Framework MapReduce

Programmazione Multicore (2021-2022), Progetto N°11
Di Marco Andrea, 1825169
Dino Livio, 1844312

Consegna del Progetto

MapReduce è un framework molto utilizzato per realizzare applicazioni parallele in un sistema a memoria distribuita. Il **JobTracker** si occupa di gestire l'allocazione del workload tra i vari **nodi** del sistema, evitando al programmatore di doversi occupare dell'allocazione e sincronizzazione delle risorse. Il progetto prevede di sviluppare un Framework simile a MapReduce utilizzando **MPI** e **OpenMP** con le seguenti funzionalità:

- Funzioni per **File System Distribuito** (*Scatter*/*Gather* di un file su diversi nodi)
- **InputReader/OutputWriter**
- **Task coordination**
- **Map**
- **Reduce**

Il framework deve essere in grado di eseguire workload tipo ***wordcount*** su un file di testo e ***find possible friends***, in cui ad ogni utente del social network, vogliamo suggerire i possibili amici da inserire nella propria lista guardando la sua lista attuale e quella degli altri utenti.

Architettura

Cos'è MapReduce

MapReduce è un framework per gestire in modo parallelo grandi quantità di dati, sparsi su più computer o memorie. In questa sezione ne descriviamo le fasi principali in modo da rendere più chiaro il funzionamento.
MapReduce si divide in 6 fasi:

- **Input**: Il programma prende in input un file di testo. Nel nostro caso si tratta di file di parole o liste di amici.

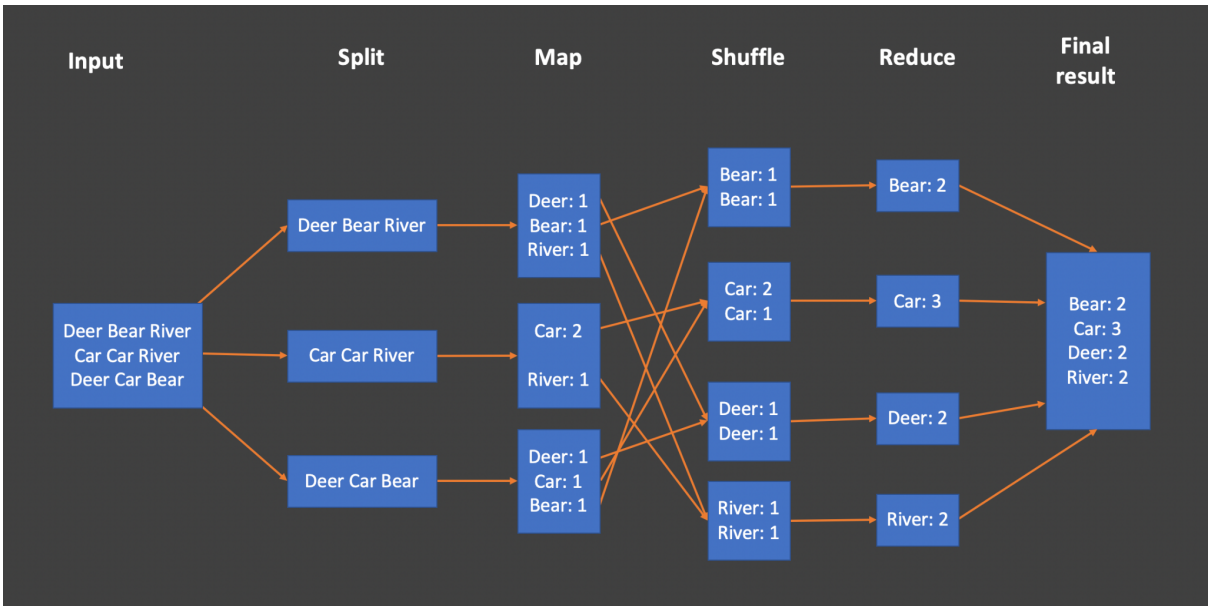
- **Splitting**: In questa fase si creano più thread ai quali vengono assegnati diverse parti del file in input così da iniziare la parallelizzazione del programma.

- **Map**: Ogni thread processa i dati che ha ottenuto, attraverso la funzione specifica, diversa in caso analizzi parole o liste di amici.

- **Shuffle**: Si ridistribuiscono i nuovi risultati fra i thread in base alla chiave dei dati, in modo da facilitare i conteggi.

- **Reduce**: Ogni thread parallelamente somma i risultati ottenuti e invia i dati al JobTracker

- **Final Result**: Il risultato viene inviato al JobTracker che lo mostra in output.



Scelte architetturali

In questa sezione descriviamo i ragionamenti che abbiamo fatto relativamente alle varie sezioni di MapReduce, e le scelte che abbiamo preso da implementare nei nostri programmi C.

Input

Abbiamo considerato l'idea di avere più file in input. Questo perché realisticamente si potrebbero voler analizzare più testi o più liste di amici contemporaneamente.

Nonostante fosse una buona idea, alla fine è stata scartata e abbiamo deciso di avere come input un singolo file, dato che suddividere le righe di più file avrebbe significato aggiungere una fase iterativa di smistamento ulteriore che avrebbe aggiunto complessità computazionale.

Splitting

Qui consideriamo che il file in input, sia per i file di testo che per la lista di amici, sia suddiviso per righe, ovvero col carattere '\n'. Questa sarà la feature principale per dare ai core la giusta quantità di dati in modo da rendere la distribuzione del workload più omogenea, per una questione di efficienza.

Ogni nodo **legge tutte le righe del fil**e, ma procederà con l'**analisi solo delle righe che gli spettano**. Non potendo sapere in anticipo quante righe di caratteri ha un file di testo scritto in linguaggio naturale, i nodi devono decidere sul momento se la riga è per loro oppure no, in questo modo:

Se la riga modulo il numero di nodi, ritorna il rank del nodo, allora quella riga è per lui.

La formula che utilizziamo è quindi la seguente:

```
if( (n_riga%comm_sz) == my_rank ) { … }
```

Dove

`n_riga` = numero della riga corrente
`comm_sz` = numero di nodi totali
`my_rank` = il rank del nodo corrente

Esempio con 4 thread:
riga 1 → Nodo 1
riga 2 → Nodo 2
riga 3 → Nodo 3
riga 4 → Nodo 4
riga 5 → Nodo 1 ← assegna di nuovo al Nodo 1
riga 6 → Nodo 2
…

Map

Una volta che i nodi calcolano i dati (nelle sezioni successive spiegheremo meglio) si pone il problema di salvare i dati processati.

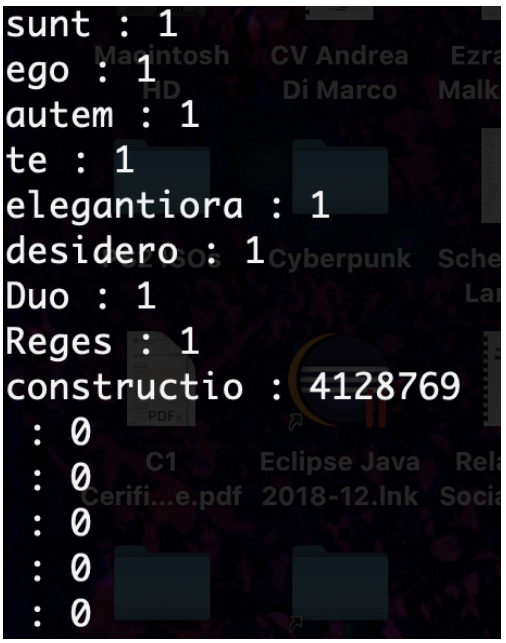
La prima idea fu quella di creare una struct contenente la parola e il conteggio.

```
typedef struct Entry {
   char parola[20];
    int occorrenza;
} Entry; /* 24 Byte <--- Data Structure Alignment */
```

Per i file piccoli, con poche parole diverse, la struttura funziona. Ma inserendo file più grandi i nostri computer sembrano avere problemi di memoria, nonostante la struttura pesi soltanto 24 Byte.
L'output da il seguente errore:

```
wordcount_seriale(25193,0x1033cedc0) malloc: *** error for
object 0x7fb65bc057a0: pointer being realloc'd was not
allocated
wordcount_seriale(25193,0x1033cedc0) malloc: *** set a
breakpoint in malloc_error_break to debug
Abort trap: 6
```

E nei casi in cui non dava dava errore, il risultato finale era questo:



Abbiamo quindi adottato l'idea di un "dizionario" composto da una matrice e una lista:

- La prima conterrà le parole/amici trovati
- La seconda conterrà il conteggio delle occorrenze di ciascuna

In questo modo l'indice delle due liste è quello che collega la parola al proprio conteggio.

Questa rappresentazione dei dati divenne però un problema nel momento in cui si deve passare la matrice `my_words` da una funzione all'altra. C non è molto clemente con le matrici.
Il massimo della complessità lo si è raggiunto quando abbiamo pensato di utilizzare matrici a 3 dimensioni per rappresentare i buffer di invio e ricezione delle parole di tutti i processi.

La soluzione è stata quella di utilizzare matrici (2D e 3D) a livello logico, ma di passarle nelle funzioni come semplici array molto lunghi, utilizzando la formula matematica qui sotto per scorrerle e leggerne i contenuti, perfezionata dopo numerosi `segmentation fault` ed errori vari:

`Matrix[y][x] = Array[(y*n_x) + x]`
con n_x = numero elementi di ogni riga. → numero di colonne.

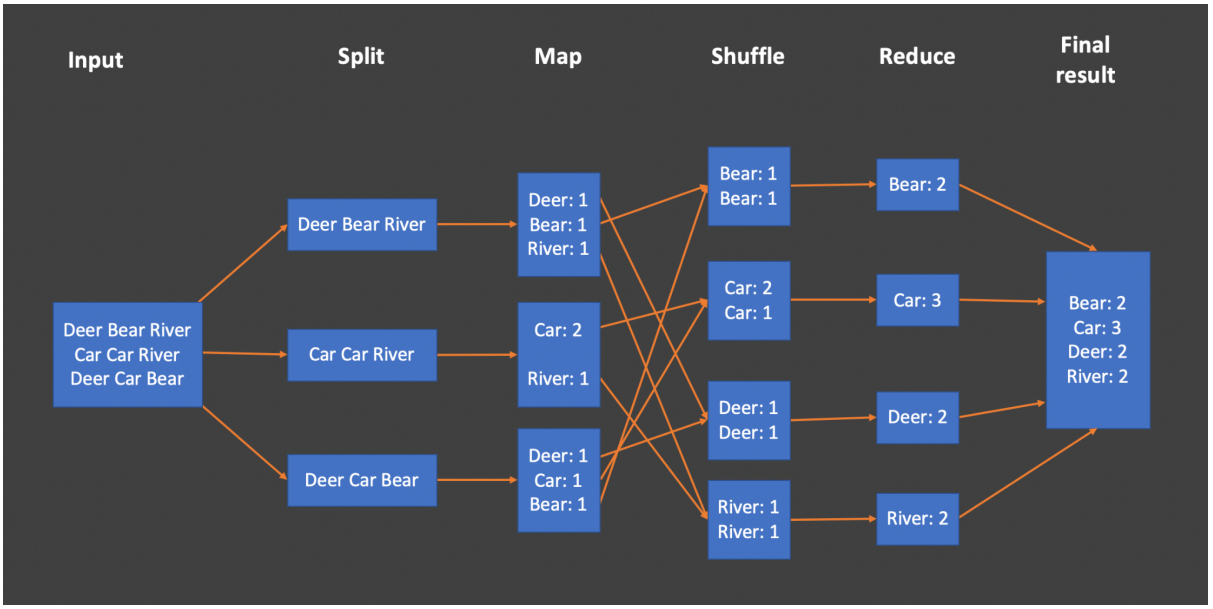
Nel caso di una matrice 3D, la formula sarebbe:

`Matrix[z][y][x] = Array [(z*n_y*n_x) + (y*n_x) + x]`
con n_x = numero elementi di ogni riga. → numero di colonne,
e n_y = numero elementi di ogni colonna. → numero di righe.

Shuffle

Questa fase è stata la più creativa e soddisfacente. La fase di shuffle è stata utilizzata molto di più con MPI, prenderemo i file di parole come esempio per spiegare meglio.

Abbiamo detto che lo shuffle serve per redistribuire le parole ottenute dai nodi ad altri nodi, questo perché se un nodo ottiene tutte le stesse parole da sommare, la somma di tutte le parole può avvenire in parallelo.



La sfida stava nell'inviare correttamente la parola giusta a nodo giusto, dopo diversi tentativi abbiamo raggiunto queste due soluzioni:

1. Per **inviare i risultati** abbiamo usato un sistema di messaggistica che comprendeva funzioni di send e receive, e buffer sia per il nodo mittente che per il nodo destinatario. In questo modo i messaggi vengono tenuti nel buffer di invio fino al momento della spedizione del messaggio, per poi entrare nel buffer del ricevente. Così il nodo può prendere finalmente il dato.

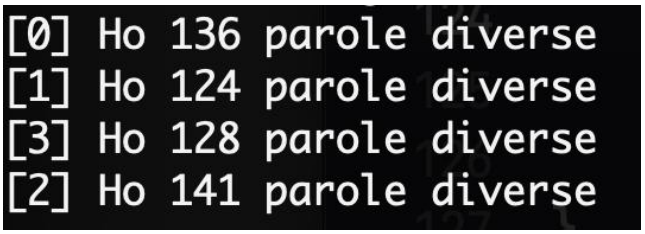
2. La **chiave** ovvia per questo programma è la parola in questione, vogliamo che la stessa parola sia inviata allo stesso nodo, ma in che modo dividerle equamente? Inizialmente avevamo pensato di dividerle alfabeticamente (*dalla A alla D al processo 0, dalla E alla H al processo 1, ...*) ma questa non sarebbe stata una divisione equa del carico di lavoro, in quanto alcune lettere sono molto più comuni di altre come iniziali.

   Ispirati dal corso di database, abbiamo definito una **funzione hash** che prende ogni `char` della parola e lo somma con tutti gli altri come fossero `int` e infine la funzione ritorna il resto della divisione tra il numero ottenuto e il numero di nodi, il risultato sarà proprio il nodo a cui si sarebbe dovuta inviare quella parola.
   Ogni processo, chiamando la funzione sulla stessa parola, sarebbe giunto alla stessa conclusione e quindi le parole sarebbero state correttamente smistate fra i nodi in base alla chiave. Presto realizzammo che non serviva sommare tutte le lettere, poteva bastare sommare solo le prime due, in questo modo la funzione sarebbe stata ancora più veloce.

Di Marco Andrea, 1825169
Dino Livio, 1844312

```
int funzione_hash(char* word, int limit)
{
    int len = strlen(word);
    int sum = 0;
        // somma tutti i caratteri
    for (int i = 0; (i < len) && (i <= 2); i++)
        sum += (int) word[i];
        // ritorna il numero entro limit
    return (sum%limit);
} /* Funzione Hash */
```

La nostra funzione hash divide correttamente il carico di lavoro tra i nodi in modo equo? **Sì.**
Come è possibile vedere dalla foto sotto, ogni nodo riceve più o meno lo stesso numero di parole durante la fase di shuffle.



Per i test abbiamo utilizzato lo script originale del film *Star Wars: A New Hope*.

Reduce

Nel reduce, ogni nodo effettua un interazione sul proprio dizionario e somma tutte le occorrenze delle parole uguali.

A questo punto facciamo convergere i dizionari aggiornati verso il JobTracker che darà in output il risultato finale.

Questo lo abbiamo fatto con funzioni di MPI specifiche oppure scrivendo su variabili globali, questa fase avviene in modo seriale con una sezione critica in modo che non ci siano problemi di scrittura sulla memoria del JobTracker, nelle sezioni dei vari programmi ci sono maggiori dettagli sulla fase di reduce e su come l'abbiamo implementata caso per caso.

Final result

Infine possiamo dare in output il risultato finale.

Wordcount

In questa sezione spieghiamo più nel dettaglio cos'è e come funziona il nostro programma `"wordcount"`, ed elenchiamo scelte, limitazioni incontrate ed eventuali ottimizzazioni effettuate.

Come abbiamo già detto, Wordcount è il programma che prende in input **un file di parole**. Specifichiamo che il file in input è della forma di un comune file di testo, scritto in linguaggio naturale. Per i nostri test abbiamo utilizzato la sceneggiatura del film *Star Wars: A New Hope*.

É importante precisare che un file di testo che consideriamo “normale” ha le seguenti caratteristiche:

- Le parole sono divise da spazi;
- Le righe terminano col carattere `'\n'`;
- Se la punteggiatura è presente si trova alla fine delle parole;

Programma seriale

Il programma seriale ha una struttura molto semplice:

- File in input
- Funzione che calcola le parole trovate
- Memorizzazione dei risultati
- Stampa dei risultati su standard output.

Per prima cosa prendiamo il file in input e lo apriamo con una semplice **`fopen(...)`**, dopo di che, ogni riga viene spezzettata prendendo una parola alla volta con la funzione di **`strtok(...)`** che venendo chiamata più volte ritornerà la parola successiva.

Ogni parola se unica (ovvero se non è già presente nella matrice `words`) viene salvata nella matrice `words` e il contatore delle occorrenze `count` viene inizializzato ad 1.

Nelle iterazioni sulle parole successive ogni parola non salvata nella matrice viene aggiunta e il count viene aggiornato. Se la parola è già stata trovata in passato non si aggiunge una nuova entry nella matrice ma si aumenta il contatore occorrenze di quella parola di **+1**.

Si itera in questo modo fino alla fine del file per poi stampare ogni parola in `words` e il suo contatore associato in `count`.

OpenMP

Scelte progettuali

In modo naturale, segue l’adattamento del programma seriale ad un programma parallelo utilizzando OpenMP.

Partiamo dal file: dato che openmp è un API che segue l’idea del **Globally Sequential Locally Parallel**, iniziamo l’esecuzione del programma con un singolo thread. L’idea iniziale era quella di far aprire il file al thread iniziale e poi spartirlo ai vari thread successivamente eseguiti. L’idea è stata scartata anche perché openMP non ha funzioni di messaggistica efficienti per inviare dati , quindi abbiamo direttamente dato il path del file da aprire ai thread così che ognuno potesse avere la sua copia senza ulteriori messaggi futili.

La fase di parallelizzazione è iniziata con **`#pragma omp parallel num_threads (thread_count)`** dove `thread_count` viene dato come parametro a runtime ed indica i thread totali da eseguire.
Con **`int my_rank= omp_get_thread_num();`** assegnare a ciascun thread il suo rank da usare successivamente per identificarlo.

Per la fase di mapping, ovvero della computazione delle parole abbiamo usato la funzione core descritta sopra nel programma seriale, con dei cambiamenti che sotto descriviamo.

Un **problema** è nato dalla fase di shuffling, ovvero in cui spartiamo ai thread per parallelizzare meglio i calcoli. Abbiamo consultato la documentazione di openMP e abbiamo pensato inizialmente di implementare lo shuffling con una struttura simile a **Produttori e Consumatori** visti anche a lezione, dove consideriamo delle code di uscita e entrata messaggi. Tuttavia non abbiamo optato per questa soluzione dato che openMP non ha funzioni di message-passing efficienti come le ha MPI, e quindi avremmo dovuto usare una memoria condivisa tra tutti i thread che avrebbe complicato e rallentato il programma.

Successivamente abbiamo pensato, una volta finiti i calcoli della funzione principale, a terminare la fase **Locally Parallel** e mandare i risultati al master thread per poi ri indirizzare ai thread destinatari per fare la somma delle occorrenze in modo parallelo.

Questa scelta sembra poco efficiente perché richiede di mandare in modo critico i dati al master thread per poi rinviare ai thread destinatari tutti i dati iterativamente da un solo thread. Poco efficiente!

infatti per aumentare l'efficienza, abbiamo scelto di **sommare le occorrenze direttamente scrivendo sul master thread** in modo critico con **#pragma omp critical** e da lì restituire il nostro output:

1) Un'idea era quella di usare una **reduction**, come visto a lezione, ma lavorando con matrici di parole la funzione di reduction non funzionava anche provando diversi simboli di operazione
2) Siamo andati sulla via giusta pensando a scrivere i risultati su un **variabile globale**, e abbiamo pensato di concatenare tutte le matrici di parole (private dei thread), ma nemmeno qui ha funzionato.

3) Scelta finale fu quella di scrivere su una **matrice globale my_words e un contatore globale my_count** in modo critico con **`#pragma omp critical`** scrivendo su un indice anch'esso globale (**index**) in modo che i thread si potessero accordare, incrementando **index** ogni volta che scrivono sulla matrice globale `my_words` e su `my_count`.

Il master thread una volta terminata la fase parallela dei thread applica una **#pragma omp barrier** che sincronizza i thread e si assicura che tutti abbiano salvato i risultati sulle variabili

globali, e rimane con **`my_words`** e **`my_count`** che devo solamente restituire in output con la funzione implementata **`stampa_conteggio(index, my_words, my_count)`**.

Performance OMP

Tempo di esecuzione

Il tempo è contato in secondi, test eseguiti sul PC di Livio.

N° Core	Tempo File Piccolo (60 KB)	Tempo File Medio (6 MB)	Tempo File Grande (20 MB)
seriale	0.548s	2.432s	5.822s
2	0.549s	1.629s	3.527s
4	0.566s	1.380s	2.655s
8	0.591s	1.307s	2.750s

Speed-Up

N° Core	Speed-Up File Piccolo (6 KB)	Speed-Up File Medio (6 MB)	Speed-Up File Grande (20 MB)
2	0,998	1,492	1,650
4	0,968	1,762	2,192
8	0,927	1,860	2,117

Efficienza

N° Core	Efficienza File Piccolo (6 KB)	Efficienza File Medio (6 MB)	Efficienza File Grande (20 MB)
2	0,499	0,746	0,825
4	0,242	0,440	0,548
8	0,115	0,232	0,264

MPI

Scelte progettuali

Tralasceremo le descrizioni delle pratiche standard di MPI come l'inizializzazione del canale di comunicazione e la terminazione con `MPI_Finalize()`.

Vogliamo simulare un network su cui eseguire il framework MapReduce immaginando ogni processore della nostra macchina come un computer a sé, con la sua memoria. Un sistema a memoria distribuita.

Inizialmente volevamo fare in modo che il JobTracker spartisse il file fra tutti i processi, ma questo approccio presenta diversi problemi:

1. L'analisi delle parole andrebbe fatta riga per riga, ma una riga è tale solo perché delimitata dai caratteri `'\n'`, quindi il nodo riconosce una riga solo dopo che l'ha letta. Per questo motivo, spartire le righe tra i processi non è conveniente in quanto il JobTracker dovrebbe leggere ogni riga da solo e poi inviarla al processo corretto. Questo ovviamente non è efficiente, anzi richiede addirittura più tempo dell'esecuzione seriale.

2. Spezzare il file ed inviarne i *chunk* ai vari processi è rischioso in quanto, essendo il file scritto in linguaggio naturale, non abbiamo modo di sapere che i chunk non vadano a spezzare le parole e quindi a produrre dei conteggi sbagliati.

Realisticamente, nel framework di MapReduce non si perde tempo a dividere i file tra i nodi, ma ogni nodo esegue le operazioni sui file che già possiede. Per questo motivo il nostro JobTracker comunica semplicemente il path del file che i nodi devono analizzare con `MPI_Bcast(...)`. Ogni nodo aprirà il file per conto suo e parallelamente agli altri nodi ne leggerà il contenuto e processerà i dati, ma solo per le righe che gli spettano.
Abbiamo provato diversi approcci, ma quello con `MPI_Bcast(...)` ci è sembrato il comprommesso migliore, senza andare contro la direttiva iniziale del progetto utilizzando `RMA`.
Uno degli approcci tentati è stato utilizzare `MPI_File_open(...)` che dovrebbe aprire lo stesso file su tutti i processi, ma se noi abbiamo già l'indirizzo, tanto vale usare la funzione `fopen(...)` di C.

La parte cruciale del MapReduce è la fase di Shuffle, come detto sopra, ogni parola deve essere inviata ad un nodo piuttosto che ad un altro in base alla **chiave**, per questo motivo un semplice `MPI_Scatter(...)` non avrebbe mai potuto funzionare in quanto non c'è garanzia che invii una stessa parola salvata nei `my_words` di due nodi diversi, allo stesso nodo. Senza contare che nonostante l'ordine in cui siano disposte le parole non ha importanza di per sé, ha invece molta importanza che le parole siano nello stesso ordine dei

conteggi in modo che alla parola `i` di `my_words` sia sempre associato correttamente il conteggio `i` all'interno di `my_counts`. L'unico modo possibile è quindi quello di utilizzare le funzioni di **send** e **receive** fornite da MPI.
Altro ostacolo, nel MapReduce i nodi che ricevono lo shuffle non sono necessariamente gli stessi che si sono occupati della fase precedente. Nel nostro caso, non avendo a disposizione abbastanza processori, e volendo utilizzare al massimo le nostre risorse, i nodi che riceveranno lo shuffle devono necessariamente essere gli stessi che prima si sono occupati della computazione. Era quindi necessario definire un nuovo modo per far comunicare ogni nodo con ogni altro nodo, sia in uscita che entrata, ed **in ordine!**
Analizziamo il problema per gradi

- **Buffer**: Le parole vanno quindi smistate in base alla chiave, ma non sarebbe efficiente inviare ogni parola separatamente. Un buon programmatore dovrebbe cercare di limitare il più possibile il numero di messaggi scambiati. Ogni nodo quindi, prima di iniziare le comunicazioni, scorre il proprio `my_words`, per ogni parola ne calcola l'hash ma invece di inviarla subito la salva nel buffer appositamente creato per il nodo ricevente. In questo modo le parole destinate ad un certo nodo **saranno inviate tutte insieme** come un unico grande array di `char`. Lo stesso per i conteggi.
  Il buffer di invio, definito come

  ```
  send_buffer[comm_sz*MAX_WORDS*WORD_LENGTH]
  ```

  sarà quindi un vettore di 3 dimensioni: Una per identificare i singoli caratteri, una per le parole ed una per i differenti nodi. All'indirizzo

  ```
  send_buffer[(i*MAX_WORDS*WORD_LENGTH)
        + (j*WORD_LENGTH)
        + k]
  ```

  Ci sarà quindi il k-esimo carattere della `j`-esima parola da dover inviare al nodo `i`.
  Servirà ovviamente definire un `receive_buffer` di egual misura dove inserire le parole ricevute dagli altri nodi. É necessario che i buffer (send e receive) siano in grado di contenere interamente il `my_words` di ogni nodo, nel caso peggiore in cui un nodo non solo abbia riempito il suo `my_words`, ma che ogni parola che ha trovato vada consegnata allo stesso nodo. Grazie alla funzione hash questi casi dovrebbero essere pressoché impossibili, ed infatti nei nostri test non è mai capitato. Meglio non rischiare però.
  É possibile che un nodo abbia delle parole da dover inviare a se stesso, in quel caso le inserisce direttamente nel suo `receive_buffer` durante la fase di comunicazione.

- **Messaggi**: L'ideale sarebbe che ogni processo inviasse tutto quelle che ha da inviare e poi si mettesse a ricevere tutto quello che deve ricevere, questo approccio presenta diversi problemi a seconda della funzione utilizzata:
  - `MPI_Send(...)`: Essendo bloccante, se tutti i nodi si mettono ad inviare senza nessuno che riceve porterà ad un deadlock il 100% delle volte.

- `MPI_Isend(...)`: Essendo non bloccante permetterebbe questo approccio, i dati da inviare vengono copiati sul buffer di invio e il programma può proseguire anche senza aspettare che vengano ricevuti, questo è particolarmente utile in quanto il nodo non deve poi andare a lavorare sui dati nel `send_buffer` quindi non servirebbe nemmeno la chiamata `MPI_Wait(...)`. Il problema però è che non abbiamo garanzie che il buffer di invio fornito da MPI sia abbastanza capiente da poter contenere tutti i dati che devo inviare, è possibile quindi che alcuni messaggi vengano persi.

Abbiamo quindi scelto un approccio intermedio, i send saranno non bloccanti ed i receive saranno bloccanti ma nel seguente modo.

- Tutti i nodi entrano nello stesso `for` ed inviano i dati solo quando è il loro turno, altrimenti si mettono a ricevere i dati dal nodo di turno. Il turno è dettato dalla variabile `i` del `for`.
  - Il primo `MPI_Recv(...)` deve essere per forza bloccante in quanto prende il numero di parole che il nodo `i` vuole inviare. É fondamentale che sia bloccante in quanto se `i` non ha parole da inviarmi (quindi ricevo `0`), non eseguo gli altri due receive per prendere le parole.
  - Il secondo ed il terzo receive possono essere bloccanti oppure no. Abbiamo testato sia con `MPI_Recv(...)` che con `MPI_Irecv(...)` ed un `MPI_Waitall(...)` alla fine di tutto, ma con le chiamate non bloccanti il programma risulta più lento di 10 millisecondi abbondanti rispetto al programma che utilizza `MPI_Recv(...)`. Per questo motivo, utilizziamo le chiamate bloccanti.
- Quando per il nodo è il turno di inviare i dati, entra in un secondo `for` ed effettua 3 `MPI_Isend(...)` in successione comunicando ad ogni altro quante parole deve mandargli e poi, ovviamente, inviando parole e dati.
  Utilizzare gli `Isend` non è più un rischio in quanto so che ci sono (o comunque ci saranno a breve) tutti gli altri nodi ad aspettare il primo messaggio con `MPI_Recv(...)`, per questo non corro il rischio di sovraccaricare il buffer di invio. I test sono risultati tutti positivi.

Inizialmente la fase era delimitata da due `MPI_Barrier()`, ci siamo resi subito conto del fatto che non solo non erano necessari, ma rallentavano l'esecuzione del programma, per questo li abbiamo tolti.

Un bug *simpatico* uscito fuori durante lo sviluppo di questa fase è stato il seguente:

```
[MacBook-Pro:32114] *** An error occurred in MPI_Recv
[MacBook-Pro:32114] *** reported by process [1006829569,3]
[MacBook-Pro:32114] *** on communicator MPI_COMM_WORLD
[MacBook-Pro:32114] *** MPI_ERR_COUNT: invalid count argument
[MacBook-Pro:32114] *** MPI_ERRORS_ARE_FATAL (processes in this communicator wil
l now abort,
[MacBook-Pro:32114] ***    and potentially your MPI job)
[MacBook-Pro.fritz.box:32110] 1 more process has sent help message help-mpi-erro
rs.txt / mpi_errors_are_fatal
[MacBook-Pro.fritz.box:32110] Set MCA parameter "orte_base_help_aggregate" to 0
to see all help / error messages
```

Dopo diversi controlli ci siamo resi conto che l'errore stava nella variabile che usavamo per segnare il numero di caratteri da inviare, erroneamente scrivevamo `send_buffer[i]` quando la variabile corretta sarebbe dovuta essere `send_buffer_index[i]`, l'errore è chiaramente che `send_buffer[i]` ritorna un carattere e non un intero, per questo motivo il compilatore ritorna "`invalid count argument`". La cosa divertente è che delle volte, approssimativamente 1 volta su 10, il programma ritornava il risultato corretto. Ad oggi non sappiamo come questo sia stato possibile.

A causa delle dimensioni degli array abbiamo dovuto limitare il numero di parole diverse a 1000 e la lunghezza delle parole a 30 caratteri.

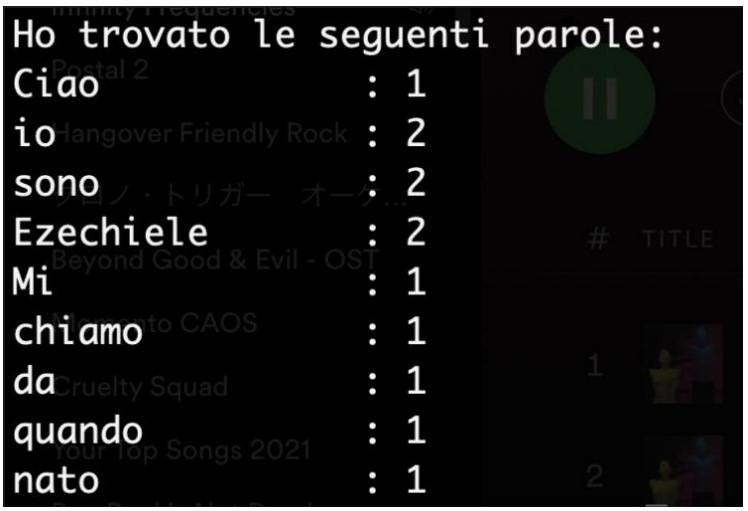
Dopo la fase di shuffle ogni nodo resetta il suo `my_words`, scorre tutte le parole nel `receive_buffer` e ne scrive il risultato finale su `my_words`.

Per quanto riguarda la fase di Reduce, ogni nodo deve consegnare i conteggi finali al JobTracker. Normalmente si dovrebbe utilizzare un `MPI_Gather(...)`, ma con questa architettura non possiamo farlo perché non c'è garanzia che il gather mantenga la correlazione tra gli indici delle parole e gli indici dei conteggi, salvati in due array diversi. Per questo motivo ogni nodo che non sia il JobTracker fa 3 `MPI_Send(...)` normali verso il JobTracker:

1. Comunica il numero di parole che ha
2. Se il numero è maggiore di zero invia le parole
3. E invia i conteggi

Abbiamo provato ad utilizzare `MPI_Isend(...)` invece del send bloccante ma ci siamo resi conto che non avrebbe fatto alcuna differenza in quanto questa è l'ultima azione di ogni nodo che non sia il JobTracker, quindi non cambia nulla se la chiamata è bloccante o meno.
Il JobTracker, nella fase di reduce chiama 3 `MPI_Recv(...)` per ogni nodo, che devono obbligatoriamente essere bloccanti in quanto i dati ricevuti verranno processati immediatamente.
Inizialmente il JobTracker salvava i messaggi nel `receive_buffer` e soltanto in un secondo momento li inseriva in `my_words`. Ci siamo resi subito conto che questo passaggio non era necessario e nella versione finale del programma i vari `MPI_Recv(...)` scrivono direttamente su `my_words`.

Finita la fase di reduce, il JobTracker stampa a schermo i risultati finali con il seguente formato.



Performance MPI

Tempo di esecuzione

Il tempo è contato in secondi, test eseguiti sul PC di Andrea.

N° Core	Tempo File Piccolo (60 KB)	Tempo File Medio (6 MB)	Tempo File Grande (20 MB)
Seriale	0.184s	1.389s	3.557s
2	0.180s	0.721s	1.867s
4	0.186s	0.452s	1.034s

Speed-Up

N° Core	Speed-Up File Piccolo (6 KB)	Speed-Up File Medio (6 MB)	Speed-Up File Grande (20 MB)
2	1.022	1.926	1.905
4	0.989	3.07	3.440

Efficienza

N° Core	Efficienza File Piccolo (6 KB)	Efficienza File Medio (6 MB)	Efficienza File Grande (20 MB)
2	0.511	0.963	0.952
4	0.247	0.767	0.860

Di Marco Andrea, 1825169
Dino Livio, 1844312

Find Possible Friends

Apriamo la seconda sezione dove spieghiamo più nel dettaglio cos'è e come funziona il nostro programma "`find_possible_friends`", ed elenchiamo scelte, limitazioni che abbiamo incontrato e eventuali ottimizzazioni effettuate.

`Find_possible_friends` è il programma che prende in input un file composto da liste di amici.
Importante è precisare che il file di liste di amici deve avere determinate **caratteristiche**:

- le righe sono formate da una persona
  A
  E dopo un '`\t`' l'elenco dei suoi amici separati da virgole
  B,C,D,E…

Nei nostri test abbiamo utilizzato delle singole lettere per identificare una persona, il programma funziona anche con nomi interi, nomi e cognomi e nomi composti da più parole

Esempio ↴

A	B,C,D,G
B	A,E
C	A,E
D	A,E
E	B,C,D
G	A,K
J	K
K	L,J,G
L	K

Programma seriale

Fin da subito ci siamo approcciati allo sviluppo con l'intento di creare una funzione cerca amici con la stessa interfaccia della funzione di `wordcount` descritta sopra, in modo tale da poterla sostituire a quest'ultima senza bisogno di dover modificare l'intero programma.

L'idea è quella di considerare le coppie di utenti come una parola intera, e quindi di utilizzare la struttura di `wordcount` per contare le occorrenze di quella "parola", ovvero quanti amici ha in comune quella coppia di utenti.

Per fare ciò abbiamo alzato la lunghezza minima di ogni parola a 50 caratteri, in questo modo ogni nome può essere lungo fino a 24 caratteri (serve spazio per il separatore `'|'` e carattere finale `'\0'`). Abbiamo infine lasciato a 1000 il numero di entry diverse per il dizionario, ovvero il numero di coppie di utenti diverse da poter analizzare.

Quello che fa il programma è quindi scorrere per ogni riga tutti gli utenti e salvarli in una lista `friends`.
A quel punto inizia a formare le coppie.
Due utenti devono essere scritti nella coppia sempre nello stesso ordine, così se il programma dovesse ritrovare la stessa coppia in un'altra riga, non commetterà l'errore di scrivere i nomi in un ordine diverso e contarla come una coppia nuova. Per fare ciò utilizziamo una variante della funzione hash vista sopra, che somma i primi due caratteri di una stringa e ritorna il valore. L'utente che ritorna il valore maggiore verrà scritto per primo nella coppia.

```
int ascii(char* word)
{
    int len = strlen(word);
    int sum = 0;
        // somma tutti i caratteri
    for (int i = 0; (i < len) && (i <= 2); i++)
        sum += (int) word[i];

    return sum;
} /* ASCII */
```

Questa funzione ci basta, dato che il nostro interesse non è scrivere i nomi in ordine alfabetico, ma solo scriverli sempre nello stesso ordine.

Dopo aver formato tutte le possibili coppie, il programma le inserirà dentro `my_words` secondo una di queste due opzioni:

1. Se gli utenti sono già amici (uno dei due è il primo della riga), il programma salverà la parola con un valore negativo pari a `-(MAX_WORDS+1)`.
   in questo modo la coppia, anche se trovata su altre righe, non potrà avere un valore positivo.

2. Altrimenti la coppia viene aggiunta al dizionario con valore 1. Se la coppia esiste già all'interno del dizionario il suo valore viene aumentato di 1.

La stampa finale omette tutte le coppie con valore negativo (che quindi sono già amici) e mostra i risultati nel seguente formato, del tutto simile al programma di `wordcount`.

```
Ho trovato le seguenti nuove amicizie possibili:
C|B             : 2
D|B             : 2
G|B             : 1
E|A             : 3
K|A             : 1
L|J             : 1
L|G             : 1
```

OpenMP / MPI

Scelte progettuali

L'idea principale per "`find_possible_friends`" è quella di mantenere la stessa struttura del programma di `wordcount(...)`, quindi manteniamo le stesse scelte progettuali di wordcount, cambiando solo la funzione principale di wordcount con la funzione `findpossiblefriends(...)`, a tale scopo abbiamo scritto la funzione in modo da essere compatibile al 100% con l'interfaccia di `wordcount`, maggiori dettagli nella sezione sul programma seriale.

Performance OpenMP

Tempo di esecuzione

Il tempo è contato in secondi, test eseguiti sul PC di Livio.

N° Core	Tempo File Piccolo (6 KB)	Tempo File Medio (6 MB)	Tempo File Grande (20 MB)
seriale	0.053s	0.893s	2.638s
2	0.058s	0.670s	1.873s
4	0.059s	0.571s	1.492s
8	0.053s	0.530s	1.314s

Speed-Up

N° Core	Speed-Up File Piccolo (6 KB)	Speed-Up File Medio (6 MB)	Speed-Up File Grande (20 MB)
2	0,913	1,332	1,408
4	0,898	1,563	1,768
8	1.000	1,684	2,007

Efficienza

N° Core	Efficienza File Piccolo (6 KB)	Efficienza File Medio (6 MB)	Efficienza File Grande (20 MB)
2	0,456	0,666	0,704
4	0,224	0,390	0,442
8	0,125	0,210	0,250

Performance MPI

Tempo di esecuzione

Il tempo è contato in secondi, test eseguiti sul PC di Andrea.

N° Core	Tempo File Piccolo (6 KB)	Tempo File Medio (6 MB)	Tempo File Grande (20 MB)
seriale	0.329s	0.587s	1.383s
2	0.176s	0.417s	0.901s
4	0.178s	0.336s	0.628s

Speed-Up

N° Core	Speed-Up File Piccolo (6 KB)	Speed-Up File Medio (6 MB)	Speed-Up File Grande (20 MB)
2	1.869	1.407	1.534
4	1.848	1.747	2.216

Di Marco Andrea, 1825169
Dino Livio, 1844312

Efficienza

N° Core	Efficienza File Piccolo (6 KB)	Efficienza File Medio (6 MB)	Efficienza File Grande (20 MB)
2	0.934	0.703	0.767
4	0.462	0.436	0.554

Conclusioni

Possiamo notare come, per quasi tutti i programmi, lo speed-up aumenta con l'aumentare del carico di lavoro.
Per i file più piccoli, l'overhead dovuto alla gestione delle comunicazioni fa sì che in alcuni i tempi di esecuzione dei programmi paralleli siano simili a quelli dei programmi seriali.

Aumentando il carico di lavoro, l'overhead dovuto alla comunicazione tra processi diventa sempre meno punitivo, portando ad uno speed-up notevole del programma parallelo rispetto a quello seriale.

Possiamo notare infine come l'efficienza dei programmi non diminuisca con l'aumentare del carico di lavoro, per questo possiamo giungere alla conclusione che i nostri programmi siano **scalabili**.