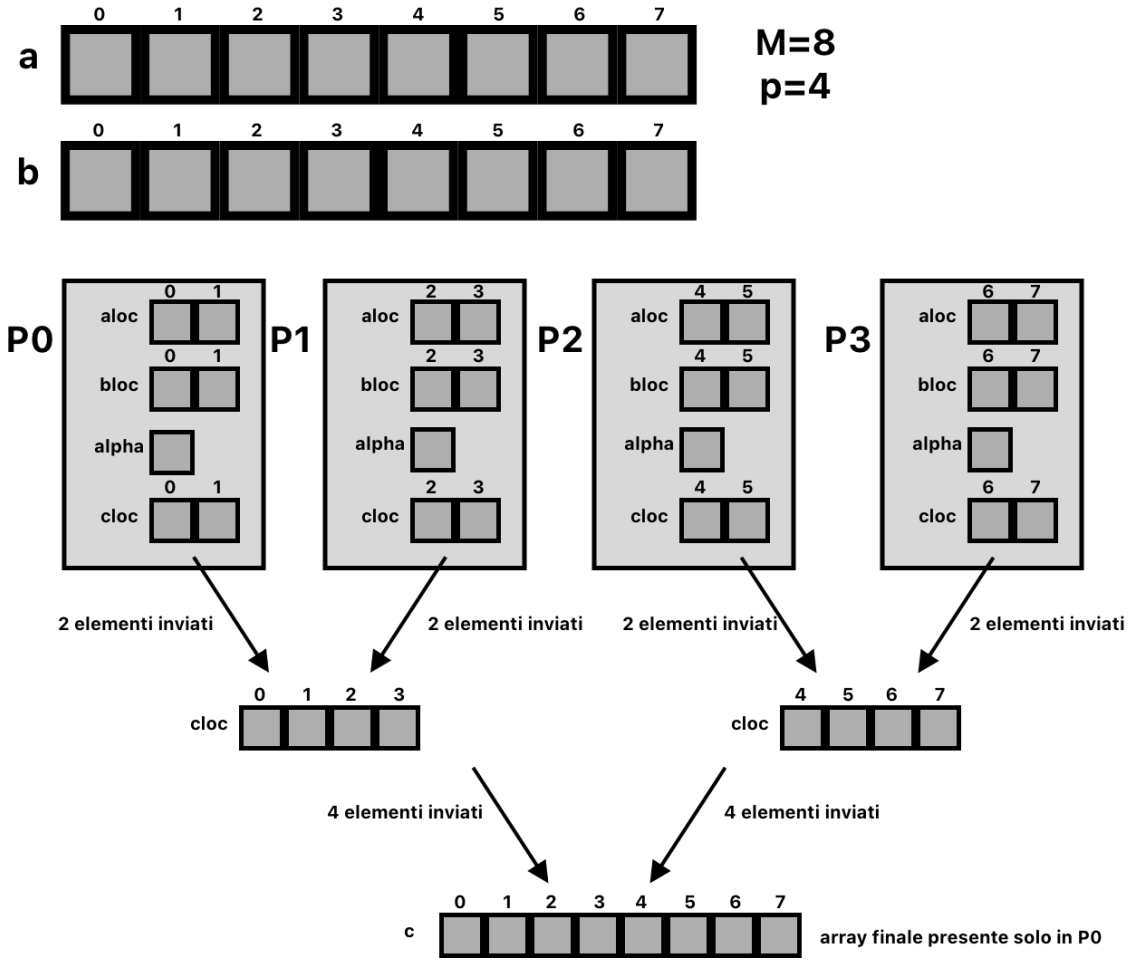
2. Descrizione dell'approccio parallelo

La strategia di parallelizzazione adottata è molto semplice. Dati p processori e due array di dimensione M contenenti i dati di input, è possibile assegnare ad ogni processore del cluster una porzione di $\frac{M}{p}$ elementi di ambedue gli array. Inoltre, ogni processore avrà a disposizione tutto il necessario per effettuare la propria fase di calcolo locale, ovvero lo scalare α coinvolto nell'operazione di *saxpy*, la dimensione della porzione degli array ed eventuali offset da considerare qualora la dimensione del problema non sia esattamente divisibile per il numero di processori. Una volta terminata la fase di distribuzione delle porzioni dei due array e degli altri dati appena menzionati dal processore *master* a tutti gli altri processori, ogni processore effettua localmente l'operazione di *saxpy* tra le due porzioni di vettori a disposizione e memorizza il risultato nel vettore risultato locale al processore.

Ora non resta che unire in maniera ordinata tutte le porzioni di array locali calcolate da ogni processore e ottenere l'array completo che sarà disponibile solo all'interno del processore *master*. Di fatto, si implementa il *modello di collezione dei risultati ad albero binario*. Questo tipo di approccio è stato scelto, in quanto permette di raggiungere dei risultati migliori rispetto alla collezione puramente sequenziale dei risultati. Alternativamente, si può fare in modo che il vettore risultante sia disponibile a tutte le unità processanti del cluster. Fare ciò però, avrebbe comportato un approccio algoritmico differente rispetto a quello illustrato nella sezione successiva, oppure un'operazione di *broadcast* finale del vettore risultante (ottenuto dal processore *master*) a tutti gli altri processori.

Ora si passa alla valutazione dell'approccio parallelo utilizzato. Nel caso più semplice in cui non è richiesto ricevere il vettore risultato in unico vettore, ma basta aver effettuato l'operazione di *saxpy* sulle varie porzioni di array distribuite, allora l'approccio parallelo risulta rientrare nella classe degli algoritmi paralleli perfetti, ovvero con speed-up pari a p, efficienza pari a 1 e overhead pari a 0. Diversamente, se necessario ottenere il vettore risultato per intero, allora la situazione cambia. In ogni caso, si precisa che nelle dimostrazioni che seguono si assume che i dati di input siano già distribuiti ai processori del cluster.

Prima di procedere, si allega di seguito un'illustrazione dell'approccio parallelo adottato quando è necessario ottenere il risultato in un unico array finale.



Si consideri M come la dimensione del singolo vettore a e del vettore b. Di conseguenza la somma delle due dimensioni vale $2M$ che indichiamo con N. La complessità computazionale con $p = 1$ dell'algoritmo sequenziale è pari a $T_1(N) = N \cdot t_{calc} = M \cdot t_{addizione} + M \cdot t_{moltiplicazione}$. Siccome il tempo per fare un'addizione e una moltiplicazione asintoticamente risultano essere simili, allora si possono raggruppare le due quantità.

Nel caso in cui non si consideri la necessità di riunire i dati in unico vettore di output, si ottiene:

$$T_p(N) = \frac{N}{p} \cdot t_{calc} = \frac{2M}{p} \cdot t_{calc}$$

Dove t_{calc} è il tempo di esecuzione di un'operazione floating point su una generica macchina. Da quest'ultima equazione segue che lo speed-up è modellato come:

$$S_p(N) = \left(\frac{N}{\frac{N}{p}} \right) = p$$

quindi l'overhead è:

$$O_p(N) = \left(p \cdot \frac{N}{p} - N \right) \cdot t_{calc} = 0$$

E l'efficienza risulta essere:

$$E_p(N) = \frac{p}{p} = 1$$

L'isoefficienza risultante in tal caso è la forma indeterminata $\left(\frac{0}{0}\right)$, e quindi qualsiasi dimensione del problema si sceglie, risulta ottimale. Infine, dall'applicazione della forma base della legge di Ware-Amdhal (viste le circostanze), si ottiene che:

$$S_p(N) = \frac{1}{\alpha + \left(\frac{1-\alpha}{p}\right)} = p$$

dal momento che:

$$\left\{(1-\alpha) = \left(p \cdot \frac{\frac{N}{p}}{N}\right) = 1\right\} \Leftrightarrow \{\alpha = 0\}$$

Supponiamo che sia necessario raccogliere le porzioni del vettore c calcolate localmente da ogni processore del cluster in un unico vettore. Per semplicità si assume un numero di processori esattamente pari a una potenza del due. In tal caso e considerando anche la non esatta divisibilità della dimensione del problema per il numero dei processori del cluster, si ottiene che:

$$T_p(N) = \left[2\left\lceil\frac{M}{p}\right\rceil + c \cdot \left(\sum_{i=1}^{log_2(p)} \left\lceil\frac{M}{p}\right\rceil^i\right)\right] \cdot t_{calc}$$

Questa relazione è valida dal momento che dopo aver terminato le operazioni relative al *saxpy*, se si immagina un albero binario rovesciato per la collezione dei risultati (si veda l'illustrazione precedente), ad ogni passo vengono inviati da ogni processore attivo il doppio degli elementi inviati al passo precedente dai processori attivi per quel passo. Cioè i processori attivi man mano che ci si avvicina alla radice dell'albero, si dimezzano e si raddoppia la lunghezza dell'array inviato da ciascun processore attivo per quel passo. Inoltre, $c \in [2,3] \subset \mathbb{R}$.

Da questa relazione segue che lo speed-up è:

$$S_p(N) = \left(\frac{2M \cdot t_{calc}}{\left[2\left\lceil\frac{M}{p}\right\rceil + c \cdot \left(\sum_{i=1}^{log_2(p)} \left\lceil\frac{M}{p}\right\rceil^i\right)\right] \cdot t_{calc}}\right) < p$$

Infatti, il termine:

$$\left(\sum_{i=1}^{log_2(p)} \left\lceil\frac{M}{p}\right\rceil^i\right) > 0 \;\; se \;\; \{M > 0 \;\wedge\; p > 1\}$$

Segue che l'overhead è:

$$O_p(N) = \left(\left(p \cdot \left[2\left\lceil\frac{M}{p}\right\rceil + c \cdot \left(\sum_{i=1}^{log_2(p)} \left\lceil\frac{M}{p}\right\rceil^i\right)\right] \cdot t_{calc}\right) - (2M \cdot t_{calc})\right) = \left(c \cdot p \cdot \left(\sum_{i=1}^{log_2(p)} \left\lceil\frac{M}{p}\right\rceil^i\right) \cdot t_{calc}\right) \neq 0$$

E l'efficienza risulta essere:

$$E_p(N) = \left[\frac{\left(\frac{2M \cdot t_{calc}}{\left[2\left\lceil \frac{M}{p} \right\rceil + c \cdot \left(\sum_{i=1}^{log_2(p)} \left\lceil \frac{M}{p} \right\rceil^i \right) \right] \cdot t_{calc}} \right)}{p} \right] = \left[\left(\frac{1}{\left[c \cdot p \cdot \left(\sum_{i=1}^{log_2(p)} \left\lceil \frac{M}{p} \right\rceil^i \right) \right] \cdot t_{calc}} \right) \right] < 1$$

L'isoefficienza risultante in questo caso risulta essere:

$$I(N_0, p_0, p_1) = \left\{ \frac{\left(c \cdot p \cdot \left(\sum_{i=1}^{log_2(p_1)} \left\lceil \frac{M_1}{p_1} \right\rceil^i \right) \cdot t_{calc} \right)}{\left(c \cdot p \cdot \left(\sum_{i=1}^{log_2(p_0)} \left\lceil \frac{M_0}{p_0} \right\rceil^i \right) \cdot t_{calc} \right)} \cdot 2M_0 \right\} = \left\{ \frac{\left(\sum_{i=1}^{log_2(p_1)} \left\lceil \frac{M_1}{p_1} \right\rceil^i \right) \cdot t_{calc}}{\left(\sum_{i=1}^{log_2(p_0)} \left\lceil \frac{M_0}{p_0} \right\rceil^i \right) \cdot t_{calc}} \cdot 2M_0 \right\}$$

Infine, si consideri l'applicazione della legge di Ware-Amdhal. Siccome non si tiene in considerazione delle "spedizioni" e delle "ricezioni" dei dati, ovvero rispettivamente della distribuzione iniziale e del raccoglimento finale dei dati in unico vettore situato nella memoria a cui accede il processore *master*, e siccome le operazioni di puro calcolo che vengono effettuate da ogni processore sono $\left(2M/p\right)$, che considerando p processori diventano esattamente $\left[p \cdot \left(2M/p\right)\right]$ operazioni totali; allora è banale individuare la parte strettamente parallela e quella strettamente sequenziale dell'algoritmo. Infatti, date le condizioni appena menzionate, la fase parallela forma l'intero ammontare delle operazioni di puro calcolo considerate, mentre la parte sequenziale e a parallelismo medio è nulla. Di conseguenza è possibile sfruttare nuovamente la formulazione base della legge di Ware-Amdhal, ottenendo che:

$$(1-\alpha) = \left(\frac{p \cdot \left(\frac{2M}{p}\right)}{2M} \right) = \left(\frac{2M}{2M}\right) = 1$$

$$\alpha = 0$$

$$S_p(N) = \left(\frac{1}{\alpha + \left(\frac{1-\alpha}{p}\right)} \right) = \left(\frac{1}{\frac{1}{p}} \right) = p$$

Si può dimostrare che l'algoritmo parallelo descritto non risulta essere scalabile.

3. Descrizione dell'algoritmo parallelo

Di seguito si riportano i passi dell'algoritmo parallelo realizzato. Per prima cosa si inizializza l'ambiente parallelo MPI, poi ogni processore ricava il proprio identificativo e il numero di processori totali all'interno del *communicator* usato (nel nostro caso *MPI_COMM_WORLD*). A questo punto si effettua una verifica sugli argomenti passati da linea di comando al programma (successiva al controllo già effettuato in fase di esecuzione mediante script *employ.sh*) e si ricava l'identificativo del processore *master* che provvederà ad impostare gli aspetti di base dell'ambiente, cioè estrarrà dal file di configurazione (il cui path relativo è stato passato da linea di comando) il tipo di operazione *saxpy* da effettuare, il path relativo del file contenente i dati di input e il path relativo del file che conterrà i dati di output. A questo punto il processore *master* continua con l'inizializzazione dell'ambiente andando a leggere dal file dei dati di input la dimensione degli array a e b da costruire, alloca l'opportuna memoria in base alla dimensione letta per entrambi gli array, legge i valori da immettere in ognuno dei due array e infine legge lo scalare α. Il processore master segnalerà la presenza di un errore nel caso in cui il numero di elementi specificato da leggere per ogni array è inferiore o uguale a zero. Inoltre, si effettua una gestione profonda e precisa della memoria, dei file aperti e degli errori durante tutte le operazioni considerate finora e successivamente.

Una volta fatto ciò, visto il quantitativo di tempo che potrebbe essere richiesto per le operazioni finora descritte, si sfrutta una barriera di sincronizzazione per tutti i processori all'interno del communicator, dopo la quale si procede con l'invio in broadcast dal processore *master* a tutti gli altri processori della modalità dell'operazione *saxpy*, della dimensione dei singoli array a e b, dello scalare α. A questo punto si incontra un'altra barriera di sincronizzazione per tutti i processori.

Ora viene gestita la non esatta divisibilità delle dimensioni del problema per il numero di processori nel communicator. L'idea è quella di andare a valutare la parte intera del rapporto fra queste due quantità e assegnare un elemento in più ad ogni processore il cui identificativo risulta essere inferiore del resto risultante dalla divisione. Il processore *master* allocherà due vettori di *interi* dalla lunghezza pari al numero di processori presenti nel communicator: il primo servirà a tenere traccia delle dimensioni locali del problema per ogni processore e il secondo per tenere traccia degli spiazzamenti da considerare da qui a breve. Entrambi i vettori saranno infatti utilizzati sia per ripartire gli elementi del vettore a e b fra i processori, che per riunire gli elementi nel vettore c. Una volta divisi gli elementi dei due vettori fra i vari processori del communicator, si esegue la fase di calcolo locale sulle porzioni di array assegnate. A questo punto, si procede col riunire gli elementi dai vari vettori locali calcolati in un unico vettore risultante che è presente solo e unicamente nel processore *master*. Infine, si procede col rilascio della memoria non più utilizzata.

Ora è necessaria una barriera di sincronizzazione per calcolare i tempi di calcolo impiegati da ogni processore, si identifica il tempo massimo impiegato all'interno del cluster e lo si stampa insieme ad altre informazioni utili. Il processore *master* salva il vettore c all'interno del file di output il cui path relativo è stato ricavato dal file di configurazione, e si rilascia l'ultima parte di memoria allocata. Infine, l'ambiente parallelo viene terminato e l'esecuzione del software termina.

Si riepilogano in pseudo-codice i passi fondamentali appena descritti:

```
1   broadcast(saxpyChosenMode, arraySize, alpha)
2   arraySizeLoc = arraySize / nProcessor
3   remainder = arraySize % nProcessor
4
5    if (remainder > 0) {
6        if (processorID < remainder) arraySizeLoc += 1
7    }
8
9    costruzione array recvcounts e displacements
10   distribuzione di a e b fra i vari processori all'interno del communicator
11   operazione saxpy locale ai processori
12   unione dei singoli vettori in un singolo vettore presente nella memoria del processore master
13   calcolo del tempo massimo impiegato per effettuare le operazioni
14   salvataggio del risultato e rilascio risorse
```

4. Input e Output

Il software realizzato, il codice, i file di configurazione e i file contenenti i dati di input e di output sono organizzati nel seguente modo. La cartella principale è "saxpy_mpi", all'interno della quale è possibile trovare le seguenti cartelle:

- la cartella "src" contiene i codici sorgenti, le librerie, il makefile, il file di configurazione shell per il setup dell'ambiente Docker-MPI (setup.sh), il file di configurazione shell per avviare il software parallelo sui vari nodi del cluster di processori ed il "machinefile" per la configurazione dei nodi del cluster;
- la cartella "doc" contiene la documentazione esterna;
- la cartella "conf" contiene i file di configurazione. In ogni file di configurazione sono presenti le seguenti informazioni divise dal carattere *newline*: tipologia di operazione *saxpy* da eseguire (se in sequenziale (0) oppure in parallelo (1)), percorso relativo del file contenente i dati di input e percorso relativo del file contenente i dati di output;
- la cartella "data" contiene i files contenenti i dati di input e di output, in cui ogni singolo dato è separato dagli altri dal carattere *newline*. Si precisa che all'interno del file contenente i dati di input sono presenti le seguenti informazioni: lunghezza del singolo vettore $arraySize$ (così facendo verranno letti $(2 \cdot arraySize)$ elementi), $arraySize$ scalari rappresentanti gli elementi appartenenti al vettore a, $arraySize$ scalari rappresentanti gli elementi appartenenti al vettore b, e infine lo scalare α. Nel file contenente i dati di output invece, sono presenti i soli valori appartenenti al vettore c.

  Si precisa che gli elementi dei due vettori e lo scalare sono di tipo *float.*

A questo punto una volta preparati il file di configurazione (.conf), il file contenente i dati di input (.dat) e una volta compilato i sorgenti mediante il *makefile* fornito, è possibile avviare il software parallelo in ambiente MPI-Docker mediante lo script *employ.sh.* Sarà necessario fornire due parametri: il percorso relativo del file di configurazione scelto e l'identificativo del processore *master.*

In output verrà prodotto il file *outputData.dat* nel path specificato opportunamente nel file di configurazione usato. Inoltre, verranno mostrate a video le seguenti informazioni: la dimensione dei vettori utilizzata in input, lo scalare α utilizzato e il tempo massimo di esecuzione impiegato fra i vari tempi di esecuzione dei differenti processori appartenenti al cluster. Si precisa che non sono previste interazioni dell'utente durante la fase di calcolo, né durante l'intera esecuzione del software stesso. In questo modo, si evitano i tempi e i ritardi dovuti all'I/O.

Si segnala che in output potrebbero essere presenti dei *warning* in caso di dimensioni del problema molto elevate. Nonostante ciò, gli output prodotti risultano essere tutti corretti. Infine, si precisa che sono state gestite tutte (a meno di errore umano) le situazioni possibili di errore generate da tutte le routine e operazioni rispettivamente invocate ed effettuate. In caso di mancato inserimento di opportuni file di configurazione, di dati di input, di parametri da passare da linea di comando, di dimensioni dell'array negative o pari a zero, il software segnalerà l'errore e terminerà l'esecuzione.

5. Routine implementate

In questa sezione si illustrano le routine presenti all'interno del codice sorgente. Si precisa che per semplicità di lettura non vengono specificati tipo di dato restituito, parametri e tipo dei parametri. Per maggiori informazioni si consulti la documentazione interna. Si considerano prima le routine del C che sono state utilizzate.

- "strtol(...)" e la "strtoul(...)" per la conversione dei valori numerici letti dal file contenente i dati di input e dal file di configurazione, a seconda se il valore numerico da leggere è con o senza segno. La routine "strtof(...)" per leggere invece i valori numerici *float* dai file.
- "fprintf(...)" per stampare nei file di output e sullo *standard output*.
- "exit(...)" alla terminazione del *main*.
- "fopen(...)" e "fclose(...)" rispettivamente per aprire e chiudere i file.
- "getline(...)" per leggere un'intera riga all'interno di un file.
- "calloc(...)" e "free(...)" rispettivamente per un'allocazione con inizializzazione della memoria e per liberare la memoria allocata non più utilizzata.

Si considerano ora le routine di MPI che sono state utilizzate.

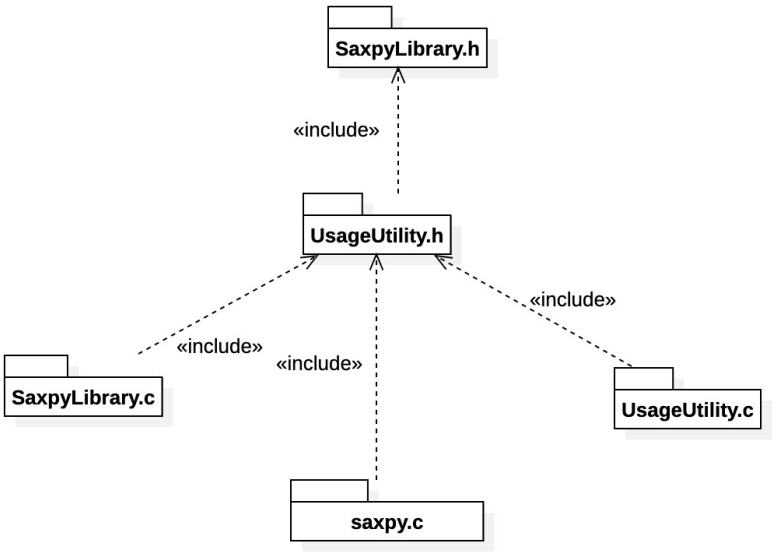
- "MPI_Init(...)" per l'inizializzazione dell'ambiente di esecuzione MPI.
- "MPI_Comm_rank(...)" per determinare l'identificativo dei singoli processori nel communicator.
- "MPI_Comm_size(...)" per determinare il numero di processori presenti nel communicator.
- "MPI_Barrier(...)" per creare una barriera di sincronizzazione fra tutti i processori del communicator.
- "MPI_Wtime(...)" per estrarre il tempo trascorso fino a quel momento.
- "MPI_Bcast(...)" per inviare dei dati dal processore *master* a tutti gli altri processori presenti nel communicator.
- "MPI_Reduce(...)" per calcolare chi fra i processori ha impiegato tempo massimo. Tecnicamente per eseguire l'operazione di *reduction* in modo semplice fra i valori di tempo presenti nei differenti processori del communicator.
- "MPI_Finalize(...)" per terminare l'ambiente di esecuzione MPI.
- "MPI_Abort(...)" per terminare l'ambiente di esecuzione MPI in caso di errori a tempo di esecuzione.
- "MPI_Gather(...)" per raccogliere dati della stessa dimensione distribuiti fra i processori del communicator in un unico dato (o array).
- "MPI_Gatherv(...)" per raccogliere dati di dimensione differente distribuiti fra i processori del communicator in un unico dato (o array).
- "MPI_Scatterv(...)" per distribuire dati di dimensione differente fra i processori del communicator.

Si considerano ora le routine implementate manualmente.

- "main(...)" si occupa dell'inizializzazione dell'ambiente di esecuzione MPI, del *broadcast* di alcuni dati fondamentali, dell'invocazione della routine che si occupa di effettuare l'operazione di *saxpy*, del prendere i tempi, del salvataggio dei risultati e della terminazione dell'ambiente di esecuzione MPI.
- "saxpy(...)" si occupa di invocare opportunamente l'operazione di *saxpy* al fine di svolgerla in maniera parallela oppure sequenziale.
- "saxpy_parallel(...)" implementa l'operazione di *saxpy* in parallelo in ambiente MIMD-DM con l'ausilio di MPI-Docker seguendo l'algoritmo descritto nella sezione tre di questo documento.
- "saxpy_sequential(...)" implementa l'operazione di *saxpy* in sequenziale. Si precisa che per invocare l'operazione *saxpy* in sequenziale, è necessario cambiare il codice utente all'interno della funzione "main(...)", in quanto ad ora è fortemente integrato per essere utilizzato in ambiente MPI.
- "checkUsage(...)" per la verifica degli argomenti passati da linea di comando all'atto dell'esecuzione.
- "raiseError(...)" per terminare l'ambiente di esecuzione MPI e mostrare l'errore avvenuto a *run-time*.
- "setEnvironment(...)" per leggere le impostazioni scelte dal file di configurazione, avviare la lettura dei dati dal file di dati e allocare la memoria necessaria.
- "createFloatArrayFromFile(...)" per creare un array di *float* a partire dai dati letti da un file.
- "printArray(...)" per stampare il vettore passato come parametro sul file puntato dal puntatore a file passato anch'esso come parametro.

- “saveResult(...)” per il salvataggio di un vettore in un file il cui path è specificato come parametro.
- “createFloatArray(...)” e “createIntArray(...)” per creare e restituire rispettivamente un array di *float* e di *interi* della dimensione specificata come parametro.
- “releaseMemory(...)” per il rilascio dei blocchi di memoria allocati passati come argomento.
- “closeFiles(...)” per la chiusura dei files passati come argomento mediante puntatori a file.

Le librerie incluse dai sorgenti sono le seguenti: stdio.h, stdlib.h, errno.h, string.h, time.h, math.h, stdarg.h, mpi.h. Gli standard di riferimento durante la scrittura dei sorgenti sono stati POSIX e ISO C99. Di seguito si allega uno schema al fine di esplicitare quali sorgenti includono quali librerie. Vengono omesse le librerie di sistema.



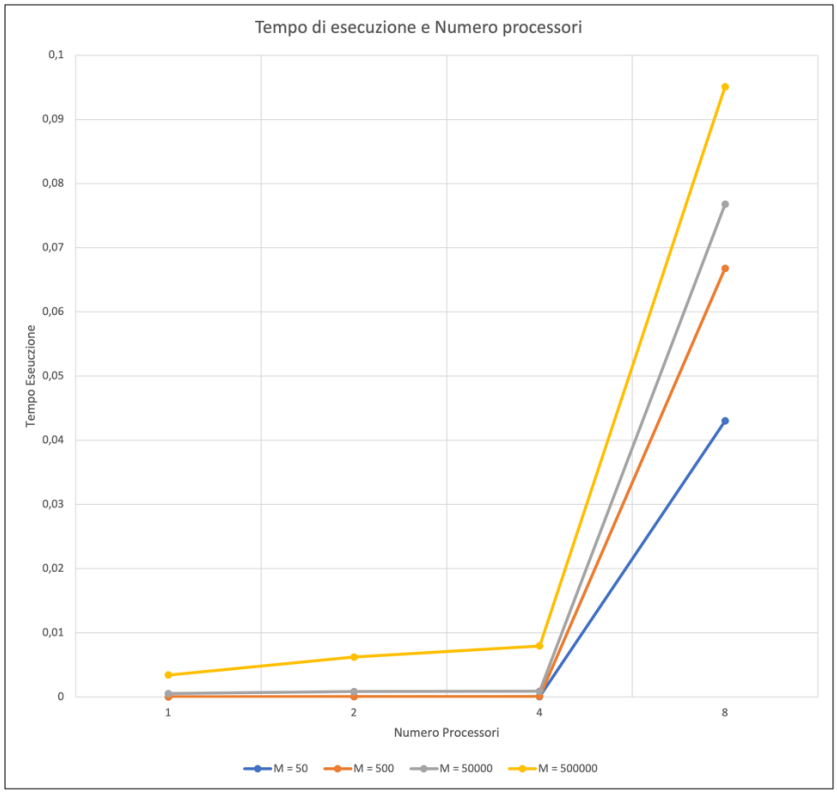
6. Analisi delle performance del software

Sono stati presi in considerazione dimensioni del problema M pari a 50, 500, 50000, 500000. Per ognuna di queste taglie del problema si è eseguita l'operazione *saxpy* per un numero di processori p pari a 1, 2, 4, 8. Si precisa che i risultati e i tempi che seguono sono stati ottenuti da ambiente MPI-Docker. Quindi comportamenti anomali sono dovuti all'ambiente stesso.

Facendo una rapida analisi, nonostante le condizioni “falsate” dalla simulazione dell'ambiente in MPI-Docker, il miglior numero di processori per la risoluzione di questo nucleo computazionale risulta essere due, ma in contesti reali tale valore è molto probabile che sia maggiore.

p	τ	M	S_τ	E_τ
1	0,0000037	50	1	1
1	0,0000054	500	1	1
1	0,0005172	50000	1	1
1	0,0034074	500000	1	1
2	0,0000253	50	0,14624506	0,07312253
2	0,000045	500	0,12	0,06
2	0,0008221	50000	0,62912054	0,31456027
2	0,0062081	500000	0,54886358	0,27443179
4	0,0000554	50	0,066787	0,01669675
4	0,0000747	500	0,07228916	0,01807229
4	0,0008893	50000	0,58158102	0,14539525
4	0,0079399	500000	0,42914898	0,10728725
8	0,0430066	50	8,6033E-05	1,0754E-05
8	0,066788	500	8,0853E-05	1,0107E-05
8	0,0767716	50000	0,00673687	0,00084211
8	0,0950831	500000	0,03583602	0,0044795

Di seguito è riportato il grafico che illustra come varia il tempo di esecuzione al variare del numero di processori utilizzati per dimensioni del problema differenti.

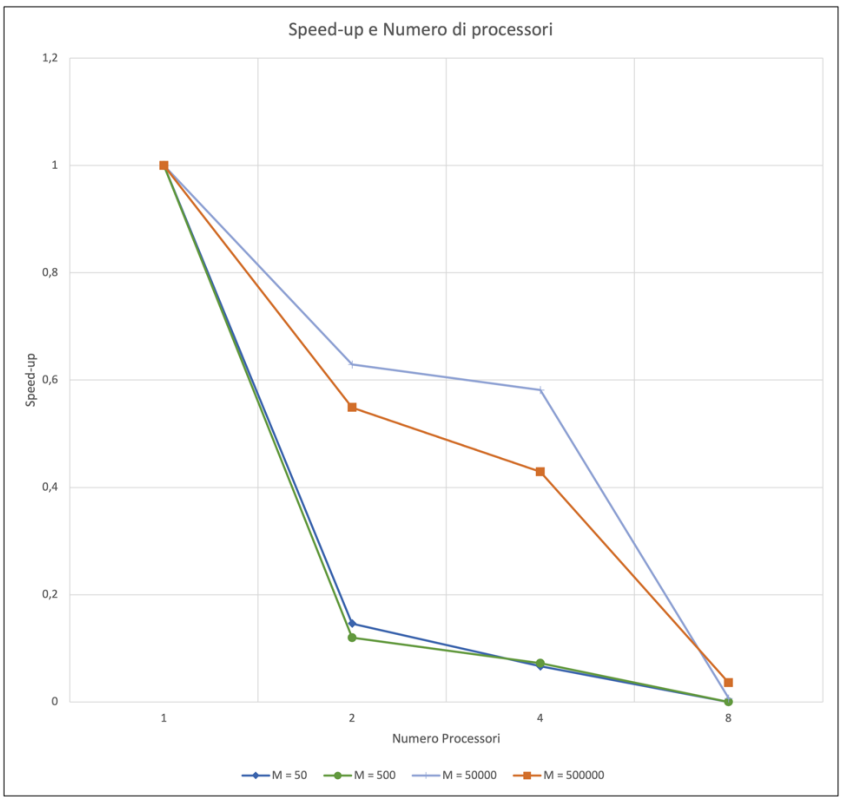


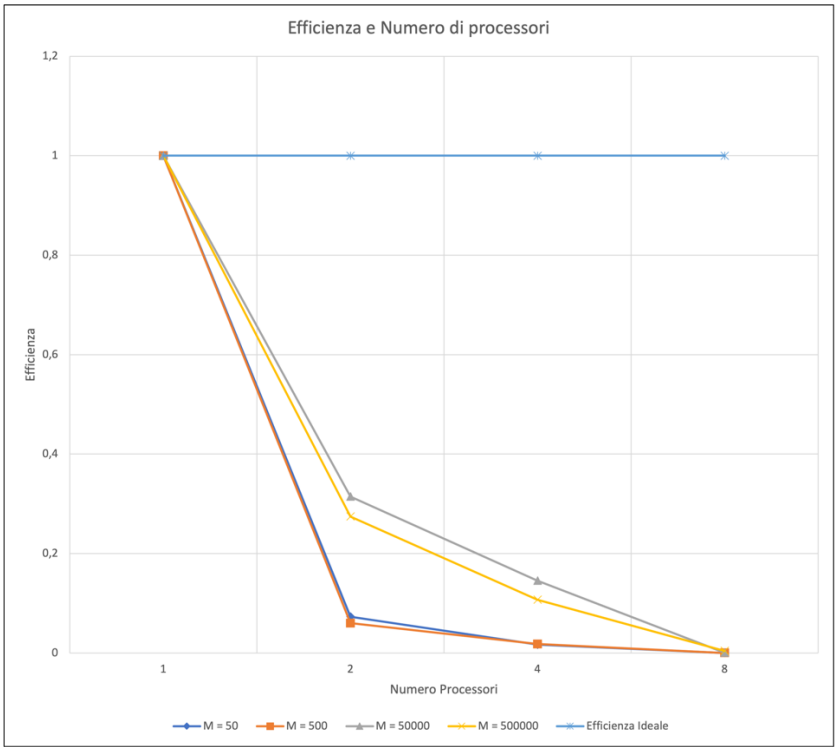
Il risultato anomalo che si può già evidenziare in questo caso è l'aumentare dei tempi di esecuzione all'aumentare del numero di processori. Di norma dovrebbe accadere che all'aumentare del numero delle unità processanti il tempo di esecuzione cali man mano. Non avendo accesso a un cluster di processori, non è possibile avere un'idea appropriata dei tempi di esecuzione effettivi dell'approccio parallelo MIMD-DM. Di conseguenza anche i successivi grafici che sono fondati sui valori dei tempi di esecuzione poc'anzi ricavati, risultano poco accurati per descrivere un caso reale.

La macchina "ospitante" sulla quale sono stati condotti i test, ricavati i tempi e i dati utili a formare i successivi grafici possiede un processore Intel Core i9-9980HK con 8 core, 16 threads per core, frequenza base a 2.4 Ghz, frequenza massima a 5.0 Ghz, cache CPU 16 MB del tipo Intel Smart Cache.

Di seguito sono riportati i grafici che illustrano rispettivamente come variano lo speed-up e l'efficienza al variare del numero di processori utilizzati per dimensioni del problema differenti. Si precisa che nel grafico dello speed-up non si riporta lo speed-up ideale in quanto disterebbe troppo dai dati già presenti, non consentendo un'opportuna visualizzazione degli stessi.

Ulteriori test e relativi risultati con M pari a 50, 500, 1000, 1500, 2000, 2500, 50000, 500000 è possibile visionarli nel file allegato "saxpy_mpi_time_2.xlsx".


Speed-up e Numero di processori
1,2
1
0,8
0,6
0,4
0,2
0
Speed-up
1
2
4
8
Numero Processori
M = 50
M = 500
M = 50000
M = 500000


Efficienza e Numero di processori
1,2
1
0,8
0,6
0,4
0,2
0
Efficienza
1
2
4
8
Numero Processori
M = 50
M = 500
M = 50000
M = 500000
Efficienza Ideale

7. Esempi d'uso

Si allegano di seguito cinque esempi in cui il numero di processori usati è pari a quattro. Nel primo esempio si mostra l'esecuzione di base del software con due array da 5 elementi, nel secondo esempio i due array contengono 50 elementi ciascuno, nel terzo esempio i due array contengono 250000 elementi ciascuno. Infine, nel quarto esempio si mostra cosa accade in caso di scelta di valori inferiori o uguali a zero per la dimensione dell'array. Mentre nel quinto esempio si mostra il funzionamento anche con una dimensione dei due vettori in input inferiore al numero dei processori presenti nel communicator del cluster usato.
Si noti come per dimensioni dei vettori molto elevate, come accade nel terzo esempio, l'ambiente Docker-MPI mostra alcuni *warnings* anomali che però non portano alla terminazione del programma. Si precisa che in tutti i casi e quindi anche in quest'ultimo, i dati presenti nei file di output generato risultano essere corretti.

```
cpd2021@b0ac8b26f8dd:~/saxpy_mpi/src$ sh employ.sh machinefile 4 ./saxpy ../conf/settings.conf 0
Starting MPI employing
saxpy                                                         100%   26KB  13.1MB/s   00:00
saxpy                                                         100%   26KB  24.9MB/s   00:00
saxpy                                                         100%   26KB  15.7MB/s   00:00



        *** Saxpy Operation completed successfully ***

Output can be found at: ../data/outputData.dat


        *** Saxpy Operation Details ***

-> Vectors Size:        5
-> Alpha:               2.00000
-> Max Time:            0.0000537 seconds
```

Esempio 1

```
cpd2021@b0ac8b26f8dd:~/saxpy_mpi/src$ sh employ.sh machinefile 4 ./saxpy ../conf/settingsMedium.co
nf 0
Starting MPI employing
saxpy                                                         100%   26KB  26.5MB/s   00:00
saxpy                                                         100%   26KB  32.1MB/s   00:00
saxpy                                                         100%   26KB  30.8MB/s   00:00



        *** Saxpy Operation completed successfully ***

Output can be found at: ../data/outputDataMedium.dat


        *** Saxpy Operation Details ***

-> Vectors Size:        50
-> Alpha:               2.00000
-> Max Time:            0.0000498 seconds
```

Esempio 2

```
cpd2021@b0ac8b26f8dd:~/saxpy_mpi/src$ sh employ.sh machinefile 4 ./saxpy ../conf/settingsExtreme.c
onf 0
Starting MPI employing
saxpy                                                         100%   26KB  21.2MB/s   00:00
saxpy                                                         100%   26KB  31.4MB/s   00:00
saxpy                                                         100%   26KB  36.3MB/s   00:00
[b0ac8b26f8dd:00573] Read -1, expected 250000, errno = 1
[b0ac8b26f8dd:00576] Read -1, expected 250000, errno = 1
[b0ac8b26f8dd:00577] Read -1, expected 250000, errno = 1
[b0ac8b26f8dd:00573] Read -1, expected 250000, errno = 1
[b0ac8b26f8dd:00576] Read -1, expected 250000, errno = 1
[b0ac8b26f8dd:00577] Read -1, expected 250000, errno = 1
[b0ac8b26f8dd:00572] Read -1, expected 250000, errno = 1
[b0ac8b26f8dd:00572] Read -1, expected 250000, errno = 1
[b0ac8b26f8dd:00572] Read -1, expected 250000, errno = 1



        *** Saxpy Operation completed successfully ***

Output can be found at: ../data/outputDataExtreme.dat


        *** Saxpy Operation Details ***

-> Vectors Size:        250000
-> Alpha:               3.50000
-> Max Time:            0.0048913 seconds
```

Esempio 3

```
cpd2021@b0ac8b26f8dd:~/saxpy_mpi/src$ sh employ.sh machinefile 4 ./saxpy ../conf/settings.conf 0
Starting MPI employing
saxpy                                                         100%   26KB  17.9MB/s   00:00
saxpy                                                         100%   26KB  29.6MB/s   00:00
saxpy                                                         100%   26KB  32.4MB/s   00:00
Scope: setEnvironment/reading array size: invalid array size. It should be greater than zero - Err
or #102
--------------------------------------------------------------------------
MPI_ABORT was invoked on rank 0 in communicator MPI_COMM_WORLD
with errorcode 102.

NOTE: invoking MPI_ABORT causes Open MPI to kill all MPI processes.
You may or may not see output from other processes, depending on
exactly when Open MPI kills them.
--------------------------------------------------------------------------
```

Esempio 4

```
cpd2021@b0ac8b26f8dd:~/saxpy_mpi/src$ sh employ.sh machinefile 4 ./saxpy ../conf/settings.conf 0
Starting MPI employing
saxpy                                                         100%   26KB  30.0MB/s   00:00
saxpy                                                         100%   26KB  32.5MB/s   00:00
saxpy                                                         100%   26KB  32.0MB/s   00:00



        *** Saxpy Operation completed successfully ***

Output can be found at: ../data/outputData.dat



        *** Saxpy Operation Details ***

-> Vectors Size:        2
-> Alpha:               -34.55000
-> Max Time:            0.0000372 seconds
```

Esempio 5

8. Bibliografia e sitografia

- "MPICH – Model MPI Implementation Reference Manual", William Groop, Ewing Lusk, Nathan Doss, Anthony Skjellum - January 13, 2003
- "MPICH User's Guide" – Pavan Balaji Sudheer Chunduri William Gropp Yanfei Guo Shintaro Iwasaki Travis Koehring Rob Latham Ken Raffenetti Min Si Rajeev Thakur Hui Zhou – 7 April, 2022
- "Richiami di Calcolo Parallelo", Livia Marcellino, Luigia Ambrosio
- Slide "Calcolo Parallelo e Distribuito e Laboratorio di Calcolo Parallelo e Distribuito", Livia Marcellino, Pasquale De Luca
- mpich.org/static/docs/
- stackoverflow.com, Jonathan Dursi
- mpi-forum.org/docs/mpi-1.1/mpi-11-html/node70.html#Node70
- valgrind.org
- github.com/delucap/Docker_MPI, Pasquale De Luca
- github.com/dennewbie/saxpy_mpi (repository attualmente privato)

9. Appendice

Si riporta di seguito il codice sorgente di "UsageUtility.h".

```
//
//  UsageUtility.h
//  saxpy_mpi
//
//  Created by Denny Caruso on 21/05/22.
//

#ifndef UsageUtility_h
#define UsageUtility_h

#include "SaxpyLibrary.h"



// Error code and scope message
static const int CHECK_USAGE_ERROR = 1;
static const char * CHECK_USAGE_SCOPE = "check usage";

static const int FPRINTF_ERROR = 2;
static const char * FPRINTF_SCOPE = "generic fprintf";

static const int FOPEN_ERROR = 3;
static const char * FOPEN_SCOPE = "generic fopen";

static const int GETLINE_ERROR = 4;
static const char * GETLINE_SCOPE = "getline: reading from file";

static const int CALLOC_ERROR = 5;
static const char * CALLOC_SCOPE = "generic calloc";

static const int CONFIGURATION_FILE_OPEN_ERROR = 6;
static const char * CONFIGURATION_FILE_OPEN_SCOPE = "setEnvironment/configurationFile opening";

static const int DATA_FILE_OPEN_ERROR = 7;
static const char * DATA_FILE_OPEN_SCOPE = "setEnvironment/dataFile opening";

static const int OUTPUT_FILE_OPEN_ERROR = 8;
static const char * OUTPUT_FILE_OPEN_SCOPE = "saveResult/outputFile opening";

static const int STRTOL_ERROR = 9;
static const char * STRTOL_SCOPE = "strtol";

static const int STRTOUL_ERROR = 10;
static const char * STRTOUL_SCOPE = "strtoul";

static const int STRTOF_ERROR = 11;
static const char * STRTOF_SCOPE = "strtof";
```

```
static const int INVALID_SAXPY_MODE_ERROR = 12;
static const char * INVALID_SAXPY_MODE_SCOPE = "saxpy invalid mode";

static const int RECURSION_OVERFLOW_ERROR = 13;
static const char * RECURSION_OVERFLOW_SCOPE = "raiseError recursion overflow";



typedef enum { FALSE, TRUE } boolean;



void checkUsage (int argc, const char ** argv, int expected_argc, const char * expectedUsageMessage,
                 MPI_Comm commWorld);

Void raiseError (const char * errorScope, int exitCode, MPI_Comm commWorld, boolean recursionOverflow);

Void setEnvironment (float ** a, float ** b, float * alpha, float ** c, unsigned int * arraySize,
                     const char * configurationFilePath, char ** outputFilePathString,
                     unsigned short int * saxpyMode, MPI_Comm commWorld);

void createFloatArrayFromFile (FILE * filePointer, float ** array, unsigned int arraySize, MPI_Comm
                               commWorld);

void printArray (FILE * filePointer, float * array, unsigned int arraySize, MPI_Comm commWorld);

void saveResult (float * array, unsigned int arraySize, const char * outputFilePath, MPI_Comm commWorld);

float * createFloatArray (unsigned int arraySize, MPI_Comm commWorld);

int * createIntArray (unsigned int arraySize, MPI_Comm commWorld);

void releaseMemory (void * arg1, ... );

void closeFiles (void * arg1, ... );

#endif /* UsageUtility_h */
```

Si riporta di seguito il codice sorgente di “UsageUtility.c”.

```c
//
//  UsageUtility.c
//  saxpy_mpi
//
//  Created by Denny Caruso on 21/05/22.
//

#include "UsageUtility.h"



/*
    Controlla i parametri passati via linea di comando.
    PARMS:
    - int argc: numero di parametri passati via linea di comando.
    - const char ** argv: elenco parametri passati via linea di comando.
    - int expected_argc: numero di parametri attesi da linea di comando.
    - const char * expectedUsageMessage: messaggio di informazioni per l'utente su quali parametri passare via
linea di comando.
    - MPI_Comm commWorld: communicator utilizzato.
*/
void checkUsage (int argc, const char ** argv, int expected_argc, const char * expectedUsageMessage, MPI_Comm
commWorld) {
    if (argc != expected_argc) {
        if (fprintf(stderr, (const char * restrict) "Usage: %s %s\n", argv[0], expectedUsageMessage) < 0)
         { raiseError(FPRINTF_SCOPE, FPRINTF_ERROR, commWorld, FALSE); }
        raiseError(CHECK_USAGE_SCOPE, CHECK_USAGE_ERROR, commWorld, FALSE);
    }
}

/*
    Gestisce gli errori con conseguente terminazione del programma su tutti processori del communicator.
    Stampa del messaggio e codice d'errore.
    PARMS:
        - const char * errorScope: scope dove è avvenuto l'errore o messaggio di errore da mostrare all'utente.
        - int exitCode: codice di errore da mostrare all'utente.
        - MPI_Comm commWorld: communicator utilizzato.
        - boolean recursionOverflow: se vero indica un errore implicito alla "MPI_Abort(...)", tale per cui si
termina in maniera locale.
*/
void raiseError (const char * errorScope, int exitCode, MPI_Comm commWorld, boolean recursionOverflow) {
    if (recursionOverflow) {
        if (fprintf(stderr, (const char * restrict) "Scope: %s - Error #%d\n", RECURSION_OVERFLOW_SCOPE,
RECURSION_OVERFLOW_ERROR) < 0) raiseError(FPRINTF_SCOPE, FPRINTF_ERROR, commWorld, TRUE);
        exit(RECURSION_OVERFLOW_ERROR);
    } else {
        if (fprintf(stderr, (const char * restrict) "Scope: %s - Error #%d\n", errorScope, exitCode) < 0)
raiseError(FPRINTF_SCOPE, FPRINTF_ERROR, commWorld, FALSE);
```

```
        if (MPI_Abort(commWorld, exitCode) != MPI_SUCCESS) raiseError(MPI_ABORT_SCOPE, MPI_ABORT_ERROR, commWorld,
TRUE);
    }
}

/*
    Imposta l'ambiente prima di procedere con l'operazione saxpy vera e propria. Si occupa di allocare la memoria
necessaria, di leggere le impostazioni dal file di configurazione, di leggere i dati di input e le dimensioni degli
array. Il tutto con un opportuno error handling.
    PARMS:
        - float ** a: array a di input che verrà prima allocato e poi riempito con i valori letti dal file di
input.
        - float ** b: array b di input che verrà prima allocato e poi riempito con i valori letti dal file di
input.
        - float * alpha: scalare alpha coinvolto nell'operazione saxpy.
        - float ** c: array c di output che verrà soltanto allocato all'interno di questa routine.
        - unsigned int * arraySize: dimensione array a, b, c.
        - const char * configurationFilePath: path dove è situato il file di configurazione.
        - char ** outputFilePathString: path dove è situato il file di output all'interno del quale memorizzare il
risultato
        dell'operazione saxpy.
        - unsigned short int * saxpyMode: modalità operazione saxpy.
        - MPI_Comm commWorld: communicator utilizzato.
*/
void setEnvironment (float ** a, float ** b, float * alpha, float ** c, unsigned int * arraySize, const char *
configurationFilePath, char ** outputFilePathString, unsigned short int * saxpyMode, MPI_Comm commWorld) {
    FILE * configurationFilePointer, * dataFilePointer;
    ssize_t getLineBytes;
    size_t dataFilePathLength = 0, nLength = 0, singleNumberLength = 0, outputFilePathLength = 0, saxpyModeLength =
0;
    char * dataFilePathString = NULL, * nString = NULL, * singleNumberString = NULL, * saxpyModeString = NULL;
    float singleNumber = 0.0F;

    // read basic settings parameter from .config file
    configurationFilePointer = fopen(configurationFilePath, "r");
    if (!configurationFilePointer) raiseError(CONFIGURATION_FILE_OPEN_SCOPE, CONFIGURATION_FILE_OPEN_ERROR,
commWorld, FALSE);
    // read saxpy approach
    if ((getLineBytes = getline((char ** restrict) & saxpyModeString, (size_t * restrict) & saxpyModeLength, (FILE
* restrict) configurationFilePointer)) == -1) raiseError(GETLINE_SCOPE, GETLINE_ERROR, commWorld, FALSE);
    * saxpyMode = (unsigned short int) strtoul((const char * restrict) saxpyModeString, (char ** restrict) NULL,
10);
    if (* saxpyMode == 0 && (errno == EINVAL || errno == ERANGE)) raiseError(STRTOUL_SCOPE, STRTOUL_ERROR,
commWorld, FALSE);
    // read input data file path
    if ((getLineBytes = getline((char ** restrict) & dataFilePathString, (size_t * restrict) & dataFilePathLength,
(FILE * restrict) configurationFilePointer)) == -1) raiseError(GETLINE_SCOPE, GETLINE_ERROR, commWorld, FALSE);
    dataFilePathString[strlen(dataFilePathString) - 1] = '\0';
    // read output data file path
```

```
    if ((getLineBytes = getline((char ** restrict) outputFilePathString, (size_t * restrict) &
outputFilePathLength, (FILE * restrict) configurationFilePointer)) == -1) raiseError(GETLINE_SCOPE, GETLINE_ERROR,
commWorld, FALSE);

    // read amount of numbers to read for each array
    dataFilePointer = fopen(dataFilePathString, "r");
    if (!dataFilePointer) raiseError(DATA_FILE_OPEN_SCOPE, DATA_FILE_OPEN_ERROR, commWorld, FALSE);
    if ((getLineBytes = getline((char ** restrict) & nString, (size_t * restrict) & nLength, (FILE * restrict)
dataFilePointer)) == -1) raiseError(GETLINE_SCOPE, GETLINE_ERROR, commWorld, FALSE);
    * arraySize = (unsigned int) strtoul((const char * restrict) nString, (char ** restrict) NULL, 10);
    if (* arraySize == 0 && (errno == EINVAL || errno == ERANGE)) raiseError(STRTOUL_SCOPE, STRTOUL_ERROR,
commWorld, FALSE);
    if (* arraySize <= 0) raiseError(INVALID_ARRAY_SIZE_SCOPE, INVALID_ARRAY_SIZE_ERROR, commWorld, FALSE);

    // read array a, b and scalar alpha from file. Create array c
    createFloatArrayFromFile(dataFilePointer, a, * arraySize, commWorld);
    createFloatArrayFromFile(dataFilePointer, b, * arraySize, commWorld);
    if ((getLineBytes = getline((char ** restrict) & singleNumberString, (size_t * restrict) & singleNumberLength,
(FILE * restrict) dataFilePointer)) == -1) raiseError(GETLINE_SCOPE, GETLINE_ERROR, commWorld, FALSE);
    * alpha = (float) strtof((const char *) singleNumberString, (char ** restrict) NULL);
    if ((* alpha == 0.0F || * alpha == HUGE_VALF) && (errno == ERANGE)) raiseError(STRTOF_SCOPE, STRTOF_ERROR,
commWorld, FALSE);

    * c = createFloatArray(* arraySize, commWorld);

    closeFiles(configurationFilePointer, dataFilePointer, (void *) 0);
    releaseMemory(dataFilePathString, nString, singleNumberString, saxpyModeString, (void *) 0);
}

/*
    Legge "arraySize" float dal file pointer specificato da "filePointer" e li memorizza nell'array "array" che
essendo parametro di output viene "cambiato" anche per la routine che invoca.
    PARMS:
    - unsigned int arraySize: dimensione array da creare.
    - MPI_Comm commWorld:  communicator utilizzato di riferimento.
*/
void createFloatArrayFromFile (FILE * filePointer, float ** array, unsigned int arraySize, MPI_Comm commWorld) {
    char * singleNumberString = NULL;
    size_t singleNumberLength = 0;
    ssize_t getLineBytes;
    float singleNumber = 0.0F;

    * array = createFloatArray(arraySize, commWorld);
    for (int i = 0; i < arraySize; i++) {
        /*
            Lettura di un float per volta e conseguente memorizzazione nell'array.
        */
        if ((getLineBytes = getline((char ** restrict) & singleNumberString, (size_t * restrict) &
singleNumberLength, (FILE * restrict) filePointer)) == -1) raiseError(GETLINE_SCOPE, GETLINE_ERROR, commWorld,
FALSE);
        singleNumber = (float) strtof((const char *) singleNumberString, (char ** restrict) NULL);
```

```
        if ((singleNumber == 0.0F || singleNumber == HUGE_VALF) && (errno == ERANGE)) raiseError(STRTOF_SCOPE,
STRTOF_ERROR, commWorld, FALSE);
        *((* array) + i) = singleNumber;
    }

    releaseMemory(singleNumberString, (void *) 0);
}

/*
    Scrive un array di float di dimensione "arraySize" sul file pointer specificato da "filePointer".
    PARMS:
    - FILE * filePointer: puntatore al file dove scrivere i valori float.
    - float * array: array contenente i dati da scrivere nel file.
    - unsigned int arraySize: dimensione array.
    - MPI_Comm commWorld: communicator utilizzato di riferimento.
*/
void printArray (FILE * filePointer, float * array, unsigned int arraySize, MPI_Comm commWorld) {
    for (int i = 0; i < arraySize; i++) if (fprintf(filePointer, "%.5f\n", array[i]) < 0) raiseError(FPRINTF_SCOPE,
FPRINTF_ERROR, commWorld, FALSE);
}

/*
    Salva un array di float di dimensione "arraySize" all'interno del file con path "outputFilePath".
    PARMS:
    - float * array: array contenente i dati da scrivere nel file.
    - unsigned int arraySize: dimensione array.
    - const char * outputFilePath: path del file di output.
    - MPI_Comm commWorld: communicator utilizzato di riferimento.
*/
void saveResult (float * array, unsigned int arraySize, const char * outputFilePath, MPI_Comm commWorld) {
    FILE * outputFilePointer = fopen(outputFilePath, "w");
    if (!outputFilePointer) raiseError(DATA_FILE_OPEN_SCOPE, DATA_FILE_OPEN_ERROR, commWorld, FALSE);
    printArray(outputFilePointer, array, arraySize, commWorld);
    closeFiles(outputFilePointer, (void *) 0);
}

/*
    Restituisce un array di float di dimensione "arraySize".
    PARMS:
    - unsigned int arraySize: dimensione array da creare.
    - MPI_Comm commWorld: communicator utilizzato di riferimento.
*/
float * createFloatArray (unsigned int arraySize, MPI_Comm commWorld) {
    float * array = (float *) calloc(arraySize, sizeof(* array));
    if (!array) raiseError(CALLOC_SCOPE, CALLOC_ERROR, commWorld, FALSE);
    return array;
}

/*
    Restituisce un array di interi di dimensione "arraySize".
    PARMS:
```

```
    - unsigned int arraySize: dimensione array da creare.
    - MPI_Comm commWorld: communicator utilizzato di riferimento.
*/
int * createIntArray (unsigned int arraySize, MPI_Comm commWorld) {
    int * array = (int *) calloc(arraySize, sizeof(* array));
    if (!array) raiseError(CALLOC_SCOPE, CALLOC_ERROR, commWorld, FALSE);
    return array;
}

/*
    Libera uno o più blocchi di memoria passati nei parametri come puntatori a void. Si tratta di una "variadic
function".
    PARMS:
    - void * arg1: primo elemento della "va_list" creata successivamente e a partire dal quale si procederà con
l'operazione specificata nel while { ... }.
*/
void releaseMemory (void * arg1, ... ) {
    va_list argumentsList;
    void * currentElement;
    free(arg1);
    va_start(argumentsList, arg1);
    while ((arg1 = va_arg(argumentsList, void *)) != 0) free(arg1);
    va_end(argumentsList);
}

/*
    Chiude uno o più file passati nei parametri come puntatori a void. Si tratta di una "variadic function".
    PARMS:
    - void * arg1: primo elemento della "va_list" creata successivamente e a partire dal quale si procederà con
l'operazione specificata nel while { ... }.
*/
void closeFiles (void * arg1, ... ) {
    va_list argumentsList;
    void * currentElement;
    fclose(arg1);
    va_start(argumentsList, arg1);
    while ((arg1 = va_arg(argumentsList, void *)) != 0) fclose(arg1);
    va_end(argumentsList);
}
```

Si riporta di seguito il codice sorgente di “SaxpyLibrary.h”.

```
//
//  SaxpyLibrary.h
//  saxpy_mpi
//
//  Created by Denny Caruso on 28/05/22.
//

#include "mpi.h"
#include <stdio.h>
// fprintf, perror
#include <stdlib.h>
// exit
#include <errno.h>
#include <string.h>
// strlen
#include <time.h>
// time
#include <math.h>
// HUGE_VALF
#include <stdarg.h>
// for variadic procedures



// Error code and scope message
static const char * MPI_INIT_SCOPE = "MPI Init";
static const char * MPI_FINALIZE_SCOPE = "MPI Finalize";
static const char * MPI_COMM_RANK_SCOPE = "MPI CommRank";
static const char * MPI_COMM_SIZE_SCOPE = "MPI CommSize";
static const char * MPI_SEND_SCOPE = "MPI Send";
static const char * MPI_RECV_SCOPE = "MPI Recv";
static const char * MPI_GATHER_SCOPE = "MPI Gather";
static const char * MPI_GATHERV_SCOPE = "MPI Gatherv";
static const char * MPI_BARRIER_SCOPE = "MPI Barrier";
static const char * MPI_BCAST_SCOPE = "MPI Bcast";
static const char * MPI_SCATTER_SCOPE = "MPI Scatter";
static const char * MPI_SCATTERV_SCOPE = "MPI Scatterv";

static const int MPI_ABORT_ERROR = 101;
static const char * MPI_ABORT_SCOPE = "MPI Abort";

static const int INVALID_ARRAY_SIZE_ERROR = 102;
static const char * INVALID_ARRAY_SIZE_SCOPE = "setEnvironment/reading array size: invalid array size. It should be
greater than zero";
```

```
static const char * OUTPUT_USER_MESSAGE = "\n\n\n\t*** Saxpy Operation completed successfully ***\n\nOutput can be
found at: %s\n\n\n\t*** Saxpy Operation Details ***\n\n-> Vectors Size:\t%d\n-> Alpha:\t\t%.5f\n-> Max
Time:\t\t%.7f seconds\n\n\n";



typedef enum {
    SAXPY_SEQUENTIAL = 0,
    SAXPY_PARALLEL = 1
} SAXPY_MODE;



void saxpy (float * a, float * b, float ** c, float alpha, unsigned int arraySize, unsigned short int
            saxpyMode, int masterProcessorID, MPI_Comm commWorld, int processorID, unsigned int
            nProcessor);

void saxpy_parallel (float * a, float * b, float ** c, float alpha, unsigned int arraySize, int
                     masterProcessorID, MPI_Comm commWorld, int processorID, unsigned int nProcessor);

void saxpy_sequential (float * a, float * b, float ** c, float alpha, unsigned int arraySize, MPI_Comm
                       commWorld);
```

Si riporta di seguito il codice sorgente di “SaxpyLibrary.c”.

```c
//
//  SaxpyLibrary.c
//  saxpy_mpi
//
//  Created by Denny Caruso on 28/05/22.
//

#include "UsageUtility.h"



/*
    Invoca la saxpy in parallelo oppure in sequenziale a seconda del valore assunto dalla variabile "saxpyMode".
    PARMS:
        - float * a: array a letto dal file di input.
        - float * b: array b letto dal file di input.
        - float ** c: puntatore ad array c in cui verrà conservato il vettore risultante finale.
        - float alpha: scalare dell'operazione saxpy.
        - unsigned int arraySize: dimensione di a, e quindi anche dimensione di b.
        - unsigned short int saxpyMode: modalità operazione saxpy, parallela o sequenziale.
        - int masterProcessorID: identificativo del processore master del communicator.
        - MPI_Comm commWorld: communicator usato di riferimento.
        - int processorID: identificativo del processore in esecuzione.
        - unsigned int nProcessor: numero di processori presenti all'interno del communicator.

*/
void saxpy (float * a, float * b, float ** c, float alpha, unsigned int arraySize, unsigned short int saxpyMode,
int masterProcessorID, MPI_Comm commWorld, int processorID, unsigned int nProcessor) {
    switch (saxpyMode) {
        case SAXPY_SEQUENTIAL:
            saxpy_sequential(a, b, c, alpha, arraySize, commWorld);
            break;
        case SAXPY_PARALLEL:
            saxpy_parallel(a, b, c, alpha, arraySize, masterProcessorID, commWorld, processorID, nProcessor);
            break;
        default:
            raiseError(INVALID_SAXPY_MODE_SCOPE, INVALID_SAXPY_MODE_ERROR, commWorld, FALSE);
            break;
    }
}

/*
    Realizza l'operazione saxpy in parallelo: c = alpa * a + b.
    PARMS:
        - float * a: array a letto dal file di input.
        - float * b: array b letto dal file di input.
        - float ** c: puntatore ad array c in cui verrà conservato il vettore risultante finale.
        - float alpha: scalare dell'operazione saxpy.
        - unsigned int arraySize: dimensione di a, e quindi anche dimensione di b.
```

```
        - int masterProcessorID: identificativo del processore master del communicator.
        - MPI_Comm commWorld: communicator usato di riferimento.
        - int processorID: identificativo del processore in esecuzione.
        - unsigned int nProcessor: numero di processori presenti all'interno del communicator.
*/
void saxpy_parallel (float * a, float * b, float ** c, float alpha, unsigned int arraySize, int masterProcessorID,
MPI_Comm commWorld, int processorID, unsigned int nProcessor) {
    int errorCode;
    int * recvcounts = NULL, * displacements = NULL, arraySizeLoc = 0;
    unsigned int remainder = 0;
    float * aLoc, * bLoc, * cLoc;
    aLoc = bLoc = cLoc = NULL;

    /*
        La variabile "errorCode" serve per mostrare gli eventuali codici di errore generati dalla chiamata delle
routine della libreria di MPI. L'array "recvounts" servirà a tenere traccia delle dimensioni locali del problema
per ogni processore mentre "displacements" per tenere traccia degli spiazzamenti da considerare da qui a breve.
Entrambi i vettori saranno infatti utilizzati sia per ripartire gli elementi del vettore a e b, che per riunire gli
elementi nel vettore c.

        Di seguito è riportato un esempio con M = 23 e p = 5:
        i    count[i]   count[i]+1   displ[i]   displ[i]-displ[i-1]
        ------------------------------------------------------------
        0       5          6           0
        1       5          6           5               5
        2       5          6          10               5
        3       4          5          15               5
        4       4          5          19               4

        La variabile "arraySizeLoc" è la dimensione della porzione di array a e b assegnata al processore. La
variabile "remainder" è il resto ottenuto dalla divisione fra la dimensione del problema ed il numero di
processori. Poi si invia in broadcast dal processore master a tutti gli altri nel communicator lo scalare alpha
coinvolto nell'operazione saxpy.
    */
    if ((errorCode = MPI_Bcast(& alpha, 1, MPI_FLOAT, masterProcessorID, commWorld)) != MPI_SUCCESS)
raiseError(MPI_BCAST_SCOPE, errorCode, commWorld, FALSE);
    if ((errorCode = MPI_Barrier(commWorld) != MPI_SUCCESS)) raiseError(MPI_BARRIER_SCOPE, errorCode, commWorld,
FALSE);

    /*
        Ripartizione del numero di elementi totali fra i processori. Ogni processore con identificativo minore del
resto, ottiene un elemento in più da entrambi gli array da considerare per l'operazione saxpy. Successivamente,
inizializzo i vettori "recvcounts" e "displacements".
    */
    arraySizeLoc = arraySize / nProcessor;
    remainder = arraySize % nProcessor;
    if (remainder > 0) {
        if (processorID < remainder) {
            arraySizeLoc += 1;
        }
    }
```

```
    if (processorID == masterProcessorID) {
        recvcounts = createIntArray(nProcessor, commWorld);
        displacements = createIntArray(nProcessor, commWorld);
        displacements[0] = 0;
    }

    /*
        Si ricava l'array "recvcounts" tramite "MPI_Gather(...)" con le varie dimensioni locali "arratSizeLoc". Si ricava poi l'array degli spiazzamenti "displacements" da considerare. Poi si alloca la memoria per gli array locali "aLoc", "bLoc" e "cLoc".
    */
    if ((errorCode = MPI_Gather(& arraySizeLoc, 1, MPI_INT, recvcounts, 1, MPI_INT, masterProcessorID, commWorld)) != MPI_SUCCESS) raiseError(MPI_GATHER_SCOPE, errorCode, commWorld, FALSE);
    if (processorID == masterProcessorID) for (int i = 1; i < nProcessor; i++) displacements[i] = displacements[i - 1] + recvcounts[i - 1];

    aLoc = createFloatArray(arraySizeLoc, commWorld);
    bLoc = createFloatArray(arraySizeLoc, commWorld);
    cLoc = createFloatArray(arraySizeLoc, commWorld);

    /*
        Si effettua la suddivisione degli elementi di a e di b in base a dimensioni locali potenzialmente differenti senza difficoltà grazie agli array "recvcounts" e "displacements". Si effettua l'operazione saxpy in locale su tali vettori, che permette di ottenere il risultato locale in "cLoc". A questo punto si effettua la "ricostruzione" del vettore finale "c" grazie ai due array precedentemente costruiti "recvcounts" e "displacements". Si precisa che non si usa la "MPI_Scatter(...)" e la "MPI_Gather(...)", ma la  "MPI_Scatterv(...)" e la "MPI_Gatherv(...)", in quanto è necessario prevedere anche una suddivisibilità non esatta del numero della dimensione del problema per il numero di processori presenti nel communicator. Infine, si rilascia la memoria allocata e non più utilizzata.
    */
    if ((errorCode = MPI_Scatterv(a, recvcounts, displacements, MPI_FLOAT, aLoc, arraySizeLoc, MPI_FLOAT, masterProcessorID, commWorld)) != MPI_SUCCESS) raiseError(MPI_SCATTERV_SCOPE, errorCode, commWorld, FALSE);
    if ((errorCode = MPI_Scatterv(b, recvcounts, displacements, MPI_FLOAT, bLoc, arraySizeLoc, MPI_FLOAT, masterProcessorID, commWorld)) != MPI_SUCCESS) raiseError(MPI_SCATTERV_SCOPE, errorCode, commWorld, FALSE);
    saxpy_sequential(aLoc, bLoc, & cLoc, alpha, arraySizeLoc, commWorld);
    if ((errorCode = MPI_Gatherv(cLoc, arraySizeLoc, MPI_FLOAT, * c, recvcounts, displacements, MPI_FLOAT, masterProcessorID, commWorld)) != MPI_SUCCESS) raiseError(MPI_GATHERV_SCOPE, errorCode, commWorld, FALSE);

    releaseMemory(aLoc, bLoc, cLoc, (void *) 0);
    if (processorID == masterProcessorID) releaseMemory(recvcounts, displacements, (void *) 0);
}

/*
    Realizza l'operazione saxpy in sequenziale: c = alpa * a + b.
    PARMS:
        - float * a: array a letto dal file di input.
        - float * b: array b letto dal file di input.
        - float ** c: puntatore ad array c in cui verrà conservato il vettore risultante finale.
        - float alpha: scalare dell'operazione saxpy.
        - unsigned int arraySize: dimensione di a, e quindi anche dimensione di b.
```

```
        - MPI_Comm commWorld: communicator usato di riferimento.
*/
void saxpy_sequential (float * a, float * b, float ** c, float alpha, unsigned int arraySize, MPI_Comm commWorld) {
    for (int i = 0; i < arraySize; i++) * ((* c) + i) = (alpha * (* (a + i))) + (* (b + i));
}
```

Si riporta di seguito il codice sorgente di “saxpy.c”.

```
//
//  saxpy.c
//  saxpy_mpi
//
//  Created by Denny Caruso on 21/05/22.
//

/*
    Somma tra un vettore "a" moltiplicato per uno scalare "alpha" e un altro vettore "b"
    in parallelo in ambiente MPI-Docker (c = alpha * a + b).
*/

#include "UsageUtility.h"



int main (int argc, char ** argv) {
    /*
        Il numero di parametri atteso è 3: nome eseguibile, path relativo del file di configurazione e ID del
processore master. La variabile "outputFilePath" è il path relativo del file di output dove salvare il risultato
calcolato. La variabile "errorCode" serve per mostrare gli eventuali codici di errore generati dalla chiamata delle
routine della libreria di MPI. La variabile "totalTime" contiene il tempo totale per ogni processore impiegato per
effettuare l'operazione di calcolo e le altre operazioni annesse. Mentre la variabile "maxTime" conterrà solo il
tempo massimo fra tutti i "tempi totali impiegati" dai processori.
    */
    const int expectedArgc = 3;
    const char * expectedUsageMessage = "<configuration filepath> <master processor ID>";
    char * outputFilePath = NULL;

    int masterProcessorID, processorID, errorCode;
    unsigned int arraySize = 0,  nProcessor = 0;
    unsigned short int saxpyChosenMode;

    float * a, * b, * c, alpha;
    double startTime, endTime, totalTime, maxTime;
    a = b = c = NULL;
    MPI_Comm myCommWorld = MPI_COMM_WORLD;

    /*
        Inizializzazione dell'ambiente MPI, si ricava poi l'identificativo del processore all'interno del
communicator e il numero di processori presenti in esso. Si verificano i parametri passati da linea di comando e
tra questi si converte il "masterProcessorID" in formato numerico intero, con verifica degli errori annessa.
Infine, si imposta l'ambiente di calcolo con "setEnvironment(...)": allocazione memoria, lettura impostazioni,
dati, etc.
    */
    if ((errorCode = MPI_Init(& argc, & argv)) != MPI_SUCCESS) raiseError(MPI_INIT_SCOPE, errorCode, myCommWorld,
FALSE);
    if ((errorCode = MPI_Comm_rank(myCommWorld, & processorID)) != MPI_SUCCESS) raiseError(MPI_COMM_RANK_SCOPE,
errorCode, myCommWorld, FALSE);
```

```
    if ((errorCode = MPI_Comm_size(myCommWorld, & nProcessor)) != MPI_SUCCESS) raiseError(MPI_COMM_SIZE_SCOPE,
errorCode, myCommWorld, FALSE);

    checkUsage(argc, (const char **) argv, expectedArgc, expectedUsageMessage, myCommWorld);
    masterProcessorID = (int) strtol((const char * restrict) argv[2], (char ** restrict) NULL, 10);
    if (masterProcessorID == 0 && (errno == EINVAL || errno == ERANGE)) raiseError(STRTOL_SCOPE, STRTOL_ERROR,
myCommWorld, FALSE);
    if (processorID == masterProcessorID) setEnvironment(& a, & b, & alpha, & c, & arraySize, argv[1], &
outputFilePath, & saxpyChosenMode, myCommWorld);

    /*
        Si stabilisce una barriera di sincronizzazione dopo aver impostato l'ambiente e si prende il tempo di
inizio. Poi si invia in broadcast dal processore master a tutti gli altri nel communicator la modalità di
operazione saxpy scelta e la dimensione dei due array ricavati dal file di dati in input. Fatto ciò, si invoca
l'operazione di "saxpy(...)".
    */
    if ((errorCode = MPI_Barrier(myCommWorld) != MPI_SUCCESS)) raiseError(MPI_BARRIER_SCOPE, errorCode,
myCommWorld, FALSE);
    startTime = MPI_Wtime();
    if ((errorCode = MPI_Bcast(& saxpyChosenMode, 1, MPI_UNSIGNED_SHORT, masterProcessorID, myCommWorld)) !=
MPI_SUCCESS) raiseError(MPI_BCAST_SCOPE, errorCode, myCommWorld, FALSE);
    if ((errorCode = MPI_Bcast(& arraySize, 1, MPI_UNSIGNED, masterProcessorID, myCommWorld)) != MPI_SUCCESS)
raiseError(MPI_BCAST_SCOPE, errorCode, myCommWorld, FALSE);

    saxpy(a, b, & c, alpha, arraySize, saxpyChosenMode, masterProcessorID, myCommWorld, processorID, nProcessor);

    /*
        Si stabilisce una barriera di sincronizzazione così da prendere correttamente il tempo di fine. Si calcola
il tempo totale per ogni processore e si ricava quello massimo. Infine, il processore "master" stampa il messaggio
di output per l'utente, salva il risultato nel file di outptut e rilascia la memoria allocata. Infine, si termina
l'ambiente MPI e il programma termina.
    */
    if ((errorCode = MPI_Barrier(myCommWorld) != MPI_SUCCESS)) raiseError(MPI_BARRIER_SCOPE, errorCode,
myCommWorld, FALSE);
    endTime = MPI_Wtime();
    totalTime = endTime - startTime;
    MPI_Reduce(& totalTime, & maxTime, 1, MPI_DOUBLE, MPI_MAX, masterProcessorID, myCommWorld);
    if (processorID == masterProcessorID) {
        if (fprintf(stdout, (const char * restrict) OUTPUT_USER_MESSAGE, outputFilePath, arraySize, alpha, maxTime)
< 0) {
            raiseError(FPRINTF_SCOPE, FPRINTF_ERROR, myCommWorld, FALSE);
        }

        saveResult(c, arraySize, outputFilePath, myCommWorld);
        releaseMemory(a, b, c, outputFilePath, (void *) 0);
    }

    if ((errorCode = MPI_Finalize() != MPI_SUCCESS)) raiseError(MPI_FINALIZE_SCOPE, errorCode, myCommWorld, FALSE);
    exit(0);
}
```