Word count – Mario Offertucci

Problema

L'obiettivo di word count è quello di contare il numero di parole che sono presenti in uno o più file al fine di raggiungere vari obiettivi come per esempio la raccolta di dati a fini statistici o la risoluzione di problemi legati al mondo dell'editoria o del giornalismo. Spesso i file di cui si vogliono contare le parole sono molto grandi, basti pensare per esempio ad un libro il quale può contenere anche milioni di parole, ragion per cui eseguire le operazioni di conta delle parole con un solo processore (o thread o processo) possono richiedere molto tempo. Si necessita quindi di un approccio distribuito che riesca a fornire delle prestazioni ben più elevate. Word count utilizza MPI per eseguire queste operazioni e contare le parole nel minor tempo possibile.

Soluzione

L'idea della soluzione è molto semplice: dividere il carico di lavoro equamente tra tutti i processori coinvolti nell'elaborazione così che non ci siano processori che debbano eseguire carichi di lavoro più elevati mentre altri invece sono in idle perché hanno terminato il proprio lavoro, e quindi velocizzare l'elaborazione.
A tal proposito le strade percorribili sono di due tipi:
* Distribuire lo stesso numero di parole tra tutti i processori;
* Distribuire lo stesso numero di bytes tra tutti i processori.

Distribuire lo stesso numero di parole tra tutti i processori

Questo approccio è stato subito scartato in quanto non era un'operazione semplice determinare quante parole fossero presenti nel file e distribuirle tra tutti i processori. Anche in questo caso erano possibili due soluzioni:
* Analisi statistica sul numero di byte;
* Conteggio delle parole da parte del master.

Nel primo caso si vuole provare a calcolare il numero di parole presenti nel documento facendo un'analisi statistica sul numero di bytes presenti nello stesso, ma questo metodo forniva risultati troppo aleatori innanzitutto perché la lunghezza media delle parole varia in base alla lingua del testo (6 caratteri per le parole Italiane e 5 per le parole Inglesi) e in secondo luogo perché questo calcolo era solamente una stima e non avrebbe portato alla distribuzione completamente equa del lavoro.

La seconda soluzione avrebbe permesso di poter asegnare lo stesso numero di parole a tutti i processori, ma altri problemi sorgono per questa soluzione: il processo master avrebbe dovuto leggere tutti i file per contare le parole impiegando tanto tempo e non era inoltre garantito che il lavoro fosse equamente distribuito in quanto le parole possono avere lunghezze molto diverse tra di loro.

Distribuire lo stesso numero di bytes tra tutti i processori

La soluzione è stata quella di dividere il numero di bytes equamente tra tutti i processori. Il numero di bytes di un file è facilmente leggibile e dal momento che un byte corrisponde ad un carattere (fatta eccezione per il carattere di carriage return) non sono state necessarie particolari conversioni.
Ad ogni processo vengono inviate tre informazioni:
* Nome del file da leggere;
* Offset di partenza da dove iniziare a leggere i byte;
* Offset di fine lettura

Questi tre elementi sono stati racchiusi un una struttura dati chiamata SubTask ed ogni processo riceve una lista di SubTask dal processo master e sequenzialmente legge quella lista di file partendo dall'offset iniziale fino all'offset finale.

È stato necessario rispondere ad alcune domande come:

- Che succede se l'offset di fine non corrisponde con la fine di una parola?
- Che succede se l'offset di inizio non corrisponde con l'inizio di una parola?

Se l'offset di fine non corrisponde con la fine di una parola il processo che sta leggendo continua fino a quando non trova un carattere di carriage return.
Se l'offet di inizio non corrisponde con l'inizio di una parola il processo che sta leggendo va avanti nella lettura e scarta tutto fino a quando non legge un carattere di carriage return. Scarta tutto perché ci sarà il processo precedente che avrà già letto quella parola.

Dettagli implementativi

- Di seguito è riportato il main del programma che è diviso in diverse fasi:
- Creazione dei Task da assegnare ai processi client;
- Invio informazioni ai processi client;
- Conteggio parole da parte dei processi client;
- Invio conteggio parole al master;
- Ordinamento parole da parte del master;
- Creazione CSV.

Creazione dei Task da assegnare ai processi client

Questa fase può essere divisa in
* Lettura dei file di input e calcolo bytes per ognuno;
* Calcolo del numero di bytes per ogni processo;
* Divisione dei file per processo in base al numero di byte per ognuno.

Lettura dei file di input e calcolo bytes per ognuno

Questa operazione viene eseguita con la funzione **getFilesInfos()** la quale chiama a sua volta la funzione **readFileNamesFromFile(fileNames)** che si occupa della lettura dell'input da file.
Dopo aver letto l'input si procede al recupero della dimensione dei singoli file di testo (contenuti nel file di input).

Le informazioni vengono salvate in una struttura dati chiamata FileInfo

```
typedef struct {
    char fileName[MAX_FILE_NAME_LENGTH];
    int fileSize;
} FileInfo;
```

```
FileInfo* getFilesInfos() {
    char fileNames[MAX_FILE_LIST_SIZE][MAX_FILE_NAME_LENGTH] = {""};
    FILE_NUMBER = readFileNamesFromFile(fileNames);

    FileInfo *filesInfos = malloc(FILE_NUMBER * sizeof(FileInfo));
    int i = 0;
    for (i = 0; i < FILE_NUMBER; i++) {
        FileInfo f;
        strcpy(f.fileName, fileNames[i]);
        f.fileSize = getFileSize(fileNames[i]);
        filesInfos[i] = f;
    }

    char message[2014];

    for (i = 0; i < FILE_NUMBER; i++) {
        sprintf(message, "[%s]\n|--+ SIZE: %d\n", filesInfos[i].fileName, filesInfos[i].fileSize);
        logMessage(message, MASTER_PROCESS_ID);
    }

    return filesInfos;
}
```

Questa funzione logga la size totale di ogni file

```
[Processo 0] >> [files/altri/01-The_Fellowship_Of_The_Ring.txt]
|--+ SIZE: 1025382
[Processo 0] >> [files/altri/02-The_Two_Towers.txt]
|--+ SIZE: 838056
[Processo 0] >> [files/altri/03-The_Return_Of_The_King.txt]
|--+ SIZE: 726684
[Processo 0] >> [files/altri/bible.txt]
|--+ SIZE: 4433689
[Processo 0] >> [files/altri/Book 1 - The Philosopher's Stone.txt]
|--+ SIZE: 474353
[Processo 0] >> [files/altri/Book 2 - The Chamber of Secrets.txt]
|--+ SIZE: 531212
[Processo 0] >> [files/altri/Book 3 - The Prisoner of Azkaban.txt]
|--+ SIZE: 676412
[Processo 0] >> [files/altri/Book 4 - The Goblet of Fire.txt]
|--+ SIZE: 1187504
[Processo 0] >> [files/altri/Book 5 - The Order of the Phoenix.txt]
|--+ SIZE: 1607344
[Processo 0] >> [files/altri/Book 6 - The Half Blood Prince.txt]
|--+ SIZE: 1058928
[Processo 0] >> [files/altri/Book 7 - The Deathly Hallows.txt]
|--+ SIZE: 1224900
[Processo 0] >> [files/altri/divina commedia.txt]
|--+ SIZE: 557042
[Processo 0] >> [files/610_parole_HP.txt]
|--+ SIZE: 6105
[Processo 0] >> [files/1000_parole_italiane.txt]
|--+ SIZE: 10197
[Processo 0] >> [files/6000_parole_italiane.txt]
|--+ SIZE: 622349
[Processo 0] >> [files/280000_parole_italiane.txt]
|--+ SIZE: 3326678
[Processo 0] >> [files/all hp books.txt]
|--+ SIZE: 6971646
```

Il file **input.txt** contiene al suo interno i nomi dei file di testo di cui si vogliono contare le parole.

```c
int readFileNamesFromFile(char fileNames[MAX_FILE_LIST_SIZE][MAX_FILE_NAME_LENGTH]) {
    FILE *input = fopen("input.txt", "r");
    char *line;
    size_t len = 0;
    int read = 0;
    int index = 0;
    while((read = getline(&line, &len, input)) != -1) {
        strcpy(fileNames[index], line);
        if (fileNames[index][read - 1] == '\n' || fileNames[index][read - 1] == '\r') {
            fileNames[index][read - 1] = '\0';
        }
        if (fileNames[index][read - 2] == '\n' || fileNames[index][read - 2] == '\r') {
            fileNames[index][read - 2] = '\0';
        }

        index++;
    }

    fclose(input);
    return index;
}
```

Esempio file **input.txt**.
Il path deve essere composto dal percorso completo del file che si vuole leggere e deve terminare con la corretta estensione.

```
files/altri/01-The_Fellowship_Of_The_Ring.txt
files/altri/02-The_Two_Towers.txt
files/altri/03-The_Return_Of_The_King.txt
files/altri/bible.txt
files/altri/Book 1 - The Philosopher's Stone.txt
files/altri/Book 2 - The Chamber of Secrets.txt
files/altri/Book 3 - The Prisoner of Azkaban.txt
files/altri/Book 4 - The Goblet of Fire.txt
files/altri/Book 5 - The Order of the Phoenix.txt
files/altri/Book 6 - The Half Blood Prince.txt
files/altri/Book 7 - The Deathly Hallows.txt
files/altri/divina commedia.txt
files/610_parole_HP.txt
files/1000_parole_italiane.txt
files/6000_parole_italiane.txt
files/280000_parole_italiane.txt
files/all hp books.txt
```

Calcolo del numero di bytes per ogni processo

Successivamente è necessario fare la somma di tutti i bytes per riuscire a dividerli equamente tra tutti i processi.

```c
int getTotalBytesFromFiles(FileInfo *fileInfo, int fileNumber) {
    int totalBytes = 0;
    int i = 0;
    for (i = 0; i < fileNumber; i++) {
        totalBytes += fileInfo[i].fileSize;
    }
    return totalBytes;
}
```

A questo punto bisogna effettuare le divisione vera e propria del carico di lavoro

```c
long* getNumberOfElementsPerProcess(int process_number, int elements_number) {
    if (process_number > elements_number) {
        // Division error
        sprintf(message, "THERE ARE TOO MANY PROCESSES\nPROCESSES: %d, TOTAL BYTES: %d\n", process_number, elements_number);
        logMessage(message, MASTER_PROCESS_ID);
        exit(0);
    }
    long resto = elements_number % process_number;
    long elements_per_process = elements_number / process_number;
    long *arrayOfElements = malloc(sizeof(long) * process_number);

    int i;
    for (i = 0; i < process_number; i++) {
        arrayOfElements[i] = i < resto ? elements_per_process + 1 : elements_per_process;
    }

    return arrayOfElements;
}
```

Il risultato prodotto sarà il seguente

```
[Processo 0] >> Process 0 should read: 8426161 bytes
[Processo 0] >> Process 1 should read: 8426160 bytes
[Processo 0] >> Process 2 should read: 8426160 bytes
```

Divisione dei file per processo in base al numero di byte per ognuno

Come ultimo passaggio bisogna dividere logicamente i file e organizzarli in triple

- Nome del file da leggere;
- Offset di partenza da dove iniziare a leggere i byte;
- Offset di fine lettura

Per fare questa divisione sono state necessarie le due seguenti strutture dati

```c
typedef struct {
    char fileName[MAX_FILE_NAME_LENGTH];
    long startFromBytes;
    long endToBytes;
} SubTask;

typedef struct {
    SubTask *subTasks;
    int size;
} Task;
```

SubTask rappresenta il task che verrà eseguito da uno dei processi, mentre Task rappresenta la lista di SubTasks che verrà assegnata ad ogni processo.

```
Task *divideFilesBetweenProcesses(long taskArraySize, long *taskArrayCurrentSize, long *arrayBytesPerProcess, FileInfo *filesInfos,
int numFiles) {
    long taskArrayMaximumSize = TASK_ARRAY_SIZE_2;
    Task *taskArray = calloc(taskArrayMaximumSize, sizeof(Task));
    *taskArrayCurrentSize = 0;
    long remainingBytesToRead = arrayBytesPerProcess[0];
    int currentFileInfo = 0;
    long startOffset = 0;
    int currentProcess = 0;
    int sendedBytes = 0;
    Task *task = newTask();
    long startFromBytes;
    long endToBytes;
    long subTaskArrayMaximumSize = TASK_ARRAY_SIZE_2;
    SubTask subTask = *newSubTask(subTask.fileName, subTask.startFromBytes, subTask.endToBytes);

    while (thereAreFilesToSplit(currentFileInfo, numFiles,currentProcess)) {
        startFromBytes = startOffset;
        if (canProcessReadWholeFile(filesInfos[currentFileInfo], startOffset, remainingBytesToRead)) {
            endToBytes = filesInfos[currentFileInfo].fileSize;
            setSubTask(&subTask, filesInfos[currentFileInfo], startOffset, endToBytes);
            sendedBytes = getSendedBytes(endToBytes, startFromBytes);
            remainingBytesToRead -= sendedBytes;
            subTaskArrayMaximumSize = addSubTask(task, subTaskArrayMaximumSize, subTask);

            currentFileInfo++;
            startOffset = 0;

            if (remainingBytesToRead == 0) {
                remainingBytesToRead = arrayBytesPerProcess[currentProcess];
                currentProcess++;
                *taskArrayCurrentSize = addTask(&taskArray, &taskArrayMaximumSize, *taskArrayCurrentSize, task);
                subTaskArrayMaximumSize = 0;
            }

        } else {
            endToBytes = startOffset + remainingBytesToRead;
            setSubTask(&subTask, filesInfos[currentFileInfo], startOffset, endToBytes);
            sendedBytes = getSendedBytes(endToBytes, startFromBytes);
            remainingBytesToRead -= sendedBytes;

            subTaskArrayMaximumSize = addSubTask(task, subTaskArrayMaximumSize, subTask);

            if (isFileEnded(sendedBytes, filesInfos[currentFileInfo], startOffset)) {
                currentFileInfo++;
                startOffset = 0;
            } else {
                startOffset = endToBytes;
            }

            if (remainingBytesToRead == 0) {
                *taskArrayCurrentSize = addTask(&taskArray, &taskArrayMaximumSize, *taskArrayCurrentSize, task);
                subTaskArrayMaximumSize = 0;
            }

            currentProcess++;
            remainingBytesToRead = arrayBytesPerProcess[currentProcess];
        }
    }

    return taskArray;
}
```

Il risultato prodotto sarà il seguente

```
[Processo 0] >> >>>>>> process[0]: files/altri/01-The_Fellowship_Of_The_Ring.txt, 0, 1025381
[Processo 0] >> >>>>>> process[0]: files/altri/02-The_Two_Towers.txt, 0, 838055
[Processo 0] >> >>>>>> process[0]: files/altri/03-The_Return_Of_The_King.txt, 0, 726683
[Processo 0] >> >>>>>> process[0]: files/altri/bible.txt, 0, 4433688
[Processo 0] >> >>>>>> process[0]: files/altri/Book 1 - The Philosopher s Stone.txt, 0, 474352
[Processo 0] >> >>>>>> process[0]: files/altri/Book 2 - The Chamber of Secrets.txt, 0, 531211
[Processo 0] >> >>>>>> process[0]: files/altri/Book 3 - The Prisoner of Azkaban.txt, 0, 396784
[Processo 0] >> >>>>>> process[1]: files/altri/Book 3 - The Prisoner of Azkaban.txt, 396785, 676411
[Processo 0] >> >>>>>> process[1]: files/altri/Book 4 - The Goblet of Fire.txt, 0, 1187503
[Processo 0] >> >>>>>> process[1]: files/altri/Book 5 - The Order of the Phoenix.txt, 0, 1607343
[Processo 0] >> >>>>>> process[1]: files/altri/Book 6 - The Half Blood Prince.txt, 0, 1058927
[Processo 0] >> >>>>>> process[1]: files/altri/Book 7 - The Deathly Hallows.txt, 0, 1224899
[Processo 0] >> >>>>>> process[1]: files/altri/divina commedia.txt, 0, 557041
[Processo 0] >> >>>>>> process[1]: files/610_parole_HP.txt, 0, 6104
[Processo 0] >> >>>>>> process[1]: files/1000_parole_italiane.txt, 0, 10196
[Processo 0] >> >>>>>> process[1]: files/6000_parole_italiane.txt, 0, 622348
[Processo 0] >> >>>>>> process[1]: files/280000_parole_italiane.txt, 0, 1872163
[Processo 0] >> >>>>>> process[2]: files/280000_parole_italiane.txt, 1872164, 3326677
[Processo 0] >> >>>>>> process[2]: files/all hp books.txt, 0, 6971645
```

La voce **[PROCESSO 0]** sulla sinistra indica che questa informazione di log è stata prodotta dal processo 0. È possibile distinguere per ogni processo quali file deve leggere, da dove iniziare a leggere e dove fermarsi. Sommando tutti i valori per ogni processo si ottiene:

```
Processo0 = 1025382 + 838056 + 726684 + 4433689 + 474353 + 531212 + 396785 = 8.426.161
Processo1 = (676412 - 396785) + 1187504 + 1607344 + 1058928 + 1224900 + 557042 + 6105 + 10197 + 622349 + 1872164 = 8.426.160
Processo2 = (3326678 - 1872164) + 6971646 = 8.426.160
```

:warning: NOTA:
Tra 0 e 1025381 ci sono in realtà 1025382 numeri ed è importante quindi sommare uno ad ogni valore quando si fa il calcolo di cui sopra.

Invio informazioni ai processi client

Una volta fatta la divisione occorre inviare questi dati ai processi client così che possano iniziare il loro lavoro.
È stato necessario inviare prima la dimensione dell'array di SubTasks e poi l'array stesso perché i processi client non potevano conoscere la dimensione dei dati da ricevere. Per rendere più veloci queste due operazioni sono state unite insieme (tramita MPI_Pack) le informazioni sulla size dell'array e l'array stesso in modo da ridurre al minimo il numero di comunicazioni.
Inoltre è stata utilizzata una send asincronca invece che una sincrona perché possono esserci molti processi a cui devono essere inviare i dati e non si vuole ogni volta attendere l'avvenuta ricezione da parte di un processo prima di effettuare il successivo invio.

```c
void scatterTasks(Task *taskArray, int taskArrayCurrentSize, MPI_Datatype subTaskType) {
    int taskIndex = 0;
    int position = 0;
    char message[PACK_SIZE];
    MPI_Request *requests = calloc(num_processes, sizeof(MPI_Request));
    for (taskIndex = 0; taskIndex < taskArrayCurrentSize; taskIndex++) {
        Task task = taskArray[taskIndex];
        long subTaskIndex = 0;
        position = 0;
        MPI_Pack(&task.size, 1, MPI_INT, message, PACK_SIZE, &position, MPI_COMM_WORLD);
        MPI_Pack(task.subTasks, task.size, subTaskType, message, PACK_SIZE, &position, MPI_COMM_WORLD);
        MPI_Isend(message, PACK_SIZE, MPI_PACKED, taskIndex + 1, TAG, MPI_COMM_WORLD, &requests[taskIndex]);
    }
    MPI_Status status;
    for (taskIndex = 0; taskIndex < taskArrayCurrentSize; taskIndex++) {
        MPI_Wait(&requests[taskIndex], &status);
    }
    free(requests);
}
```

Conteggio parole da parte dei processi client

Una volta che i processi client hanno ricevuto i dati possono iniziare a contare le parole. In accordo a quanto già accennato i processi riescono a determinare se la posizione di partenza corrisponde all'inizio di una parola oppure no:

- Se il processo ha come valore di startFromBytes 0 allora sicuramente sta leggendo la parola dall'inizio;
- Altrimenti se il carattere precedente al suo startFromBytes è uno spazio o uno '\n' allora sta leggendo la parola dall'inizio;
- Se il carattere precedente è un altro carattere non sta leggendo la parola dall'inizio e la scarta perché sarà letta da un altro processo.

Tutte le parole vengono conservate in un AVL così che sia molto rapida la ricerca di parole già contate e che necessitano di un aggiornamento di valore e l'ordinamento delle parole in base al numero di occorrenze. Sono state quindi necessarie le seguenti strutture dati per l'organizzazione dell'AVL.

```c
typedef struct {
    char word[MAX_WORD_SIZE];
    long occurrences;
} Item;

struct BTreeNode {
    char word[MAX_WORD_SIZE];
    long occurrences;
    int height;
    struct BTreeNode *left;
    struct BTreeNode *right;
};
```

Item è una struttura che viene utilizzata per wrappare le informazioni sulla parola e sul numero di occorrenze, mentre BTreeNode rappresenta un nodo dell'albero.

```c
struct BTreeNode* countWords(struct BTreeNode *btree, SubTask *subTask, int rank) {
    rank = rank - 1;
    FILE *file = fopen(subTask -> fileName, "r");
    long start = subTask -> startFromBytes;
    int remainingBytesToRead = subTask -> endToBytes + 1 - subTask -> startFromBytes;
    int lineSize = 1000;
    int parole = 0;
    char c;
    int chread = 0;
    if (start != 0) {
        fseek(file, start-1, SEEK_SET);
        c = fgetc(file);
        if (isCharacter(c)) {
            while (remainingBytesToRead > 0 && (c = fgetc(file)) != EOF && c != '\n' && c != ' ' && isCharacter(c)) {
                remainingBytesToRead--;
                chread++;
            }
            remainingBytesToRead--;
        }
    }
    int readedBytes;
    char line[lineSize];
    while (remainingBytesToRead > 0) {
        readedBytes = 0;
        while (remainingBytesToRead > 0 && (c = fgetc(file)) != EOF && (c == ' ' || c == '\r' || c == '\n' || !isCharacter(c))) {
            remainingBytesToRead--;
            chread++;
        }
        if (remainingBytesToRead <= 0) break;
        while (c != EOF && c != ' ' && c != '\r' && c != '\n' && isCharacter(c)) {
            line[readedBytes++] = c;
            c = fgetc(file);
        }
        line[readedBytes] = 0;
        if (c != EOF) readedBytes++;
        remainingBytesToRead -= readedBytes;
        parole++;
        chread += readedBytes;

        toLowerString(line);
        btree = addToAVL(btree, *newItemWithValues(line, 1), compareByName);
    }
    fclose(file);
    return btree;
}
```

La funzione isCharacter serve per considerare solamente i caratteri, vengono quindi scartati valori come virgole, apici, e quant'altro.

```c
int isCharacter(char ch) {
    return ch >= 65 && ch <= 90 || ch >= 97 && ch <= 122;
}
```

Invio conteggio parole al master

L'invio dei dati al master avviene sempre tramite comunicazione asincrona.

```
MPI_Request sizeRequest, wordsRequest;
MPI_Status status;
MPI_Isend(&size, 1, MPI_LONG, MASTER_PROCESS_ID, TAG, MPI_COMM_WORLD, &sizeRequest);
MPI_Isend(wordsList, size, itemType, MASTER_PROCESS_ID, TAG, MPI_COMM_WORLD, &wordsRequest);
MPI_Type_free(&itemType);

MPI_Wait(&sizeRequest, &status);
MPI_Wait(&wordsRequest, &status);
strcpy(message, "Data sended!\n");
logMessage(message, rank);

free(wordsList);
```

Ordinamento parole da parte del master

Il master riceve delle liste di Item da parte di client e deve necessariamente unirle per poter ordinare i dati.

```
int mergeData(struct BTreeNode *avl, long *size, int *rank) {
    struct BTreeNode *orderedByOccurrencesAVL = NULL;
    orderedByOccurrencesAVL = orderAVLByOccurrences(avl, orderedByOccurrencesAVL);
    free(avl);
    strcpy(message, "Creating CSV...\n");
    logMessage(message, *rank);
    long wordsNumber = 0;
    createCSV(orderedByOccurrencesAVL, *size, *rank, &wordsNumber);
    free(orderedByOccurrencesAVL);
    strcpy(message, "CSV created!\n");
    logMessage(message, *rank);
    return wordsNumber;
}
```

La funzione orderAVLByOccurrences si occupa di ordinare l'AVL ricevuto in base al numero decrescente di occorrenze delle parole.

```
struct BTreeNode* orderAVLByOccurrences(struct BTreeNode *oldTree, struct BTreeNode *newTree) {
    if (oldTree != NULL) {
        newTree = orderAVLByOccurrences(oldTree -> left, newTree);
        newTree = addToAVL(newTree, *newItemWithValues(oldTree -> word, oldTree -> occurrences), compareByOccurrences);
        newTree = orderAVLByOccurrences(oldTree -> right, newTree);
    }
    return newTree;
}
```

La funzione addToAVL prende come terzo argomento una funzione che verrà utilizzata per determinare l'ordine nell'AVL.
In primo luogo gli elementi all'interno dell'AVL vengono ordinati alfabeticamente per rendere veloce l'individuazione di parole già esistenti in fase di conteggio delle parole.

Quando il master riceve i dati dai client viene utilizzata la funzione di ordinamento compareByOccurrences perché in questo caso si vuole che l'AVL sia ordinato per numero di occorrenze decrescente.
Questa soluzione rende il codice di addToAVL riutilizzabile e riduce al minimo la duplicazione di codice.

```
int compareByName(Item *item, struct BTreeNode *btreeNode) {
    return strcmp(item -> word, btreeNode -> word);
}

int compareByOccurrences(Item *item, struct BTreeNode *btreeNode) {
    if (item -> occurrences > btreeNode -> occurrences) return -1;
    if (item -> occurrences < btreeNode -> occurrences) return 1;
    return compareByName(item, btreeNode);
}

struct BTreeNode* addToAVL(struct BTreeNode *btree, Item item, int (*compareFunction)()) {
    ...
}
```

:information_source: In un primo momento era stato utilizzato un semplice BTree che però dava problemi nel caso in cui l'input fosse già ordinato (caso peggiore) riducendo di fatto le performance a quelle di una lista (l'albero si sviluppava solo a sinistra o solo a destra). L'AVL risolve questo problema eseguendo una rotazione dei nodi nel momento in cui gli elementi si sviluppano solamente verso sinistra o solamente verso destra.

Creazione CSV

La funzione createCSV si occupa di produrre il CSV di output.

```
void createCSV(struct BTreeNode *wordsTree, long size, int rank, long *wordsNumber) {
    if (size < 0) return;

    char fileName[MAX_FILE_NAME_LENGTH];
    sprintf(fileName, "output.csv");
    FILE *output = fopen(fileName, "w");
    fprintf(output, "WORD,COUNT\n");

    writeTree(wordsTree, output, wordsNumber);

    fclose(output);
}
```

Funzione main

```
int main(int argc, char **argv) {
    int rank;

    MPI_Datatype subTaskType, itemType;
    MPI_Status status;

    SubTask subTask;
    struct BTreeNode *avl;
    long size = TASK_ARRAY_SIZE;
    double startTime;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (num_processes == 1) {
        strcpy(message, "THERE ARE TOO FEW PROCESSES! PLEASE RUN WITH 2 OR MORE PROCESSES\n");
        logMessage(message, rank);
        exit(0);
    }

    num_processes -= 1;

    createSubTaskMPIStruct(&subTaskType);
    createItemMPIStruct(&itemType);

    if (rank == MASTER_PROCESS_ID) {
        FileInfo *filesInfos = getFilesInfos();
        int totalBytes = getTotalBytesFromFiles(filesInfos, FILE_NUMBER);
        Task *taskArray = calloc(TASK_ARRAY_SIZE_2, sizeof(Task));
        long *taskArrayCurrentSize = malloc(1 * sizeof(long));
        long *arrayBytesPerProcess = getNumberOfElementsPerProcess(num_processes, totalBytes);
        printArray(arrayBytesPerProcess, 0, num_processes);
        taskArray = divideFilesBetweenProcesses(TASK_ARRAY_SIZE_2, taskArrayCurrentSize, arrayBytesPerProcess, filesInfos, FILE_NUMBER);
        printTaskArray(taskArray, *taskArrayCurrentSize);

        startTime = MPI_Wtime();
        scatterTasks(taskArray, *taskArrayCurrentSize, subTaskType);

        free(filesInfos);
        free(taskArray);
        free(taskArrayCurrentSize);
        free(arrayBytesPerProcess);
        MPI_Type_free(&subTaskType);

        char *packMessage = calloc(PACK_SIZE, sizeof(char));

        int taskIndex;
        MPI_Request *requests = calloc(num_processes, sizeof(MPI_Request));
        long wordsSize[num_processes];
        Item **wordsList = calloc(num_processes, sizeof(Item));

        avl = receiveDataFromClients(avl, num_processes, wordsList, wordsSize, itemType, requests);

        free(requests);
        free(wordsList);
        MPI_Type_free(&itemType);

        long wordsNumber = mergeData(avl, &size, &rank);
        double endTime = MPI_Wtime();
        double totalTime = endTime - startTime;
        sprintf(message, "Processed %ld words in %f seconds\n", wordsNumber, totalTime);
        logMessage(message, rank);
    } else {
```

```
    } else {
        long wordsListSize = TASK_ARRAY_SIZE_2;
        Item *wordsList = calloc(wordsListSize, sizeof(Item));
        Task *task = newTask();
        int position = 0;
        char *packMessage = calloc(PACK_SIZE, sizeof(char));
        MPI_Recv(packMessage, PACK_SIZE, MPI_PACKED, MASTER_PROCESS_ID, TAG, MPI_COMM_WORLD, &status);
        MPI_Unpack(packMessage, PACK_SIZE, &position, &task -> size, 1, MPI_INT, MPI_COMM_WORLD);
        task -> subTasks = calloc(task -> size, sizeof(SubTask));
        MPI_Unpack(packMessage, PACK_SIZE, &position, task -> subTasks, task -> size, subTaskType, MPI_COMM_WORLD);
        avl = processTasks(task, &subTask, avl, &wordsList, &wordsListSize, &size, &rank);

        free(task);
        free(packMessage);
        MPI_Type_free(&subTaskType);

        strcpy(message, "Word counted! Sending data to master process...\n");
        logMessage(message, rank);

        MPI_Request sizeRequest, wordsRequest;
        MPI_Status status;
        MPI_Isend(&size, 1, MPI_LONG, MASTER_PROCESS_ID, TAG, MPI_COMM_WORLD, &sizeRequest);
        MPI_Isend(wordsList, size, itemType, MASTER_PROCESS_ID, TAG, MPI_COMM_WORLD, &wordsRequest);
        MPI_Type_free(&itemType);

        MPI_Wait(&sizeRequest, &status);
        MPI_Wait(&wordsRequest, &status);
        strcpy(message, "Data sended!\n");
        logMessage(message, rank);

        free(wordsList);
    }

    MPI_Finalize();
    return 0;
}
```

Correttezza dell'algoritmo

La correttezza non è stata formalmente provata ma sono state fatte diverse prove sia con vari file di input semplici (così da poter controllare a mano la correttezza) sia con un diverso numero di processi e i risultati sono stati uguali per ogni esecuzione.
I file di input per i test si trovano nella cartella files/tests.

Benchmarks

Sono stati eseguiti test relativi sia alla scalabilità forte che alla scalabilità debole. Per entrambi i test è stato utilizzato il seguente file ```input.txt```. I file di testo sono stati facilmente reperiti sul web.

```
files/01-The_Fellowship_Of_The_Ring.txt
files/02-The_Two_Towers.txt
files/03-The_Return_Of_The_King.txt
files/bible.txt
files/Book 1 - The Philosopher's Stone.txt
files/Book 2 - The Chamber of Secrets.txt
files/Book 3 - The Prisoner of Azkaban.txt
files/Book 4 - The Goblet of Fire.txt
files/Book 5 - The Order of the Phoenix.txt
files/Book 6 - The Half Blood Prince.txt
files/Book 7 - The Deathly Hallows.txt
files/divina commedia.txt
files/610_parole_HP.txt
files/1000_parole_italiane.txt
files/6000_parole_italiane.txt
files/280000_parole_italiane.txt
files/all hp books.txt
```

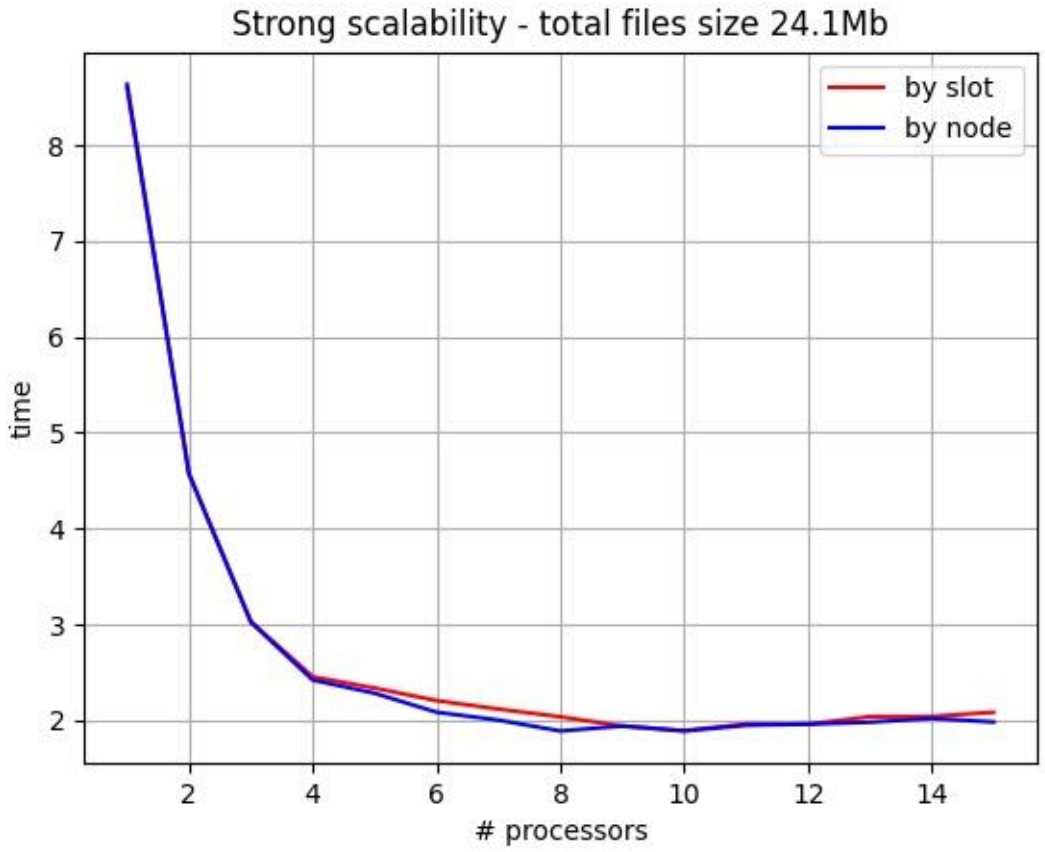
I test sono stati eseguiti con un cluster di 4 macchine AWS di tipo EC2 t2.xlarge dotate di 4 vCPUs e 16GB di ram.
Sono stati eseguiti test con due diverse opzioni di scheduling:

- by slot: un certo carico di lavoro viene assegnato ad un processore diverso solo quando tutti gli slot del processore "precedente" sono pieni. È attivabile avviando mpicc con l'opzione --map-by slot;
- by node: un certo carico di lavoro viene assegnato in logica round robin a tutti i processori coinvolti. È attivabile avviando mpicc con l'opzione --map-by node.

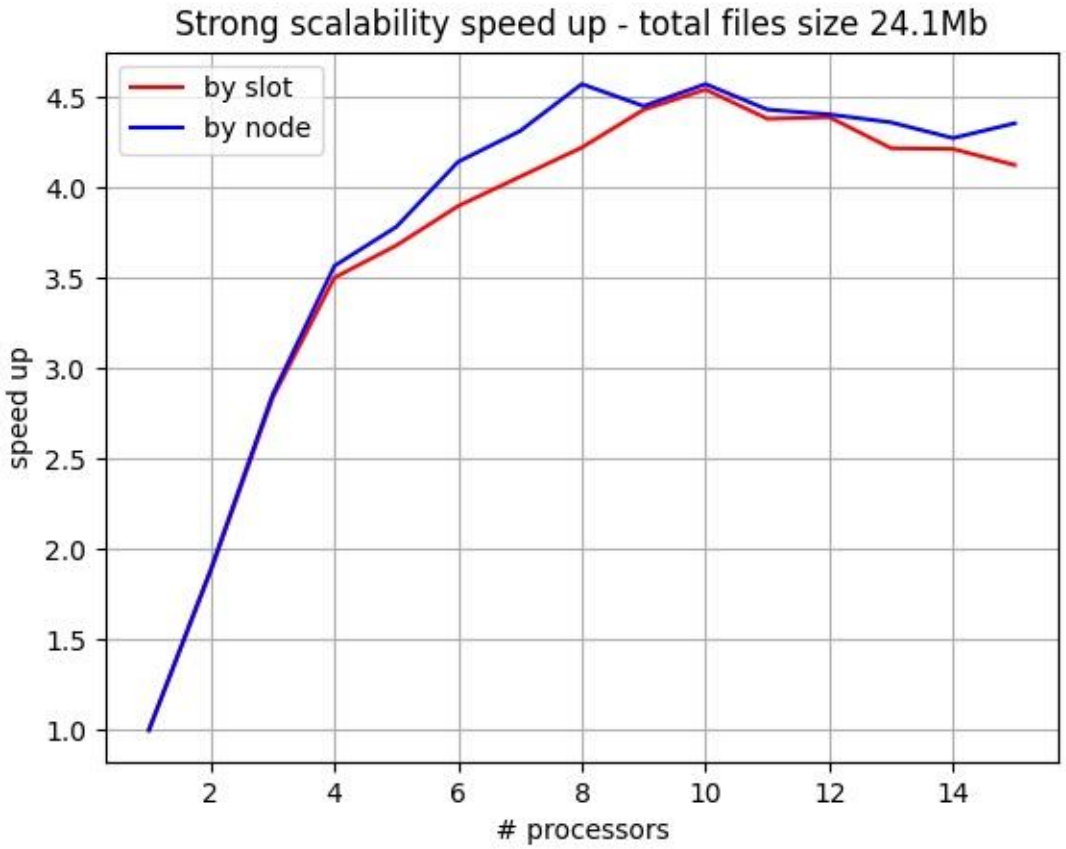
Scalabilità forte

Test con una taglia totale di 24.1Mb

# processors	Tyme by slot (seconds)	Time by node (seconds)	Speed up by slot	Speed up by node
1	8.6	8.638	1	1
2	4.572	4.572	1.881	1.889
3	3.03	3.024	2.838	2.856
4	2.454	2.42	3.504	3.569
5	2.336	2.282	3.682	3.785
6	2.205	2.084	3.9	4.145
7	2.117	2.002	4.062	4.315
8	2.036	1.888	4.224	4.575
9	1.94	1.94	4.433	4.453
10	1.892	1.888	4.545	4.575
11	1.962	1.948	4.383	4.434
12	1.958	1.96	4.392	4.407
13	2.038	1.98	4.22	4.368
14	2.04	2.02	4.216	4.276
15	2.084	1.982	4.127	4.358

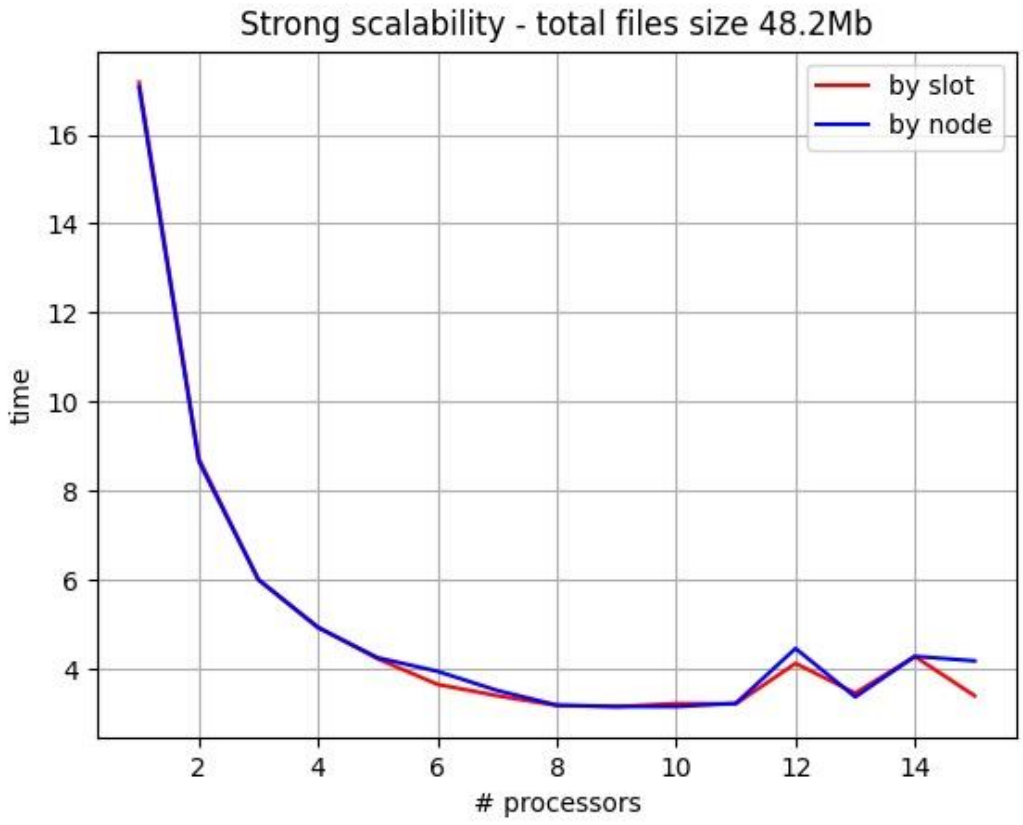


Speed up

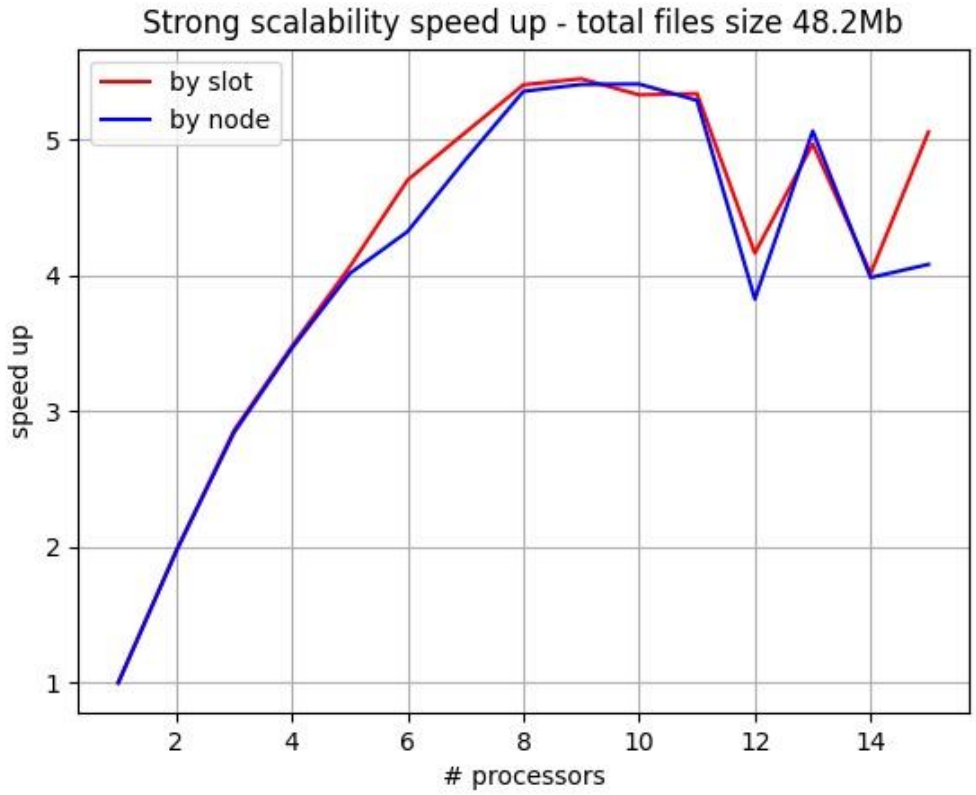

Strong scalability speed up - total files size 24.1Mb
by slot
by node
speed up
processors
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
2
4
6
8
10
12
14

Test con una taglia totale di 48.2Mb

# processors	Tyme by slot (seconds)	Time by node (seconds)	Speed up by slot	Speed up by node
1	17.174	8.638	1	1
2	8,728	8,666	1,968	1,969
3	6,006	6,002	2,859	2,843
4	4,936	4,926	3,479	3,464
5	4,226	4,25	4,064	4,015
6	3,652	3,946	4,703	4,324
7	3,398	3,516	5,054	4,853
8	3,178	3,186	5,404	5,356
9	3,152	3,156	5,449	5,407
10	3,222	3,154	5,33	5,41
11	3,216	3,226	5,34	5,29
12	4,124	4,46	4,164	3,826
13	3,456	3,37	4,969	5,064
14	4,278	4,282	4,014	3,985
15	3,396	4,18	5,057	4,082



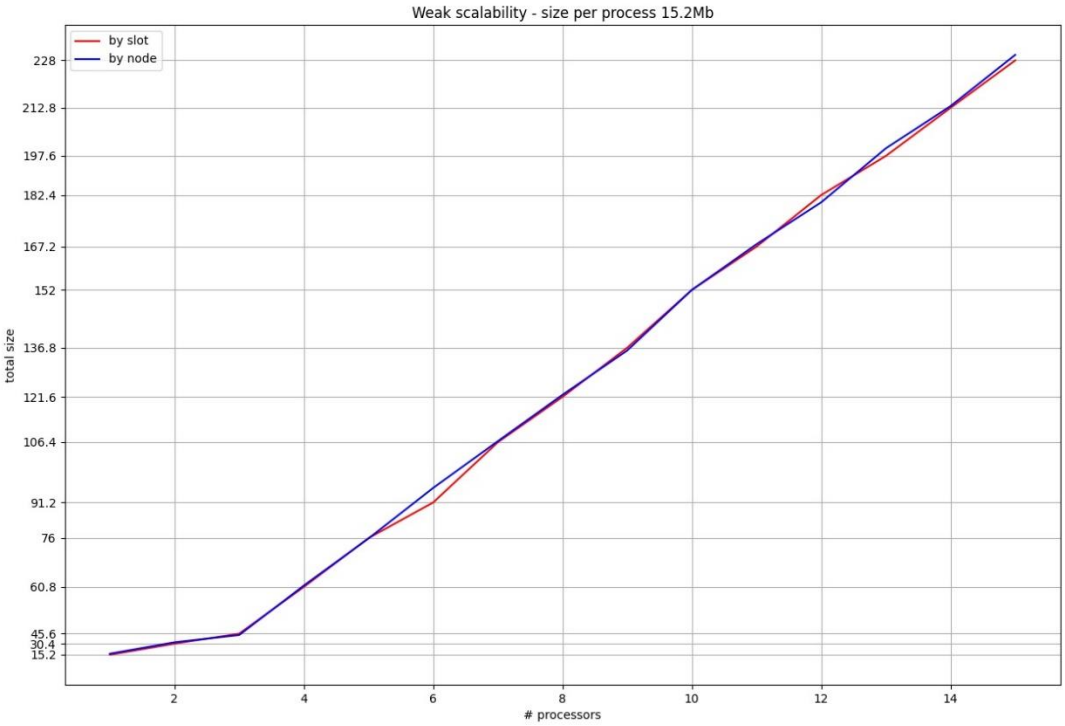
Speed up



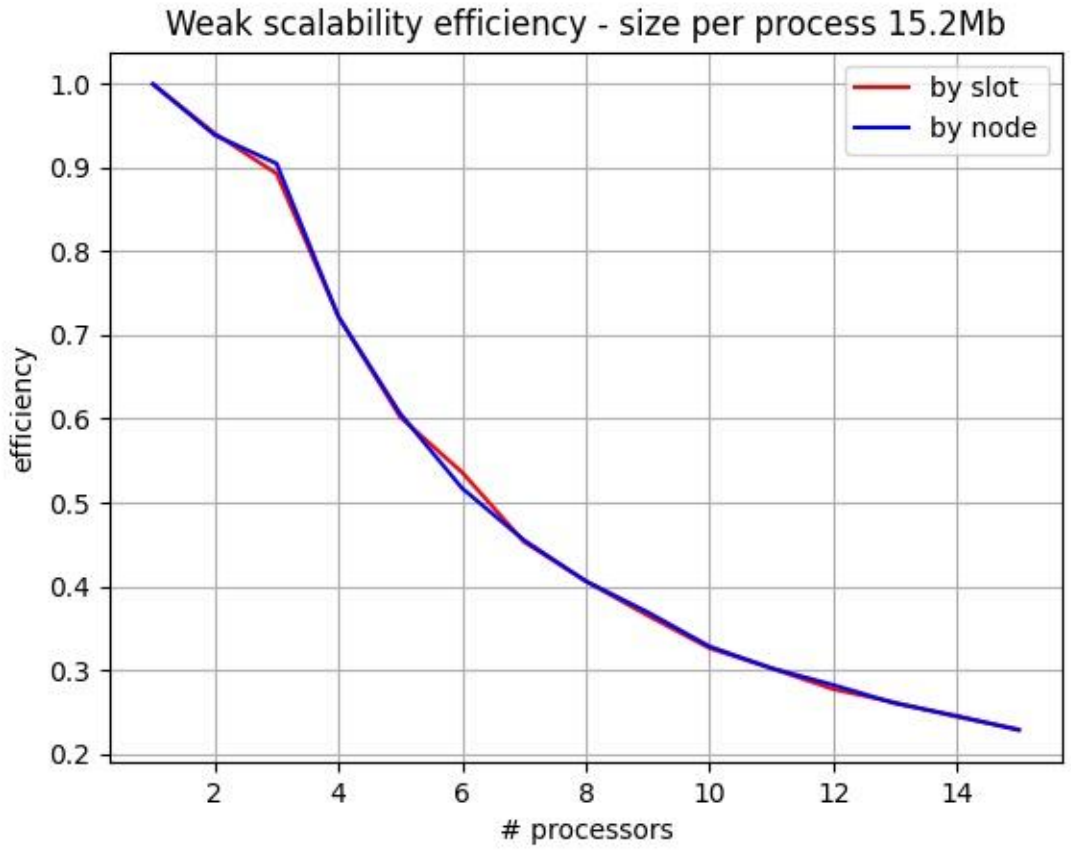
Scalabilità debole

Test con una taglia totale di 15.2Mb

# processors	Total file size	Tyme by slot (seconds)	Time by node (seconds)	Speed up by slot	Speed up by node
1	15,2	5,09	5,12	1	1
2	30,4	5,41	5,45	0,941	0,939
3	45,6	5,7	5,66	0,893	0,905
4	60,8	7,05	7,09	0,722	0,722
5	76	8,46	8,45	0,602	0,606
6	91,2	9,49	9,91	0,536	0,517
7	106,4	11,23	11,26	0,453	0,455
8	121,6	12,54	12,6	0,406	0,406
9	136,8	13,96	13,88	0,365	0,369
10	152	15,63	15,63	0,326	0,328
11	167,2	16,88	16,95	0,302	0,302
12	182,4	18,37	18,16	0,277	0,282
13	197,6	19,5	19,72	0,261	0,26
14	212,8	20,89	20,94	0,244	0,245
15	228	22,26	22,42	0,229	0,228

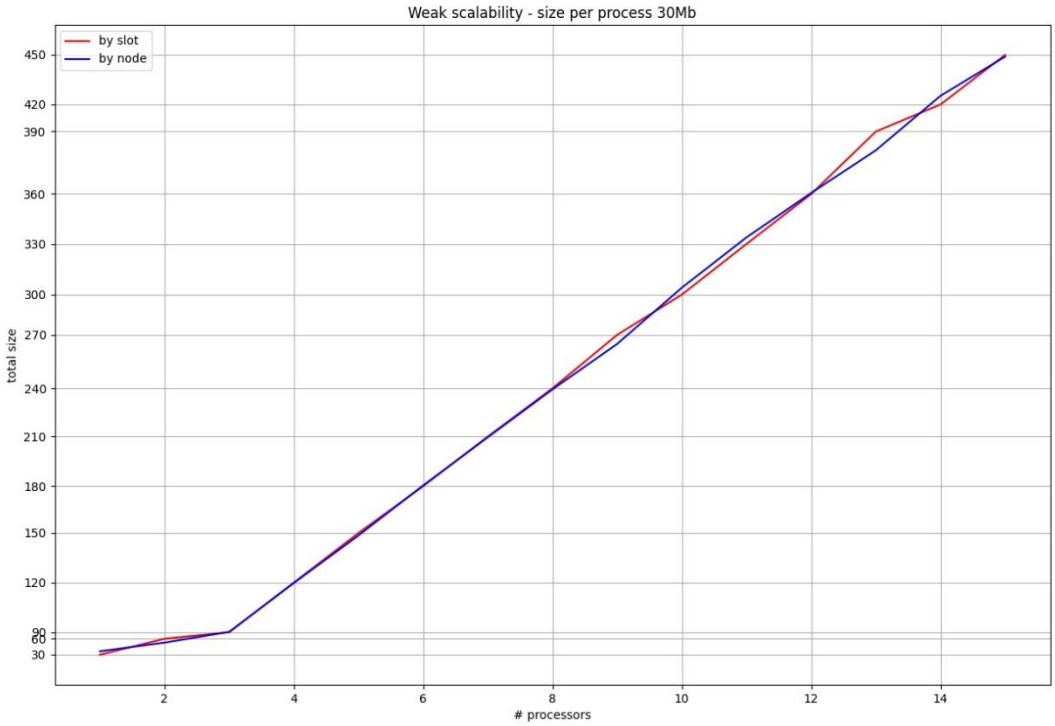

Weak scalability - size per process 15.2Mb
by slot
by node
total size
228
212.8
197.6
182.4
167.2
152
136.8
121.6
106.4
91.2
76
60.8
45.6
30.4
15.2
2
4
6
8
10
12
14
processors

Efficiency

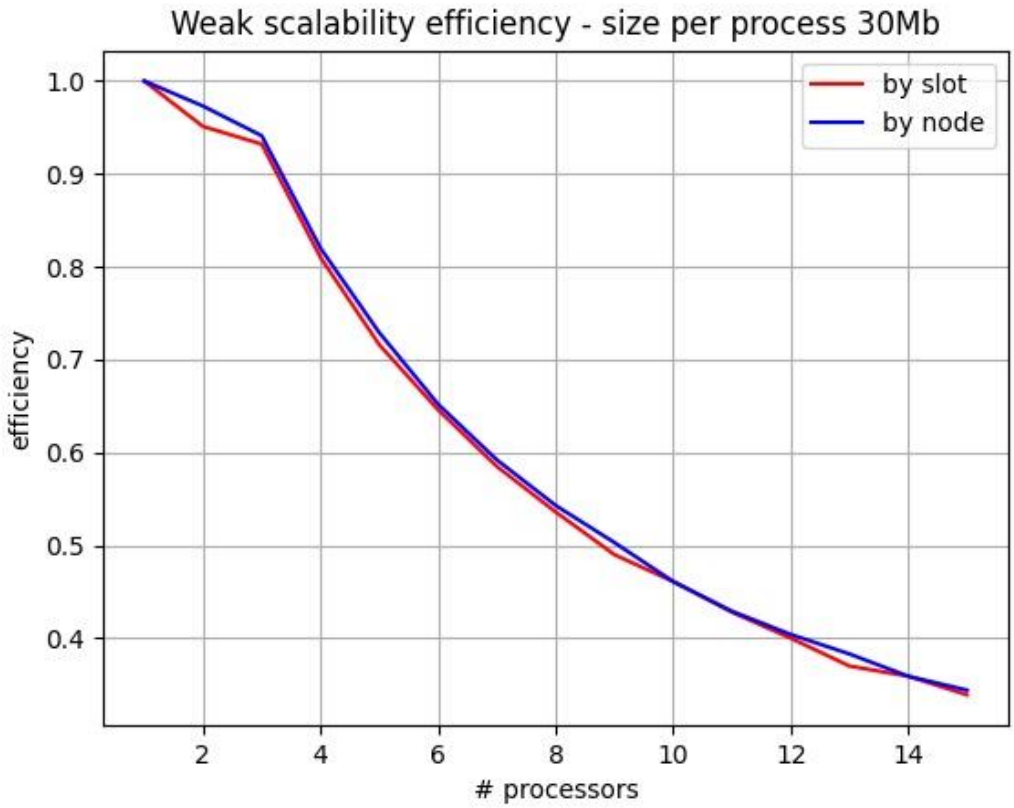

Weak scalability efficiency - size per process 15.2Mb
by slot
by node
efficiency
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
2
4
6
8
10
12
14
processors

Test con una taglia totale di 30Mb

# processors	Total file size	Tyme by slot (seconds)	Time by node (seconds)	Speed up by slot	Speed up by node
1	30	8,87	8,97	1	1
2	60	9,33	9,22	0,951	0,973
3	90	9,52	9,53	0,932	0,941
4	120	10,95	10,94	0,81	0,82
5	150	12,38	12,31	0,716	0,729
6	180	13,73	13,75	0,646	0,652
7	210	15,16	15,14	0,585	0,592
8	240	16,55	16,52	0,536	0,543
9	270	18,09	17,83	0,49	0,503
10	300	19,25	19,45	0,461	0,461
11	330	20,71	20,9	0,428	0,429
12	360	22,15	22,18	0,4	0,404
13	390	23,95	23,41	0,37	0,383
14	420	24,73	24,98	0,359	0,359
15	450	26,16	26,11	0,339	0,344



Efficiency



Conclusioni

A seguito di questi test si può dedurre che la parallelizzazione offre vantaggi considerevoli. Nel test di scalabilità forte con dimensione di 48.2Mb si può notare che già con 12 processori il tempo necessario per l'elaborazione risale lentamente, questo è dovuto all'overhead necessario per lo scambio di dati tra i vari processori. Tale comportamento sarebbe più evidente con file di dimensioni maggiori che purtroppo non è stato possibile utilizzare a causa della memoria necessaria al programma per poter funzionare. Con file troppo grandi infatti il programma terminerebbe la memoria, nonostante tutti gli accorgimenti sull'utilizzo della stessa. È tuttavia ancora possibile fare altri miglioramenti su di essa rimuovendo le poche allocazioni statiche di array che sono rimaste e liberandola quando questa non è più necessaria. Per quanto riguarda questo aspetto purtroppo è presente una sorta di "collo di bottiglia" nel senso che anche se un processo client invia i dati al master e subito dopo libera la memora (come viene già fatto) il processo master dovrebbe comunque mantenere in memoria una quantità considerevole di informazioni (e di strutture dati) per poter ordinare le parole e poter scrivere il CSV, per cui nel caso di file molto grandi purtroppo la memoria termina.

Per quanto riguarda il tempo di esecuzione c'è una piccola perdita di tempo dovuta al fatto che il master deve riordinare tutte le parole che vengono ricevute dai client prima alfabeticamente (in modo da poter aggiornare il contatore di parole simili trovate in altri file) e poi per numero di occorrenze (per poterle stampare in ordine decrescente). Nonostante ogni processo client ordini man mano le parole alfabeticamente è necessaria un’altra operazione di ordinamento da parte del master perché la divisione delle parole tra i processi non è fatta per ordine alfabetico (e sarebbe anche troppo complesso, computazionalmente parlando, farlo).