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 ed 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 ognuna delle quali si eseguirà la somma di un vettore con l'altro vettore che viene anticipatamente moltiplicato per uno scalare.

Keywords: MPI, *saxpy*

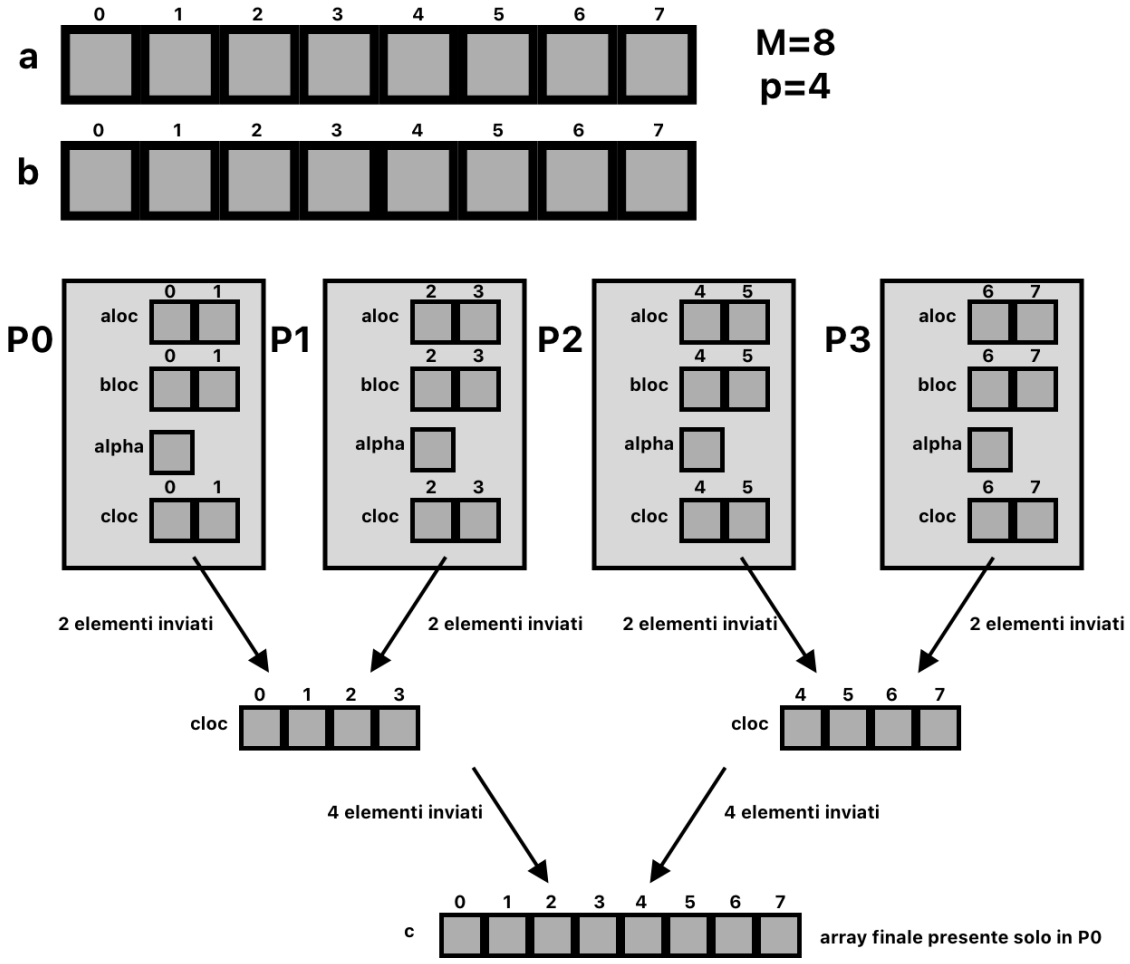
2. Descrizione dell'approccio parallelo

La strategia di parallelizzazione adottata è molto semplice. Dati p processori e due array di dimensione N contenenti i dati di input, è possibile assegnare ad ogni processore del cluster una porzione di $\frac{N}{p}$ 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 seguente, oppure un'operazione di *broadcast* finale dal processore *master* a tutti gli altri processori del vettore risultante ottenuto.

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 necessario ottenere l'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 o una moltiplicazione asintoticamente risulta essere simile, 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, allora si ottiene:

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

Da cui segue lo 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. In tal caso e considerando anche la non esatta divisibilità della dimensione del problema per il numero dei processori del cluster, si ottiene che:

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

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

Da questa relazione segue che lo speed-up è:

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

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:

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

E l'efficienza risulta essere:

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

L'isoefficienza risultante in questo caso risulta essere:

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

Infine, dall'applicazione della forma generalizzata della legge di Ware-Amdhal (viste le circostanze), si ottiene che:

$$S_p(N) = \frac{1}{\alpha_1 + \left(\sum_{k=2}^{p-1} \left(\frac{\alpha_k}{k} \right) \right) + \frac{\alpha_p}{p}} = ?$$

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 caspo *MPI_COMM_WORLD*). A questo punto si effettua una verifica sugli argomenti passati da linea di comando al programma (successiva al controllo già effettuata 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 contenente 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, allora 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 ora 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 dell'identificativo stesso del processore *master* (?) , 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 somme cumulative del numero di elementi 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, 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 presente solo e unicamente nel processore *master*. Infine, si procede col rilascio della memoria non più usata.

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

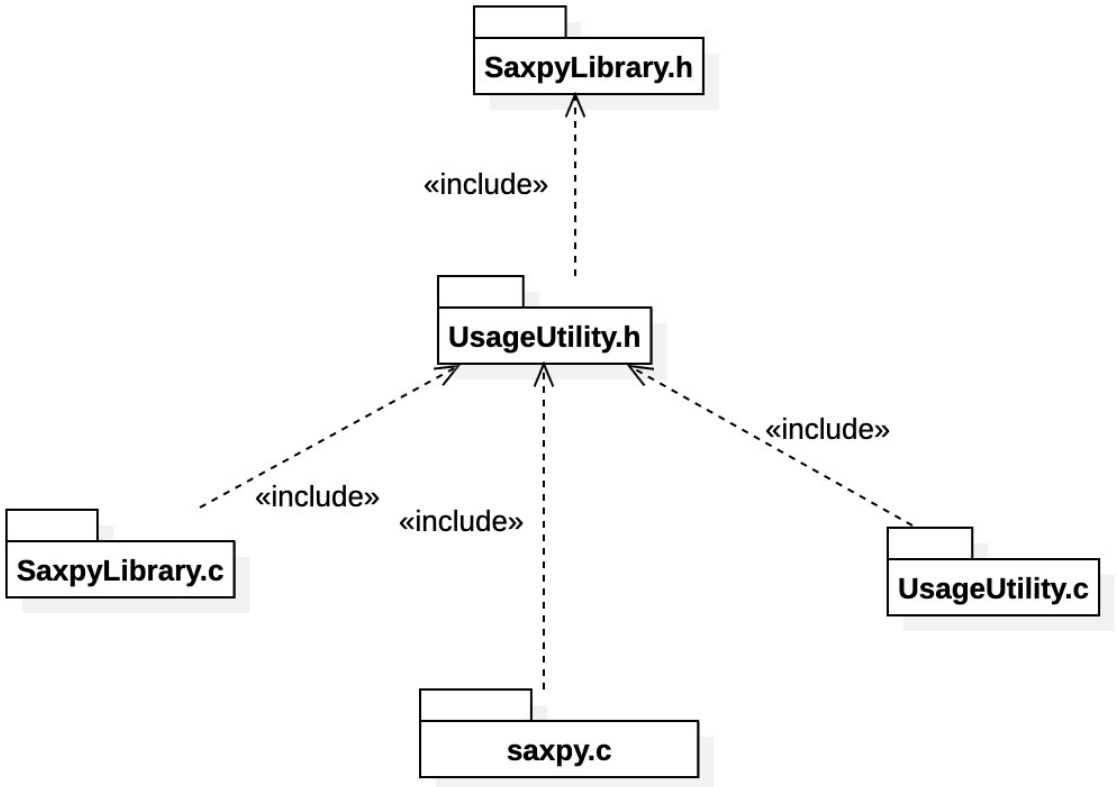
Si considerano ora le routine di MPI che sono state utilizzate.

- "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 un 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 3 di questo documento.
- "saxpy_sequential(...)" implementa l'operazione di *saxpy* in sequenziale.
- "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 mediante puntatori.
- "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 duante la scrittura dei sorgenti sono POSIX e ISO C99. Di seguito si allega uno schema utile al fine di capire quali sorgenti dell'architettura includono quali librerie. Vengono omesse le librerie di sistema.



6. Analisi delle performance del software

Prendere i tempi d'esecuzione e riportarli in tabelle e grafici significativi, al variare della dimensione dell'input e del numero di processori/core impiegati. Corredare lo studio anche con grafici di speed-up ed efficienza.

7. Esempi d'uso

Riportare esempi di esecuzione del software, così come appare a video. Se ci sono casi particolari o casi limite, riportare almeno un esempio.

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
- mpich.org/static/docs/
- stackoverflow.com, Jonathan Dursi
- mpi-forum.org/docs/mpi-1.1/mpi-11-html/node70.html#Node70
- valgrind.org

9. Appendice

Riportare il codice scritto, compresa la DOCUMENTAZIONE INTERNA: commentate opportunamente il codice perché sia di facile lettura e comprensione per chi lo analizza, che ne potrà dare così migliore valutazione. Per un eventuale approfondimento, si consiglia di consultare il sito:
http://www.nag.co.uk/numeric/FD/manual/html/FDlibrarymanual.asp
dove sono disponibili documentazioni esterne delle routine di una libreria (parallela) commerciale.