A paraller WordCount using MPI in C

Alessandro Oliviero 0522501083

June 2021

Sommario

WordCount consiste nel contare il numero di parole in uno o più documenti di testo. Il conteggio delle parole può essere utile quando è necessario che un testo non superi un numero specifico di parole. Questa implementazione di WordCount in MPI è stata realizzata durante il corso di "Programmazione Concorrente, Parallela e su Cloud" (PCPC - 0522500102) presso l'Università degli Studi di Salerno.

1 Introduzione

Sarà presentata una versione di map-reduce usando MPI per eseguire il conteggio delle parole su un gran numero di file:

- Ogni nodo legge l'elenco dei file (o una directory), che conterrà i nomi di tutti i file da contare. Il nodo MASTER calcolerà che porzioni di file spettano ad ogni processo. Una volta che un processo ha ricevuto il suo elenco di file da elaborare, dovrebbe quindi leggere in ciascuno di essi ed eseguire un conteggio delle parole, tenendo traccia della frequenza con cui si verifica ogni parola trovata nei file. Chiameremo l'istogramma prodotto l'istogramma locale. Questo è simile alla fase *map* di map-reduce.

- La seconda fase è combinare le frequenze delle parole attraverso i processi. Ad esempio, la parola "gatto" potrebbe essere conteggiata in più processi e dobbiamo sommare tutte queste occorrenze. Questo è simile alla fase *reduce* di map-reduce.

- L'ultima fase consiste nel fare in modo che ciascuno dei processi invii i propri istogrammi locali al processo MASTER. Il processo MASTER ha solo bisogno di raccogliere tutte queste informazioni. É molto probabile che ci siano parole duplicate tra i processi. Il MASTER dovrebbe creare un file in formato CSV con la lista di parole (e il relativo numero) ordinate in modo decrescente in base alla frequenza.

2 Soluzione proposta

L'approccio utilizzato consente di leggere parole da un file, senza vincoli su come questo sia formattato: ad esempio, le parole che contiene non devono essere separate obbligatoriamente dal carattere '\n', o da qualsiasi altro "carattere speciale"; basta semplicemente che ci sia uno o più spazi, segni di punteggiatura e così via. In questo modo è possibile analizzare anche potenzialmente libri, o file di testo di questo tipo:

> *Nunc fermentum, arcu sed iaculis ultrices, enim arcu blandit nibh, bibendum auctor nisi magna ac velit. Phasellus egestas vehicula lacus nec tincidunt. Pellentesque diam metus, vulputate eget faucibus eget, maximus at nulla. Nunc volutpat turpis vel leo sagittis, in condimentum eros aliquet. Aliquam finibus, erat id hendrerit tincidunt, mi sapien hendrerit mauris, vehicula malesuada turpis turpis quis est. Fusce sodales condimentum enim, et placerat metus tempus sed. Phasellus lectus ante, ullamcorper at placerat ac, scelerisque sed dolor. Nam egestas nibh eu risus convallis, et ullamcorper odio fermentum.*

N.B.: Le parole contenute nei file e i nomi dei file devono avere, per implementazione, lunghezza minore o uguale di 100 caratteri.

Bisogna stabilire in che modo viene suddiviso il carico di lavoro tra i vari nodi, poiché di certo non è possibile farlo in base al numero dei file: è molto probabile che si verifichi uno sbilanciamento del lavoro tra i processi siccome i file possono essere di grandezze molto diverse tra loro. Una tecnica potrebbe essere quella di suddividere in base al numero di parole, garantendo un buon equilibrio del carico tra gli slave, ma è stata scelta come soluzione la divisione per byte perché la divisione per parole avrebbe generato un maggiore overhead per il conteggio iniziale di esse.

La soluzione proposta consiste brevemente in:

1. Calcolare la somma dei byte di tutti i file da analizzare;
2. Si dividono (equamente) i byte dei file da leggere per ogni processo;
3. Ogni processo calcola il proprio istogramma locale e lo invia al master;
4. Il processo master raggruppa le informazioni ricevute, producendo in output i risultati in un file CSV.

2.1 Step 1 - Scansione dei file

Nella funzione *fileScan()* vengono scansionati i file nella directory definita, e vengono calcolati i byte totali da analizzare:

- la directory viene aperta;
- usando *fseek()* ci si posiziona alla fine del file;
- usando *ftell()* si prende la posizione del puntatore che corrisponde alla dimensione del file.

2.2 Step 2 - Suddivisione del lavoro

Nella funzione *chunkAndCount()* ogni processo calcola i byte da analizzare in base ai file forniti in input nella cartella *../txt/*. Vengono calcolati sia quanti byte ogni processo deve analizzare, sia da quale byte di quale file deve iniziare la lettura e da quale byte di quale file deve terminare. Per esempio:

> Il processo 3 deve analizzare 1000 byte: dal byte 400 del file 5 fino a 150 byte del file 7.

Chiaramente se la divisione dei byte totali da analizzare sul numero di processi non è perfetta, i primi "resto" processori analizzano un byte in più.

2.3 Step 3 - Conteggio delle parole

Nella funzione *wordCount()* ogni processo analizza la sua porzione di file e restituisce un array della struttura **Word**, costituito dalla parola e dal numero delle sue occorrenze. Il funzionamento di base dell'algoritmo è molto semplice: ogni volta che un processo legge una parola dal file controlla se è presente nel suo array locale, se sì ne incrementa il contatore, in caso contrario deve aggiungere un blocco in più all'array, inserire la parola e inizializzare il suo contatore a 1.

Ogni parola viene letta dal file tramite la funzione *getWord()*, che estrae la parola leggendo carattere per carattere, evitando di leggere caratteri non alfanumerici come segni di punteggiatura o spazi bianchi; la parola viene restituita con tutti i caratteri minuscoli per evitare problemi di mancata corrispondenza tra parole identiche. Utilizzando questa tecnica, le parole separate da un segno di punteggiatura, ad esempio un trattino, vengono trattate come due parole diverse.

Poiché le porzioni da analizzare sono state suddivise per byte e non per parole, è possibile che un processo termini la sua computazione all'interno di una parola e di conseguenza un altro processo inizi in mezzo a quella parola. Questa situazione è stata gestita in modo tale che il processo che dovrebbe finire a metà della parola la legge nella sua interezza e la inserisce nel suo array, mentre l'altro processo controlla se all'inizio si trova nel mezzo di una parola, se sì lo ignora e passa alla parola successiva, altrimenti la elabora.

2.4 Step 4 - Raggruppamento delle informazioni

Ogni volta che un processo termina l'elaborazione dei propri dati, esegue un *MPI_Gather()* per comunicare al master la dimensione del proprio array locale. Viene creato il tipo MPI per la struttura **Word** e poi, tramite *MPI_Gatherv()*, tutti i dati degli slave vengono raccolti dal master. Le stesse parole ricevute da processi diversi vengono raggruppate e infine l'array risultante viene ordinato in ordine decrescente delle frequenze delle parole e viene creato un file CSV per l'output. Il file viene chiamato *ResultsParallel.csv*.

3 Analisi del codice

Di seguito viene riportata un'analisi dei punti più salienti del codice, prima dell'implementazione concorrente su MPI, poi della soluzione sequenziale, per andare a verificare la correttezza dei risultati prodotti nel Capitolo 5.

3.1 Soluzione parallela

3.1.1 Strutture utilizzate

```
1 typedef struct{
2   char name[100];
```

```
  int size;
}fileStats;
```

La struttura **fileStats** viene utilizzata per tenere traccia delle informazioni relative ai file, quali nome e dimensione espressa in byte, quest'ultima verrà utilizzata per dividere l'area di calcolo dei processori.

```
typedef struct{
  char parola[100];
  int count;
}Word;
```

La struttura **Word** viene utilizzata da ogni processore per tenere traccia delle informazioni (parola e frequenza) relative alle parole lette o che sta leggendo. Un array di questa struttura all'interno dei processi rappresenta l'istogramma locale delle parole.

3.1.2 Conteggio byte dei file

```
#define FOLDER "txt"

fileStats* fileScan(int* countFiles, int* totalByte){
    DIR *directory;
    FILE *file;
    struct dirent *Dirent;
    int *fileByte;
    fileStats* myFiles;
    directory = opendir(FOLDER);
    if(directory){
        while(Dirent=readdir(directory))
            *countFiles = *countFiles + 1;
        seekdir(directory, 0);
        myFiles = (fileStats*) malloc(sizeof(fileStats) * (*
    countFiles));
        *countFiles = 0;
        while(Dirent=readdir(directory)){
            char filepath[100] = FOLDER;
            strcat(filepath, "/");
            strcat(filepath, Dirent->d_name);
            if(Dirent->d_type==8){    //regoular file
                file = fopen(filepath, "r");
                strcpy(myFiles[*countFiles].name,Dirent->d_name);
                if(file){
                    fseek(file, 0L, SEEK_END);
                    myFiles[*countFiles].size = ftell(file);
                    *totalByte = *totalByte + myFiles[*countFiles].
    size;
                    *countFiles = *countFiles + 1;
                }
                fclose(file);
            }
        }
    }
    else
        printf("Directory non leggibile\n");
    closedir(directory);
```

```
    return myFiles;
}
```

Questa funzione restituisce un array della struttura *fileStats* appena vista. Il suo funzionamento è semplice:

- la directory definita in *FOLDER* viene aperta;
- si aprono tutti i regolar file all'interno (d_type=8)
- usando *fseek(file, 0L, SEEK_END)* ci si posiziona alla fine del file;
- usando *ftell(file)* si prende la posizione del puntatore che corrisponde alla dimensione del file;
- si salvano nome e dimensione nell'array.

3.1.3 Suddivisione input per byte

```
Word* chunkAndCount(int myrank, int totalByte, fileStats* myFiles,
    int countFiles, int p, int *size){
    int resto = totalByte % p;
    int byteProcess = totalByte / p;
    //da che byte parto (riferito ai byte totali da elaborare)
    int inizio = (myrank<resto) ? (byteProcess + 1) * myrank :
    resto + (byteProcess * myrank);
    //a che byte arrivo (riferito ai byte totali da elaborare)
    int fine = (myrank<resto) ? (byteProcess + 1) * (myrank + 1) :
    resto + (byteProcess * (myrank + 1));
    int inizioFile = 0, fineFile = 0; //da che file parto a che
    file arrivo
    for(int i = 0; i<countFiles; i++){
        if(inizio<=myFiles[i].size){
            inizioFile = i;
            break;
        }else
            inizio -= myFiles[i].size;
            //da che byte parto (riferito al file)
    }
    for(int i = 0; i<countFiles; i++){
        if(fine<=myFiles[i].size){
            fineFile = i;
            break;
        }else
            fine -= myFiles[i].size;
            //a che byte arrivo (riferito al file)
    }

    return wordCount(inizio, inizioFile, fineFile, resto,
    byteProcess, myFiles, myrank, size);
}
```

All'interno di questa funzione, ogni processo si calcola quale porzione dei file deve analizzare, richiamando infine la funzione di *wordCount()* passando come

parametri i valori ottenuti. Anziché far calcolare questi valori solo al master, comunicando in seguito i risultati a tutti, è stato preferito sfruttare a pieno il parallelismo; in questo modo la funzione *chunkAndCount()* calcola i valori necessari solo al processo che la esegue, dato che verrà eseguita concorrentemente da tutti.

3.1.4 Estrazione parola

```
int getWord(FILE *file, char* str){
    int i=0;
    char c;
    do{
        c = fgetc(file);
        if(isalnum(c))
            str[i++] = tolower(c);
    }while(isalnum(c));
    str[i] = '\0';
    return i;
}
```

Questa funzione legge una parola da *file* e la inserisce in *str*; la parola viene letta carattere per carattere dal file e vengono salvati solo i caratteri alfanumerici, ignorando gli eventuali segni di punteggiatura. In output viene restituito il numero di caratteri alfanumerici letti, nonché la lunghezza della parola che si troverà in *str*; se il primo carattere letto non è alfanumerico *str* sarà vuota e verrà restituito 0.

3.1.5 IndexOf

```
int indexOf(Word *words, char* str, int size){
    for(int i=0; i<size; i++)
        if(!strcmp(str, words[i].parola))
            return i;
    return -1;
}
```

Questa semplice funzione serve a controllare se la parola presa in input, *str*, è presente nell'array **words*, se sì restituisce l'indice, altrimenti -1.

3.1.6 Conteggio delle parole

```
Word* wordCount(int inizio, int inizioFile, int fineFile, int resto
    , int byteProcess, fileStats* myFiles, int myrank, int *size){
    DIR *directory;
    FILE *file;
    struct dirent *Dirent;
    int countByte = 0;
    Word* words;
    words = (Word*) malloc(sizeof(Word));
    byteProcess = (myrank<resto) ? byteProcess + 1 : byteProcess;

    directory = opendir(FOLDER);
```

```
    if(directory){
        for(int iFile = inizioFile; iFile <= fineFile; iFile++){
            char filepath[100] = FOLDER;
            strcat(filepath, "/");
            strcat(filepath, myFiles[iFile].name);
            file = fopen(filepath, "r");
            if(file){
                if(iFile == inizioFile)
                fseek(file, inizio, SEEK_SET);
                while(!feof(file)){
                if(countByte<byteProcess){
                    char str[100]={};
                    int prima = ftell(file);
                    if(!getWord(file, str)){
                        int dopo = ftell(file);
                        countByte+=(dopo-prima);
                        continue;
                    }
                    int dopo = ftell(file);
                    int salto = 0;
                    if(iFile == inizioFile && prima == inizio &&
    inizio!=0){
                        fseek(file, inizio-1, SEEK_SET);
                        char c;
                        c = fgetc(file);
                        if(!isspace(c))
                            salto = 1;
                        fseek(file, dopo, SEEK_SET);
                    }
                    countByte+=(dopo-prima);
                    if(salto)
                        continue;

                    int i = indexOf(words, str, *size);
                    if(i!=-1)
                        words[i].count++;
                    else{
                        *size=*size+1;
                        words = (Word*) realloc(words, (*size) *
    sizeof(Word));
                        strcpy(words[*size-1].parola, str);
                        words[*size-1].count = 1;
                    }
                }else
                    break;
                }
            }
            fclose(file);
        }
        return words;
    }
}
```

Questa funzione viene eseguita da tutti i processori ed il codice al suo interno serve a calcolare l'istogramma locale delle parole rappresentato con l'array della struttura **Word**, **words*. Dopo aver aperto la directory definita in precedenza, si itera sul numero dei file che spettano al processo in questione, e dopodiché

tramite il *while* della riga 20 e il successivo *if*, si controllano sia se si raggiungono i byte da leggere ogni volta, sia se si è arrivati alla fine del file. Bisogna, quindi, tener conto dei byte letti ogni volta; questo viene fatto semplicemente eseguendo due chiamate alla funzione *ftell(file)*, che restituisce la posizione del puntatore di lettura all'interno del file, prima e dopo aver letto la parola tramite la funzione *getWord()* vista nella Sezione 3.1.4, e aggiornando il contatore alla riga 39.

Siccome quest'implementazione fornita prevede che se una parola si trova a cavallo tra un processo ed un altro, questa venga letta dal primo processo, c'è la necessità di controllare se un nodo inizia a leggere nel mezzo di una parola la quale è stata già considerata dal nodo precedente. Il blocco di codice dell'if alle righe 31-38 serve a verificare proprio questa situazione. Se il nodo in questione sta leggendo la sua prima parola *(prima=inizio)* del primo file che gli è stato assegnato *(iFile=inizioFile)*, e non mi trovo all'inizio del file *(inizio!=0)*, allora sposta il puntatore un byte (carattere) indietro facendo in modo di leggere il carattere precedente a quello che gli spetterebbe realmente. Ora, se questo carattere non è uno spazio, e quindi fa parte di una parola, bisogna saltare la parola poiché è già stata letta dal processo precedente, altrimenti bisogna leggerla. Il processo master (rank 0) non eseguirà mai questo blocco siccome il suo *inizio* è pari a 0. Il "salto" della parola verrà comunque effettuato dopo aver aggiornato il contatore dei byte letti.

Una volta letta la parola e salvata in *str*, bisogna aggiornare **words*; per farlo viene richiamata la funzione *indexOf()* vista nella Sezione 3.1.5. Se quest'ultima restituisce un valore diverso da -1, significa che in **words* è già presenta la parola e quindi bisogna solo incrementare il contatore delle occorrenze relativo, altrimenti sgnifica che in **words* non è presente la parola e quindi bisogna aumentare la dimensione dell'array, inserire la parola appena letta come ultimo elemento ed infine inizializzare il contatore ad 1.

3.1.7 Reduce degli istogrammi

```
1  MPI_Gather(&size, 1, MPI_INT, sizeRecv, 1, MPI_INT, 0,
       MPI_COMM_WORLD);
2      if(myrank==0)
3          for(int i=0; i<p; i++)
4              sizeTotal += sizeRecv[i];
5
6      MPI_Datatype wordtype, oldtypes[2], parolatype;
7      int blockcounts[2];
8      MPI_Aint offsets[2], lb, extent;
9      MPI_Type_contiguous(100, MPI_CHAR, &parolatype);
10     MPI_Type_commit(&parolatype);
11     offsets[0] = 0;
12     oldtypes[0] = parolatype;
13     blockcounts[0] = 1;
14     MPI_Type_get_extent(parolatype, &lb, &extent);
15     offsets[1] = extent;
16     oldtypes[1] = MPI_INT;
17     blockcounts[1] = 1;
18     MPI_Type_create_struct(2, blockcounts, offsets, oldtypes, &
       wordtype);
```

```
19      MPI_Type_commit(&wordtype);
20
21      Word *wordsRecv;
22      wordsRecv = (Word *) malloc(sizeTotal * sizeof(Word));
23      int displacements[p];
24      if(myrank==0)
25          for(int i=0; i<p; i++)
26              displacements[i] = (i==0) ? 0 : displacements[i-1] +
        sizeRecv[i-1];
27
28      MPI_Gatherv(words, size, wordtype, &wordsRecv[0], sizeRecv,
        displacements, wordtype, 0, MPI_COMM_WORLD);
29      free(words);
30      Word *wordsTotal;
31      int totalWords = 0, dim = 0;
32      if(myrank==0){
33          wordsTotal = (Word *) malloc(sizeof(Word));
34          int index = 0;
35          for (int i=0; i<sizeTotal; i++){
36              totalWords+=wordsRecv[i].count;
37              index = indexOf(wordsTotal, wordsRecv[i].parola, dim);

38              if(index!=-1)
39                  wordsTotal[index].count+=wordsRecv[i].count;
40              else{
41                  dim++;
42                  wordsTotal = (Word*) realloc(wordsTotal, dim *
        sizeof(Word));
43                  strcpy(wordsTotal[dim-1].parola, wordsRecv[i].
        parola);
44                  wordsTotal[dim-1].count = wordsRecv[i].count;
45              }
46          }
47          free(wordsRecv);
48          printf("Total words: %d - different words: %d\n",totalWords
        ,dim);
49      }
50  }
```

Una volta che un nodo ha terminato il suo conteggio può comunicare tramite *MPI_Gather()* al master la dimensione del suo istogramma e il master calcola la dimensione totale, che corrisponde al numero di elementi totali che riceverà in seguito.

Le righe 6-19 servono a definire la struttura **Word**, vista nella Sezione 3.1.1, su MPI: viene prima creato il tipo contiguo per i 100 caratteri che formano la parola, dopodiché insieme all'intero che rappresenta in numero di occorrenze, viene definito il tipo *wordtype*. A questo punto si setta l'array *displacements*, dove ogni elemento di indice i rappresenta lo spostamento relativo al buffer di ricezione a cui posizionare i dati in arrivo dal processo i. Si effettua la *MPI_Gatherv()* in modo tale che il master abbia a disposizione tutti gli istogrammi prodotti dai nodi, messi in sequenza.

Il processo master dovrà aggregare i dati ricevuti, siccome è probabile che più processi abbiano una o più parola uguali tra loro con diversi conteggi. In

modo analogo al *wordCount()* della Sezione 3.1.6, il master scrive l'istogramma finale in **wordsTotal*.

3.1.8 Output del risultato

```
int compare(const void * a, const void * b){
    return ((Word*)b)->count - ((Word*)a)->count;
}
```

Questa è la funzione di confronto per gli elementi dell'array di **Word**; serve per ordinare in modo decrescente in base alla frequenza, la lista di parole. Viene presa come parametro della funzione di libreria *qsort()*.

```
    if(myrank==0){
        qsort(wordsTotal,dim,sizeof(Word),compare);
        FILE *fpt;
        fpt = fopen("ResultsParallel.csv", "w+");
        fprintf(fpt,"Word, Count\n");
        for(int i=0; i<dim; i++)
            fprintf(fpt,"%s, %d\n",wordsTotal[i].parola,wordsTotal[
    i].count);
        ...
    }
```

Nella fase finale del programma è previsto l'ordinamento del risultato con relativa stampa in un file CSV. Questo Snippet di codice serve a proprio ad effettuare queste operazioni sull'array **wordsTotal* definito nella Sezione 3.1.7, producendo il file *ResultsParallel.csv* in output.

3.2 Soluzione sequenziale

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <dirent.h>
#include <ctype.h>
#include <sys/time.h>

#define FOLDER "txt"

typedef struct{
    char parola[100];
    int count;
}Word;

int indexOf(Word *words, char* str, int size);
int getWord(FILE *file, char* str);
int compare(const void * a, const void * b);

int main(int argc, char** argv) {
    struct timeval start;
    struct timeval end;
```

```
    float elapsed;
    gettimeofday(&start, 0);
    DIR *directory;
    FILE *file;
    struct dirent *Dirent;
    Word* words;
    words = (Word*) malloc(sizeof(Word));
    int size = 0, totalWords = 0;

    directory = opendir(FOLDER);
    if(directory){
        while(Dirent=readdir(directory)){
            char filepath[100] = FOLDER;
            strcat(filepath, "/");
            strcat(filepath, Dirent->d_name);
            if(Dirent->d_type==8){
                file = fopen(filepath, "r");
                printf("Filename: %s\n",Dirent->d_name);
                if(file){
                    while(!feof(file)){
                        char str[100]={};
                        if(!getWord(file, str))
                            continue;
                        totalWords++;
                        int index = 0;
                        int i = indexOf(words, str, size);
                        if(i!=-1)
                            words[i].count++;
                        else{
                            size++;
                            words = (Word*) realloc(words, (size) *
 sizeof(Word));
                            strcpy(words[size-1].parola, str);
                            words[size-1].count = 1;
                        }
                    }
                }
                fclose(file);
            }
        }
        printf("Total words: %d - different words: %d\n",totalWords
,size);
    }
    else
        printf("Directory non leggibile\n");
    closedir(directory);

    qsort(words,size,sizeof(Word),compare);
    FILE *fpt;
    fpt = fopen("ResultsSequential.csv", "w+");
    fprintf(fpt,"Word, Count\n");
    for(int i=0; i<size; i++)
        fprintf(fpt,"%s, %d\n",words[i].parola,words[i].count);

    gettimeofday(&end, 0);
    elapsed = (end.tv_sec - start.tv_sec) * 1000.0f + (end.tv_usec
    - start.tv_usec) / 1000.0f;
```

```
78      printf("Code executed in %.2f milliseconds.\n", elapsed);
79
80      free(words);
81      fclose(fpt);
82  }
83
84  int indexOf(Word *words, char* str, int size){
85      for(int i=0; i<size; i++)
86          if(!strcmp(str, words[i].parola))
87              return i;
88      return -1;
89  }
90
91  int getWord(FILE *file, char* str){
92      int i=0;
93      char c;
94      do{
95          c = fgetc(file);
96          if(isalnum(c))
97          str[i++] = tolower(c);
98      }while(isalnum(c));
99      str[i] = '\0';
100     return i;
101 }
102
103 int compare(const void * a, const void * b){
104     return ((Word*)b)->count - ((Word*)a)->count;
105 }
```

Questo è il codice della soluzione sequenziale. Come si può notare, la parte cruciale del programma, ovvero il conteggio delle parole, è pressoché identica all'implementazione parallela.

É stata sviluppata in principio questa soluzione e successivamente elaborata per renderla adatta al calcolo distribuito, aggiungendo la comunicazione e tutti gli altri comportamenti che generano overhead. Infatti, questa soluzione viene utilizzata non solo per andare a controllare la correttezza dei risultati, ma anche analizzare l'effettivo guadagno in termini di prestazioni ed efficienza sfruttando l'implementazione concorrente.

4 Istruzioni per l'esecuzione

Di seguito verranno riportate le istruzioni per l'esecuzione in locale del programma che sfrutta MPI e del programma sequenziale, dopo aver clonato la **repository** su github all'url: `https://github.com/aoliviero7/wordcount-mpi`.

4.1 Esecuzione parallela

All'interno della directory *wordcount-mpi* si può lanciare il seguente comando:

```
mpirun -np NUMBER_OF_PROCESSORS wordcount
```

Aggiungere `--allow-run-as-root` e `--oversubscribe` se necessario, in questo modo:

```
mpirun -np --allow-run-as-root --oversubscribe NUMBER_OF_PROCESSORS
wordcount
```

In questo modo è possibile avviare il programma con l'utilizzo di *NUMBER_OF_PROCESSORS* processori e contare parallelamente le frequenze delle parole dei file dentro la cartella *wordcount-mpi/txt*; se si vogliono provare file *.txt* di input differenti, si possono sostituire quelli già presenti.

4.2 Esecuzione sequenziale

All'interno della directory *wordcount-mpi* è presente un file chiamato *sequential.c* che permette di eseguire lo stesso algoritmo implementato nella soluzione descritta, ma sequenzialmente e senza tutte quelle funzionalità che generano overhead quali: il byte count di ogni file, il calcolo della porzione dei file da analizzare, il raggruppamento delle informazioni da parte del master e in generale tutta la comunicazione. All'interno della directory *wordcount-mpi* lancia il seguente comando:

```
sequential
```

In questo modo è possibile avviare il programma e contare sequenzialmente le frequenze delle parole dei file dentro la cartella *wordcount-mpi/txt*; anche in questo caso, se si vogliono provare file *.txt* di input differenti, si possono sostituire quelli già presenti nella directory.

5 Correttezza dei risultati

Per analizzare i risultati prodotti dalle due soluzioni, parallela e sequenziale, sono state eseguite entrambe su una vasta gamma di input di diverse dimensioni suddivise in uno o più file. Per una semplice comprensione viene riportato un breve file di input con relativo output.

Input:

> *Nunc fermentum, arcu sed iaculis ultrices, enim arcu blandit nibh, bibendum auctor nisi magna ac velit. Phasellus egestas vehicula lacus nec tincidunt. Pellentesque diam metus, vulputate eget faucibus eget, maximus at nulla. Nunc volutpat turpis vel leo sagittis, in condimentum eros aliquet. Aliquam finibus, erat id hendrerit tincidunt, mi sapien hendrerit mauris, vehicula malesuada turpis turpis quis est. Fusce sodales condimentum enim, et placerat metus tempus sed. Phasellus lectus ante, ullamcorper at placerat ac, scelerisque sed dolor. Nam egestas nibh eu risus convallis, et ullamcorper odio fermentum.*

Output:

```
Word, Count
turpis, 3
nunc, 2
arcu, 2
enim, 2
nibh, 2
ac, 2
phasellus, 2
egestas, 2
vehicula, 2
tincidunt, 2
metus, 2
at, 2
hendrerit, 2
et, 2
placerat, 2
ullamcorper, 2
fermentum, 1
sed, 1
iaculis, 1
...
```

6 Benchmark

Dopo aver analizzato la correttezza della soluzione proposta, si è passati alla fase di test e benchmarking su cluster. É stato utilizzato Amazon Web Services (AWS) che fornisce servizi di cloud computing su una piattaforma on-demand. Per questa fase sono state utilizzate due principali metriche per la valutazione delle prestazioni, quali: ***Speedup*** ed ***Efficiency***.

Lo *Speedup* serve ad indicare l'incremento prestazionale tra l'esecuzione sequenziale e l'esecuzione parallela, a parità di input.

$$Speedup = \frac{TIME_{sequential}}{TIME_{parallel}} \tag{1}$$

L' *Efficiency* è una normalizzazione dello *Speedup* e serve ad indicare quanto si avvicinano i tempi dell'esecuzione sequenziale e di quella parallela, sempre a parità di input.

$$Efficiency = \frac{TIME_{sequential}}{N * TIME_{parallel}} \tag{2}$$

Per lo *Speedup*, il valore massimo raggiungibile è N con N processori, mentre l' *Efficiency* massima è sempre 1.

Dopo aver definito l'architettura utilizzata verranno esposti i risultati in termini di:

- *Strong Scalability*;
- *Weak Scalability.*

Per un'affidabilità maggiore dei risultati, ogni test è stato eseguito 10 volte e, dopodiché, è stata fatta la media dei 10 risultati ottenuti.

6.1 Architettura

I benchmark sono stati effettuati su un cluster in AWS di 8 istanze *t2.small*, ciascuna con:

- 1 vCPU
- 2 GiB di RAM
- Ubuntu Linux 18.04 LTS Server Edition - ami-0747bdcabd34c712a
- 8 GiB storage EBS

Durante la fase di testing tutte le istanze erano prive di processi attivi, se non quelli necessari al sistema operativo e all'esecuzione del programma.

6.2 Strong Scalability

Nella Strong Scalability viene calcolato il tempo di esecuzione del programma con taglia dell'input costante, ma al variare del numero di processori.

Sia T_k il tempo di esecuzione del programma su k processori ed N il numero di processori, vengono calcolati anche:

- *Strong Scalability Speedup*:

$$SS_{Speedup} = \frac{T_1}{T_N} \tag{3}$$

- *Strong Scalability Efficiency*:

$$SS_{Efficiency} = \frac{T_1}{(N * T_N)} \tag{4}$$

L'input, che resterà costante per tutte le esecuzioni, è composto da 6,38 MB di dati suddivisi nei seguenti file:

- *25k_1.txt* (175 KB): un file di circa 25000 parole casuali in inglese con ripetizioni ammesse;
- *The Hitchhikers Guide to the Galaxy .txt* (280 KB): libro di Douglas Adams;
- *HP - The Chamber of Secrets.txt* (539 KB): il secondo capitolo della saga di Harry Potter;
- *HP - The Goblet of Fire.txt* (1,16 MB): il quarto capitolo della saga di Harry Potter;
- *Bible_KJV.txt* (4,24 MB): The King James Bible.

Di seguito vengono riportati i risultati ottenuti.

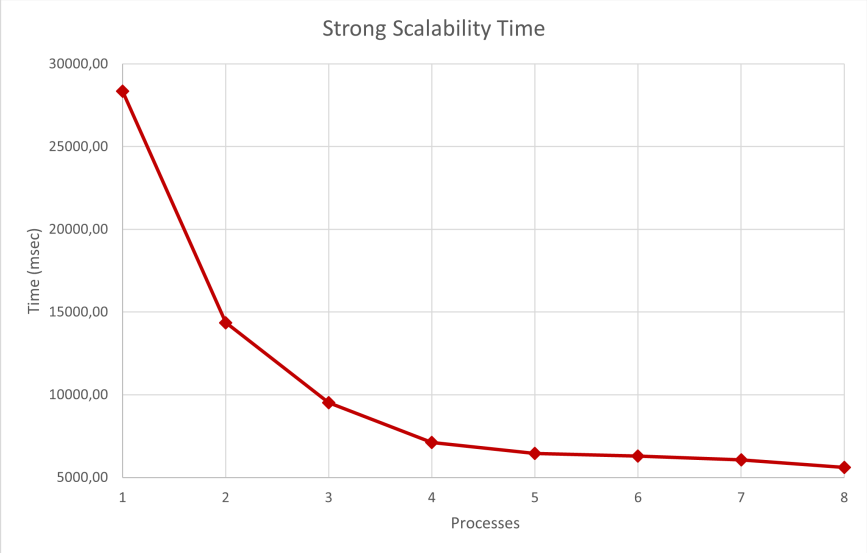

Figura 1: Tempi di esecuzione[1]

Processes	Time (msec)
1	28351,19
2	14359,45
3	9523,15
4	7125,02
5	6450,92
6	6302,86
7	6065,21
8	5597,24

Tabella 1: Tempi di esecuzione[1]

La tabella e il grafico mostrano come i tempi di esecuzione[1] diminuiscono all'aumentare dei processi. Il tempo di esecuzione scende in maniera inversamente proporzionale all'aumentare dei processori coinvolti; questo fino a 4 processi, dopodiché diminuisce in modo più lineare. Questo comportamento lo si può evidenziare maggiormente con i grafici relativi allo *Speedup* e all'*Efficiency*.

[1]Calcolati su una media di 10 esecuzioni per ogni processo

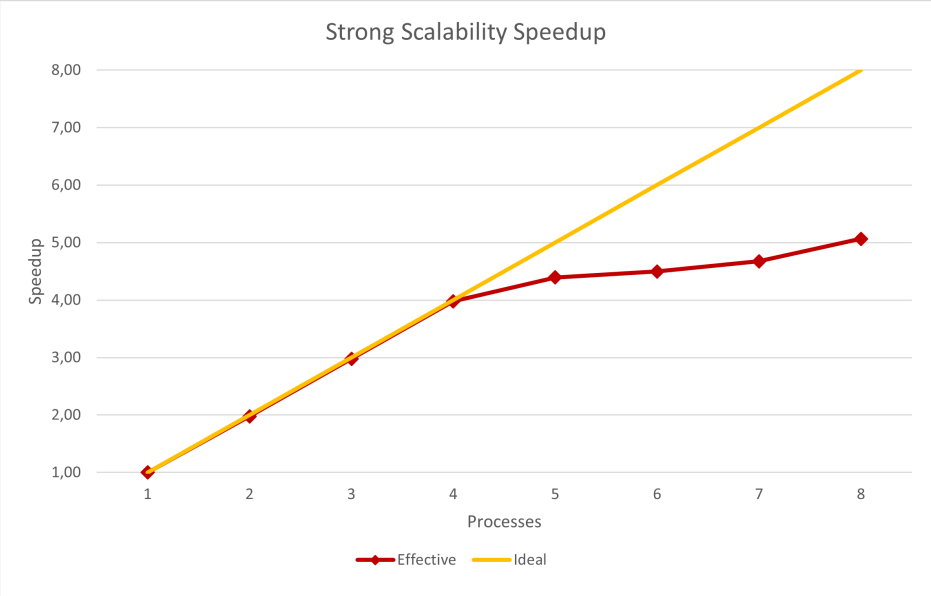

Figura 2: Speedup

Processes	Speedup
1	1,00
2	1,97
3	2,98
4	3,98
5	4,39
6	4,50
7	4,67
8	5,07

Tabella 2: Speedup

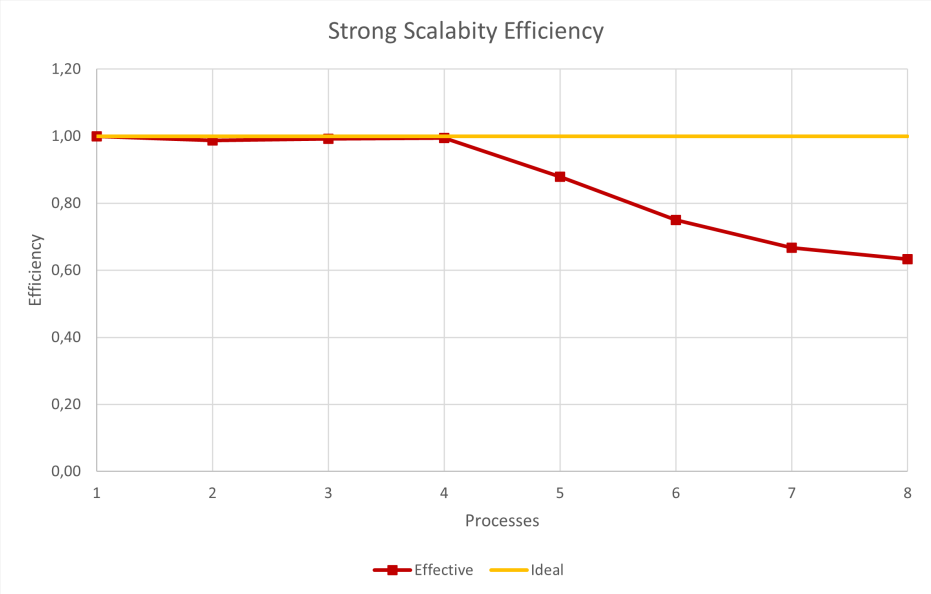

Figura 3: Efficiency

Processes	Efficiency
1	1,00
2	0,99
3	0,99
4	0,99
5	0,88
6	0,75
7	0,67
8	0,63

Tabella 3: Efficiency

Come detto precedentemente sia lo *Speedup* che l'*Efficiency* sono ottimi fino a 4 processori utilizzati, dopodiché le prestazioni migliorano comunque ma in modo meno evidente, raggiungendo nel caso peggiore un'efficienza del 63% e un valore di Speedup pari a 5,07/8.

6.3 Weak Scalability

Nella Weak Scalability viene calcolato il tempo di esecuzione del programma con la taglia dell'input che cresce proporzionalmente al numero di nodi del cluster.

Sia T_k il tempo di esecuzione del programma su k processi, viene calcolata anche la *Weak Scalability Efficiency*:

$$WS_{Efficiency} = \frac{T_1}{T_N} \tag{5}$$

L'input sarà composto da un file di questo tipo per ogni processo coinvolto:

- *25k_N.txt* (175 KB): un file di circa 25000 parole casuali in inglese con ripetizioni ammesse.

Processes	1	2	3	4	5	6	7	8
Words	25k	50k	75k	100k	125k	150k	175k	200k

Tabella 4: Input per la Weak Scalability

Di seguito vengono riportati i risultati ottenuti.

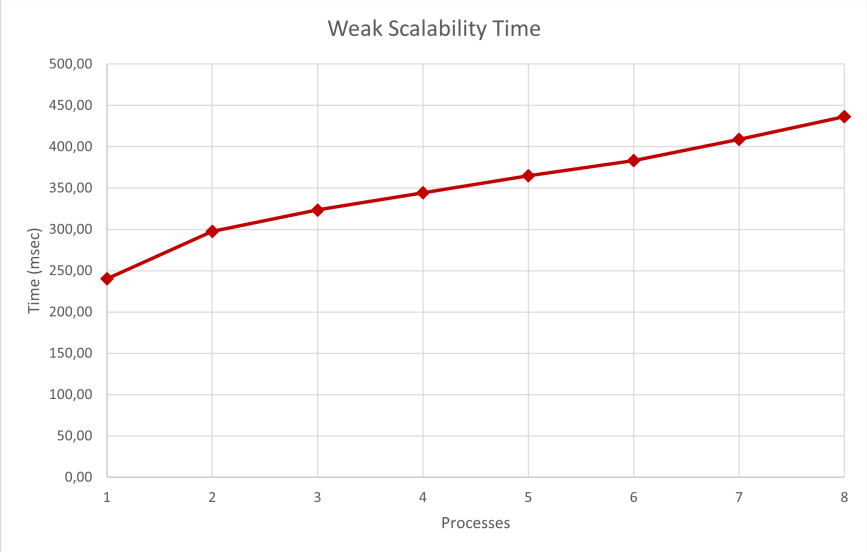

Figura 4: Tempi di esecuzione

Processes	Time (msec)
1	240,18
2	297,55
3	323,41
4	344,06
5	365,08
6	383,19
7	409,07
8	436,40

Tabella 5: Tempi di esecuzione

Come si può vedere dalla Figura 4, il tempo di esecuzione cresce leggermente, ma in modo costante all'aumentare dei processori; questo è dovuto al costo della comunicazione tra i vari nodi del cluster. Infatti, si può notare che lo scarto di tempo più grande lo si ottiene passando da 1 (quindi comunicazione nulla tra processori diversi) a 2 processori coinvolti.

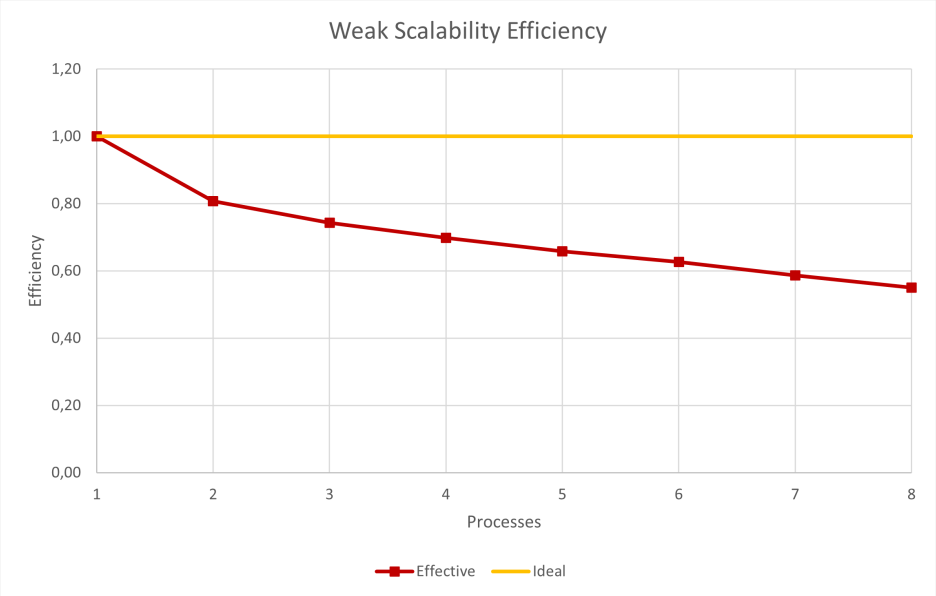

Figura 5: Efficiency

Processes	Efficiency
1	1,00
2	0,81
3	0,74
4	0,70
5	0,66
6	0,63
7	0,59
8	0,55

Tabella 6: Efficiency

Dalla Figura 5, invece, si nota che, a causa dei tempi precedentemente visti, anche l'efficienza ha un comportamento simile: decresce leggermente in modo costante all'aumentare dei processori. Le motivazioni sono le stesse, ed anche qui si evidenzia lo scarto maggiore, di circa del 19%, tra l'utilizzo di 1 e 2 processori.

6.4 Overhead

Per analizzare l'overhead generato dalla soluzione su MPI, vengono confrontati il tempo di esecuzione della soluzione parallela avviata con un processore e il tempo di esecuzione della soluzione sequenziale.

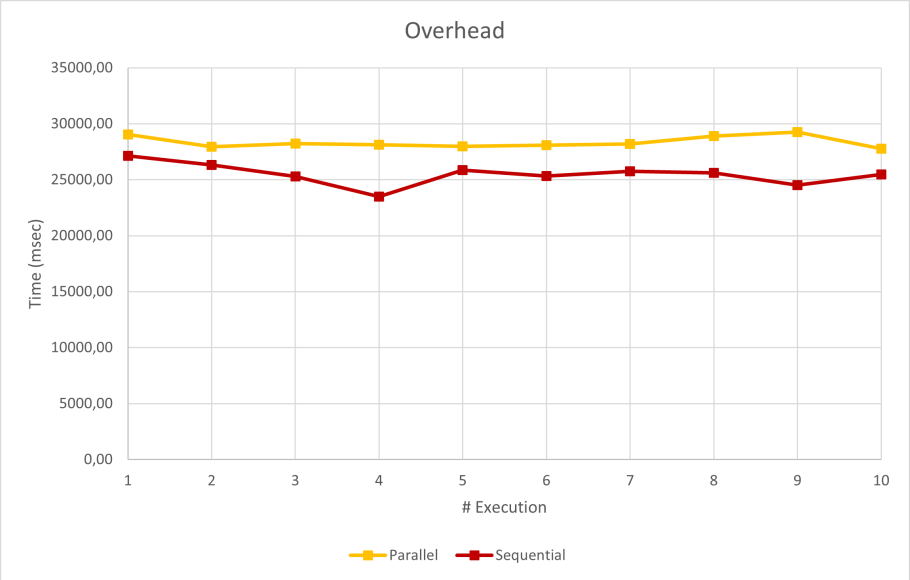

Figura 6: Overhead

Solution	AVG Time (msec)
Parallel	28351,19
Sequential	25474,97

Tabella 7: Overhead

In media calcolando la differenza dei tempi di esecuzione, si nota che l'overhead risultante è di $2876,22$ msec, che corrisponde a circa il 10%.

7 Conclusioni

É stato presentato il problema del Word Count che consiste nel determinare la frequenza delle parole dei file forniti in input. Dopodiché è stata fornita una soluzione parallela al problema utilizzando MPI, analizzando l'approccio utilizzato e il codice dell'implementazione. Infine sono stati analizzati i risultati in termini di Strong e Weak Scalability su un cluster di 8 nodi implementato su AWS. La soluzione ha mostrato risultati ottimi con l'utilizzo di massimo 4 processori; successivamente, da 4 a 8 processori i risultati possono ritenersi comunque discreti.