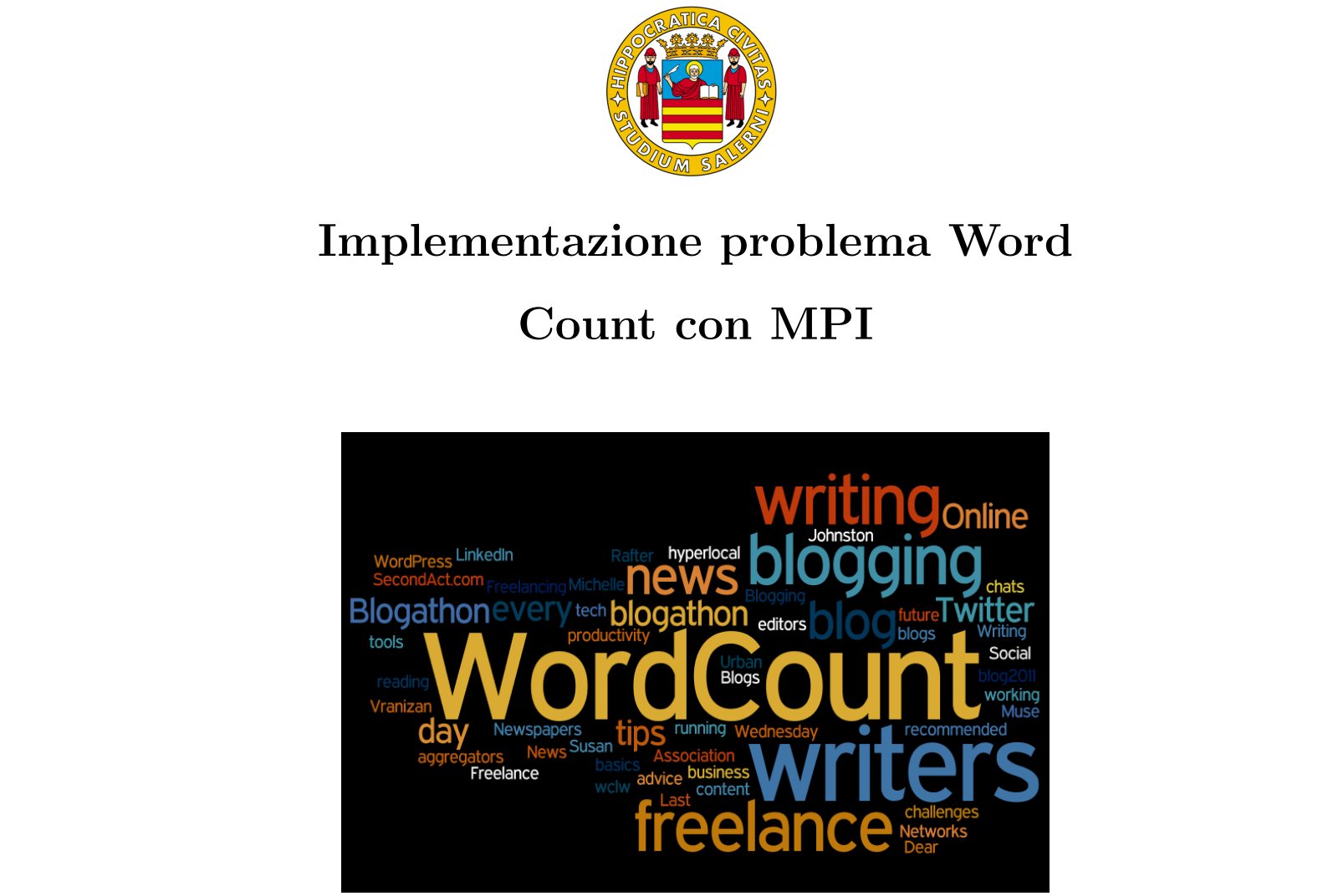


STUDENTE:

Carmine Buondonno

Mat. 05225 01093

DATA : GIUGNO 2021

INDICE

CAPITOLO 1

INTRODUZIONE

Il problema Word Count consiste nel determinare il numero di occorrenze di una parola in un testo dato in input. Generalmente si presenta in applicazioni in cui è necessario mantenere le parole inserite in limiti definiti.Il conteggio delle parole è comunemente usato anche dai traduttori per determinare il prezzo di un lavoro di traduzione, oppure può essere utilizzato anche per calcolare le misure di leggibilità e per misurare la velocità di digitazione e lettura. Anche se sembra semplice nella sua definizione, il problema Word Count rappresenta una sfida nel campo della programmazione distribuita a causa dell'enorme taglia dei problemi che possono presentarsi: infatti possono esserci situazioni in cui i documenti da esaminare hanno dimensioni anche di 100 TB, dove un'esecuzione sequenziale può arrivare a richiedere mesi se non anni di computazione. Quindi bisogna pensare a Word Count come un problema di programmazione distribuita, in cui la distribuzione dell'input tra i nodi di un cluster può significativamente diminuire il tempo necessario. Di solito quando si risolve il problema Word Count con la programmazione distribuita utilizzando un cluster si effettuano i seguenti passaggi:

- Il nodo MASTER legge l'elenco dei file dove bisogna contare le parole in modo da assegnare il lavoro a tutti gli altri nodi. Quindi ciascuno dei

nodi dovrebbe ricevere la propria parte di lavoro dal MASTER. Una volta che un processo ha ricevuto il suo elenco di file da elaborare, dovrebbe quindi leggere in ciascuno dei file ed eseguire il conteggio delle parole, tenendo traccia delle frequenze. Ogni nodo ovviamente dovrà utilizzare una stuttura dati in modo da mantenere in memoria le occorenze.

- La seconda fase prevede che ciascuno dei processi invii la propria stuttura al nodo master. Il MASTER dovrà quindi raccogliere tutte queste informazioni.

- L'ultima fase consiste nel combinare il lavoro svolto da ogni nodo del cluster. Ad esempio una parola potrebbe essere conteggiata da più processi bisgona quindi sommare tutte queste occorrenze e salvare poi il lavoro.

CAPITOLO 2

SOLUZIONE PROPOSTA

Viene proposta una soluzione in un ambiente distribuito, verrà quindi descritto di seguito un programma scritto in C che utilizza MPI per la comunicazione. MPI è di fatto lo standard per la comunicazione tra nodi appartenenti a un cluster che eseguono un programma parallelo. Inoltre ogni nodo del cluster ha accesso agli stessi dati. La soluzione proposta segue un paradigma simile a quello di Map Reduce, infatti come spiegato nell'introduzione ci sarà un prima fase di Map dove ogni nodo elabora la propria parte di input, dopodichè ci sarà la fase di Reduce, dove i risultati di ogni nodo vengono raggruppati. Nel capitolo riguardante i Benchmark verrà derscritto in modo accurato l'architettura del cluster utilizzata, adesso passiamo alla descrizione della soluzione e quindi del codice.

2.1 Partizionamento del lavoro

Listing 2.1: Partizionamento del lavoro

```
1 int main(int argc, char**argv) {
2
3   //VARIABILI PER TEMPO DI ESECUZIONE
4   double start,end;
```

```
5
6
7   //VARIABILI E INIZIALIZZAZIONE MPI//
8   MPI_Init(NULL,NULL);
9   int myrank,p;
10  MPI_Status status;
11  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
12  MPI_Comm_size(MPI_COMM_WORLD, &p);
13  start = MPI_Wtime();
14
15
16
17  DIR* dir;
18  struct dirent* Dirent;
19  FILE* file;
20  dir = opendir("file");
21  int sommabyte = 0;
22  fileS* fileDaLeggere;
23  int count = 0;
24  int inizioByte, fineByte;
25  int indiceInizioFile;
26
27  int size = 0;
28  char** parola;
29  int* countParola;
30
31  countParola = (int*)malloc(sizeof(int));// ALLOCAZIONE
     DINAMICA PER COUNT_PAROLA E LA MATRICE PAROLA
32  parola = (char**)malloc(sizeof(char*));
33  parola[0] = (char*)malloc(500*sizeof(char));
34
35  if(dir!=NULL) {   //CONTROLLO PER DIRECTORY VUOTA
36
37    while((Dirent=readdir(dir))!=NULL) {
38      count++;
39      }
40    fileDaLeggere = (fileS*)malloc(sizeof(fileS)*count);
```

```
41
42    seekdir(dir,0);
43    count = 0;
44    while((Dirent=readdir(dir))!=NULL) {
45      if(Dirent->d_type==8) {
46        char stringa[100] ="file/";
47        strcpy(fileDaLeggere[count].nome,stringa);
48        strcat(fileDaLeggere[count].nome,Dirent->d_name);
49        file=fopen(fileDaLeggere[count].nome,"r");
50        if(file!=NULL) {
51          fseek(file,0L,SEEK_END);
52          fileDaLeggere[count].dimensioneByte = ftell(file);
53          count++;
54          fseek(file,0L,SEEK_SET);
55          fclose(file);
56        }
57      }
58
59    }
60
61    for(int i=0;i<count;i++) {
62      if(myrank==0){ printf("byte %s : %d\n",fileDaLeggere[i].
   nome,fileDaLeggere[i].dimensioneByte);fflush(stdout);}
63      sommabyte += fileDaLeggere[i].dimensioneByte;
64    }
65    if(myrank==0){ printf("somma byte : %d\n\n",sommabyte);
   fflush(stdout);}
66
67  }
68
69  closedir(dir);
70
71  //CALCOLO INIZIO_BYTE E FINE_BYTE PER OGNI SINGOLO PROCESSORE
72  int resto = sommabyte%p;
73  if(myrank<resto){
74    inizioByte = myrank*(sommabyte/p+1);
75    fineByte = inizioByte+(sommabyte/p+1);
```

```
76      printf("rank %d inizio : %d fine : %d\n",myrank,inizioByte,
    fineByte);
77      fflush(stdout);
78    }
79    else {
80      inizioByte = myrank*(sommabyte/p)+resto;
81      fineByte = inizioByte+(sommabyte/p);
82      printf("rank %d inizio : %d fine : %d\n",myrank,inizioByte,
    fineByte);
83      fflush(stdout);
84    }
85
86
87    //CALCOLO INDICE_INIZIO_FILE PER OGNI SINGOLO PROCESSORE
88    int sommaFile = fileDaLeggere[0].dimensioneByte;
89    //printf("size filedimension %d\n",count);
90    for(int i = 0;i<count;i++) {
91      if(inizioByte<sommaFile) {
92        indiceInizioFile = i;
93        break;
94      }
95      else {
96        sommaFile = sommaFile+fileDaLeggere[i+1].dimensioneByte;
97      }
98    }
99
100   printf("rank %d Inizio File : %d\n",myrank,indiceInizioFile);
101   fflush(stdout);
```

In questa fase avviene per prima cosa il calcolo in byte della dimensione dell'input. Il calcolo viene effetuato in modo tale da conoscere la dimensione su cui andare a lavorare, in questo modo il lavoro verrà suddiviso in byte. Questo lavoro viene svolto da tutti i nodi del cluster, in pratica ogni nodo si calcola la taglia dell'input totale, dopodichè si calcola la propria porzione di lavoro sempre in byte, su quale file deve iniziare a leggere, e da quale byte deve partire. Si è scelto di far effettuare questo calcolo ad ogni nodo in modo da eliminare

la comunicazione per quanto riguarda il partizionamento, siccome in termini di overhead la comunicazione è più dispendiosa rispetto la computazione, ad esempio un'altra possibile scelta poteva essere di far computare il partizionamento al nodo MASTER e alla fine del lavoro quest'ultimo comunicava tutto agli altri nodi.

2.2 Word Count

Word Count è la funzione che svolge il compito principale del problema, cioè contare le occorrenze delle parole.

Listing 2.2: Prototipo della funzione Word Count

```
void wordCount(int *countParola, char** parola, int* size,int
    inizioByte,int fineByte,int indiceInizioFile,fileS*
    fileDaLeggere,int myrank,int p);
```

Prima di passare al commento della funzione Word Count è meglio commentare alcune funzioni che vengono usate all'interno e che stutture dati sono state utilizzate. Come stuttura dati per salvare le occorrenze si è scelto di utilizzare un array di interi per quanto riguarda le frequenze e una matrice di char per memorizzare le parole. Le due strutture vengono allocate dinamicamente e poi deallocate alla fine del lavoro.

Listing 2.3: Strutture dati

```
countParola = (int*)malloc(sizeof(int));
parola = (char**)malloc(sizeof(char*));
parola[0] = (char*)malloc(500*sizeof(char));
```

Listing 2.4: Deallocazione

```
free(countParola);
for(int i = 0;i<*size;i++) {
  free(parola[i]);
}
free(parola);
```

Per quanto riguarda invece la lettura delle stringhe all'interno dei file si è utilizzata la seguente funzione.

Listing 2.5: Lettura di una stringa

```
char* takeAndCheckString(FILE *file) {
/*  INPUT:
  FILE *file:   PUNTATORE RELATIVO AL FILE CHE BISOGNA LEGGERE
  OUTPUT:
  char*:      STRINGA MODIFICATA*/

  char* newString = (char*)malloc(sizeof(char)*500);
  int i = 0;
  int c = fgetc(file);
  while(isalnum(c)) {
    newString[i] = c;
    c = fgetc(file);
    i++;
    }
  newString[i] = '\0';
  return newString;

}
```

In pratica viene letto un carattere alla volta e scartati i caratteri non alfanumerici di una stringa. Per l'ordinamento, che viene eseguito solo dal Master una volta che ha compattato tutte le occorrenze calcolate dagli n nodi, è stata utilizzata la seguente funzione. L'ordinamento avviene in base al numero di occorrenze in modo decrescente.

Listing 2.6: Ordinamento delle occorrenze

```
void Sort(int* a,char** b, int dim){
/*  INPUT:
  int* a:     ARRAY CHE CONTEINE I CONTEGGI RELEATIVI ALLE
   PAROLE
  char** b:   MATRICE DELLE PAROLE
  int dim:    DIMENSIONE DELLE DUE STRUTTURE
  */
```

```
7
8   int i, j, max, tmp;
9   char temp[500];
10  for(i=0;i<dim-1;i++){
11    max=i;
12    for(j=i+1;j<dim; j++){
13      if(a[j]>a[max])
14                      max=j;
15    }
16  tmp=a[max];
17  strcpy(temp,b[max]);
18        a[max]=a[i];
19        strcpy(b[max],b[i]);
20        a[i]=tmp;
21        strcpy(b[i],temp);
22  }
23 }
```

Infine viene utilizzata una funzione che viene eseguita solo dal Master per salvare tutto il lavoro e quindi le occorrenze delle parole in un file csv.

Listing 2.7: Salvataggio file CSV

```
1  void CSV(int* a,char** b,int dim) {
2  /*  INPUT:
3    int* a:     ARRAY CHE CONTEINE I CONTEGGI RELEATIVI ALLE
       PAROLE
4    char** b:   MATRICE DELLE PAROLE
5    int dim:    DIMENSIONE DELLE DUE STRUTTURE
6    */
7
8    FILE* file;
9    file = fopen("wordCount.csv","w+");
10   fprintf(file,"WORD , COUNT\n");
11   for(int i = 0;i<dim;i++) {
12     fprintf(file,"%s , %d\n",b[i],a[i]);
13   }
14   fclose(file);
15 }
```

Esiste ancora un'altra funzione che viene utilizzata, questa però verrà commentata in seguito con la funzione Word Count che è la seguente.

Listing 2.8: Word Count

```
1  void wordCount(int *countParola, char** parola, int* size,int
       inizioByte,int fineByte,int indiceInizioFile,fileS*
       fileDaLeggere, int myrank, int p) {
2  /*  INPUT:
3    *countParola:   ARRAY PER IL CONTEGGIO DELLE WORD LETTE
4    char** parola:    MATRICE DELLE WORD LETTE
5    int *size:    DIMENSIONE PER COUNT_PAROLA E LA MATRICE PAROLA
       USATA PER L'ALLOCAZIONE DINAMICA
6    int inizioByte    INDICA IL BYTE DI INIZIO DI OGNI PROCESSORE
7    int fineByte    INDICA IL BYTE DI FINE DI OGNI PROCESSORE
8    int indiceInizioFile  INDICA L'INDICE DEL FILE DA CUI IL
       PROCESSORE DEVE PARTIRE PER LEGGERE
9    FileS* fileDaLeggere  STRUTTURA CHE CONTIENE I NOMI DEI FILE
       E I RELATIVI BYTE
10   int myrank    INTERO RELATIVO AL PROCESSO
11   int p     INTERO CHE INDICA IL NUMERO DI PROCESSI */
12
13   int recvcounts[p];
14   int displs[p];
15
16   int partenza = 0;
17   int indice = indiceInizioFile;
18   int byteLetti = 0;
19   int byte = 0;
20   int byteDaLeggere = fineByte - inizioByte;
21   bool endFile = false;
22   FILE* file;
23
24   int backWordControl = 1;
25   while(byteLetti<byteDaLeggere) {    //PRIMO CICLO DI
       CONTROLLO OGNI PROCESSORE LEGGE IN TOTALE UN NUMERO DI BYTE
       <= DI ByteDaLeggere
26    file=fopen(fileDaLeggere[indice].nome,"r");
```

```
27    if(file!=NULL) {
28      //CALCOLO DELLA PARTENZA
29      if(indice==0) { partenza = inizioByte;}
30      if(indice!=indiceInizioFile) { partenza = 0;}
31      if(indice==indiceInizioFile && indice>0) {
32        int s = 0;
33        for(int i = 0;i<indice;i++) { s = s+fileDaLeggere[i].
    dimensioneByte;}
34        partenza = inizioByte - s;
35      }
36      //MI SPOSTO IN PARTENZA
37      fseek(file, partenza, SEEK_SET );
38
39
40      byte = 0;
41      while(byte<byteDaLeggere) { //SECONDO CICLO DI CONTROLLO
    UTILE SICCOME UN PROCESSO POTREBBE DOVER LEGGERE DA PIU'
    FILE
42
43        if(feof(file)) {endFile = true; break;}
44
45        int verifica = 0;
46        int indiceWord = 0;
47        char *stringa;
48
49        //CONTROLLO SE IL PROCESSO DEVE SPOSTARSI INDIETRO
    SICCOME IL PROCESSO P-1 HA SALTATO LA PAROLA DATO CHE SI
    TROVAVA ALLA FINE DEI SUOI BYTE
50        if(backWordControl&&myrank>0) {
51          backWord(file,partenza,myrank);
52          backWordControl = 0;
53        }
54
55        //FUNZIONE CHE LEGGE UNA PAROLA NEL FILE
56        stringa = takeAndCheckString(file);
57
58
```

```
59        //CALCOLO DEI BYTE LETTI
60        if(indice == indiceInizioFile) { byte = ftell(file) -
    partenza; }
61        else { byte = ftell(file) + byteLetti - partenza; }
62
63        //CONTROLLO SE POSSO SALVARE LA PAROLA LETTA E QUINDI
    SE I BYTE DA LEGGERE SONO TERMINATI
64        if(byte-1<=byteDaLeggere && strlen(stringa)>0) {
65
66
67          for(int i=0;i<*size;i++) {
68            if(strcmp(stringa,parola[i])==0) {
69            verifica = 1;
70            indiceWord = i;
71            }
72          }
73
74          if(verifica==1) {
75            countParola[indiceWord]++;
76          }
77          else {
78            *size = *size+1;
79            countParola = (int*)realloc(countParola,sizeof(int)
    * (*size));
80            countParola[*size-1] = 1;
81            parola = (char**)realloc(parola,*size*sizeof(char*)
    );
82            parola[*size-1] = (char*)malloc(500*sizeof(char));
83            strcpy(parola[*size-1],stringa);
84          }
85        }
86
87      }
88      fclose(file);
89      byteLetti = byte;
90      if(endFile) {indice++; endFile = false;}
91    }
```

```
92  }
93
94
95  int charInt = (sizeof(char)*500)+sizeof(int);
96  int bufsiz = *size*charInt;
97
98  char message[bufsiz];
99  int position = 0;
100
101 //EFFETTUO IL PACK DELLE DUE STRUTTURE
102 for(int i = 0;i<*size;i++) {
103   MPI_Pack(parola[i],500,MPI_CHAR,message,bufsiz,&position,
    MPI_COMM_WORLD);
104   MPI_Pack(&countParola[i],1,MPI_INT,message,bufsiz,&position
    ,MPI_COMM_WORLD);
105 }
106
107 //OGNI PROCESSO INVIA AL RANK 0 LA DIMENSIONE DELLA PROPRIA
     STUTTURA
108 MPI_Gather(size, 1, MPI_INT, recvcounts, 1, MPI_INT, 0,
     MPI_COMM_WORLD);
109
110 int bufSizeRecv = 0;
111 //IL RANK 0 SI CALCOLA RECVCOUNTS E DISPLS PER LA GATHERV
112 if(myrank==0) {
113   for(int i = 0;i<p;i++) {
114     recvcounts[i] = recvcounts[i]*charInt;
115     bufSizeRecv = bufSizeRecv +recvcounts[i];
116   }
117   displs[0] = 0;
118       for(int i=1; i<p;i++) {
119         displs[i] = recvcounts[i-1]+displs[i-1];
120       }
121     }
122     char* recvbuf = (char*)malloc(sizeof(char)*bufSizeRecv);
123
124
```

```
125 //ONGI PROCESSO INVIA AL RANK 0 LE DUE STRUTTURE
126 MPI_Gatherv(message, bufsiz, MPI_PACKED, recvbuf, recvcounts,
      displs , MPI_PACKED, 0, MPI_COMM_WORLD);
127
128 free(countParola);
129 for(int i = 0;i<*size;i++) {
130   free(parola[i]);
131 }
132 free(parola);
133
134
135 char** matrixWord = (char**)malloc(sizeof(char*)*(bufSizeRecv
     /charInt));
136 for(int i = 0; i<(bufSizeRecv/charInt);i++) {
137   matrixWord[i] = (char*)malloc(sizeof(char)*500);
138 }
139 int* matrixCount = (int*)malloc(sizeof(int)*(bufSizeRecv/
     charInt));
140 position = 0;
141
142 int dimensione = 0;
143 int* countFinalWord = (int*)malloc(sizeof(int));
144 char** matrixFinalWord = (char**)malloc(sizeof(char*));
145 matrixFinalWord[0] = (char*)malloc(500*sizeof(char));
146
147 int verifica = 0;
148 int indiceWord = 0;
149
150 //IL RANK 0 EFFETTUA L'UNPACK E SALVA LE STRUTTURE RICEVUTE
     NELLA MATRICE MATRIX WORD PER QUANTO RIGUARDA LE PAROLE E L'
     ARRAY MATRIXCOUNT PER IL CONTEGGIO
151 if(myrank==0) {
152   for(int i=0;i<(bufSizeRecv/charInt);i++) {
153     MPI_Unpack(recvbuf,bufSizeRecv,&position,matrixWord[i
    ],500,MPI_CHAR,MPI_COMM_WORLD);
154     MPI_Unpack(recvbuf,bufSizeRecv,&position,&matrixCount[i
    ],1,MPI_INT,MPI_COMM_WORLD);
```

```
155     }
156     //IL RANK 0 EFFETTUA LA FUNZIONE DI REDUCE IN MODO DA
        SOMMARE I CONTEGGI DELL STESSE PAROLE CHE GLI ALTRI PROCESSI
         HANNO INVIATO
157     for(int i=0;i<bufSizeRecv/charInt;i++) {
158       for(int j = 0;j<dimensione;j++) {
159         if(strcmp(matrixWord[i],matrixFinalWord[j])==0) {
160           verifica = 1;
161           indiceWord = j;
162           }
163         }
164
165
166       if(verifica==1) {
167         countFinalWord[indiceWord] = countFinalWord[indiceWord]
         + matrixCount[i];
168         verifica = 0;
169         indiceWord = 0;
170       }
171       else {
172         dimensione++;
173         countFinalWord = (int*)realloc(countFinalWord,sizeof(
        int)*dimensione);
174         countFinalWord[dimensione-1] = matrixCount[i];
175         matrixFinalWord = (char**)realloc(matrixFinalWord,
        dimensione*sizeof(char*));
176         matrixFinalWord[dimensione-1] = (char*)malloc(500*
        sizeof(char));
177         strcpy(matrixFinalWord[dimensione-1],matrixWord[i]);

178         }
179       }
180       //ORDINAMENTO PER FREQUENZA
181       Sort(countFinalWord,matrixFinalWord,dimensione);
182       CSV(countFinalWord,matrixFinalWord,dimensione);
183       printf("WORD COUNT TERMINATO\n");
184       fflush(stdout);
```

```
185   }
186
187   free(countFinalWord);
188   for(int i = 0;i<dimensione;i++) {
189     free(matrixFinalWord[i]);
190   }
191   free(matrixFinalWord);
192   free(recvbuf);
193   free(matrixCount);
194   for(int i = 0;i<(bufSizeRecv/charInt);i++) {
195     free(matrixWord[i]);
196   }
197   //printf("buffer%d\n",bufSizeRecv);
198
199 }
```

La funzione WordCount come detto prima è la funzione principale del programma, in cui viene svolto lavoro in parallelo, lavoro effettuato solo dal Master e la comunicazione con MPI tra i nodi del cluster. Alla fine del lavoro di partizionamento ogni nodo del cluster conosce quanti byte deve leggere, da quale byte deve iniziare e da quale file deve partire. Infatti vengono utilizzate queste due informazioni memorizzate nelle variabili "byteDaLeggere" e "indiceInizioFile" nel primo ciclo while. In questo ciclo ogni nodo del cluster inizia a leggere dal proprio file assegnato fino a quando non esaurisce i byte oppure arriva alla fine del file e quindi ne apre uno nuovo. Ogni volta ovviamente per leggere una stringa ogni nodo chiama la funzione "takeAndCheckString". Partizionando il lavoro in byte ovviamente si crea un problema, cioè può capitare che un nodo inizia a leggere all'interno di una stringa e quindi il conteggio delle parole non sarà più preciso. Per risolvere questo problema viene utilizzata la funzione "backWord" ma non solo. In pratica quando un nodo arriva alla fine dei sui byte si controlla se riesce a leggere interamente la sua ultima parola oppure si ferma prima. Se si ferma prima in pratica il nodo non memorizza quella parola quindi non la conta, ma passa il lavoro al nodo che viene dopo. La funzione "backWord" viene eseguita una sola volta da ogni nodo e controlla se il nodo

si trova all'interno di una parola, se questo succede significa che il nodo precedente non ha memorizzato la parola, quindi bisogna tornare indietro all'inizio della stringa e partire da quel punto.

Listing 2.9: backWord

```
void backWord(FILE* file, int partenza, int myrank) {
/*  INPUT:
  FILE* file:   FILE DI RIFERIMENTO PER RECUPERARE LA PAROLA
   CHE IL PROCESSORE P-1 HA SALTATO
  int partenza:   INTERO CHE DETERMINA IN QUALE POSIZIONE
   BISGONA INIZIARE A LEGGERE
  int myrank    INTERO RELATIVO AL PROCESSO
  */
  int verifica = 1;
  char c;
  int indietro = partenza;
  //if(myrank==1){printf("partenza:%d\n",partenza);}
  if(file!=NULL) {
    while(verifica) {
      fseek(file,indietro, SEEK_SET );
      c = fgetc(file);
      if(indietro==0||(!isalnum(c))) {verifica = 0;}
      else {
        indietro--;
      }
    }
    fseek(file,indietro, SEEK_SET );
  }
}
```

Una volta che un nodo finisce il suo lavoro quindi arriva alla fine dei suoi byte ed ha sicuramente memorizzato il tutto nelle due strutture commentate prima, è pronto ad inviare il tutto al nodo MASTER. Quindi dopo la lettura e il conteggio delle parole avviene la fase di comunicazione con MPI. Per comunicare è stata utilizzata la comunicazione collettiva e quindi le funzioni MPI-Gather ed MPI-Gatherv. Ogni nodo prima di inviare le due strutture al MASTER,

con la MPI-Gather invia prima la dimensione di queste, siccome la dimensione è variabile per ognuno. Dopo aver inviato la dimensione delle strutture, ogni nodo effettua il Pack di queste, in pratica invece di inviare le due strutture si invia un buffer contiguo e viene creato quindi un nuovo tipo di dato.

Listing 2.10: Pack delle strutture

```
1 for(int i = 0;i<*size;i++) {
2   MPI_Pack(parola[i],500,MPI_CHAR,message,bufsiz,&position,
     MPI_COMM_WORLD);
3   MPI_Pack(&countParola[i],1,MPI_INT,message,bufsiz,&position,
     MPI_COMM_WORLD);
4 }
```

Una volta effettuato il Pack ogni nodo invia il nuovo tipo di dato con la MPI-Gatherv, utilizzata siccome la dimensione del nuovo tipo di dato può variare per ogni nodo. Il nodo Master in ricezione salverà il tutto in un buffer contiguo che dovrà scompattare con la funzione MPI-Unpack. Scompattando il buffer il MASTER ricrea le due strutture, una per il conteggio delle parole e una per le parole. A questo punto il Master avrà a disposizione il lavoro svolto da tutti i nodi incluso il suo. Infine dovrà riorganizzare il conteggio siccome i nodi potrebbero aver contato le stesse parole in file differenti, quindi crea altre due strutture uguali alle precedenti ed effettua la funzione di "Reduce". Finito il seguente lavoro, con le funzioni commentate precedentemente effettua sia l'ordinamento che il salvataggio delle occorrenze in un file CSV.

CAPITOLO 3

BENCHMARK

La soluzione proposta è stata testata su Amazon Aws dove è stato istanziato un Cluster. Prima di commentare i Test effettuati, sarà descritta l'architettura del Cluster e il tipo di Input utilizzato per il test.

3.1 Architettura

IL Cluster è stato istanziato su Aws e comprende 8 istanze tutte dello stesso tipo. Le istanze erano del tipo m4.large ed hanno la seguente architettura hardware:

- 2vCPU
- 8GB RAM;
- 450 Mbs bandwidth
- 25 GB storage EBS

La configurazione software è invece composta da Ubuntu® 18.04.

3.2 Scalabilità forte

In questo tipo di test l'input rimane fisso ma vengono incrementati il numero di nodi del cluster su cui eseguire il programma. Il test aiuta a capire come il programma reagisce all'aumento del numero di nodi, quindi sono state utilizzate due metriche: Speedup ed efficienza. Inoltre è importante sottolineare che sono stati effettuati 5 test per numero di istanza. Lo speedup nella scalabilà forte si calcola nel seguente modo:
Sia n il numero di nodi, t1 il tempo di esecuzione per 1 nodo e tn il tempo per n nodi, allora lo Strong Scalability Speedup è :

$$Sstrong = (\frac{t1}{tn}) \tag{3.1}$$

Per quanto riguarda l'efficienza il calcolo è il seguente:
Sia n il numero di nodi, t1 il tempo di esecuzione per 1 nodo e tn il tempo per n nodi, allora la Strong Scalability Efficiency è :

$$Estrong = (\frac{t1}{n * tn}) \tag{3.2}$$

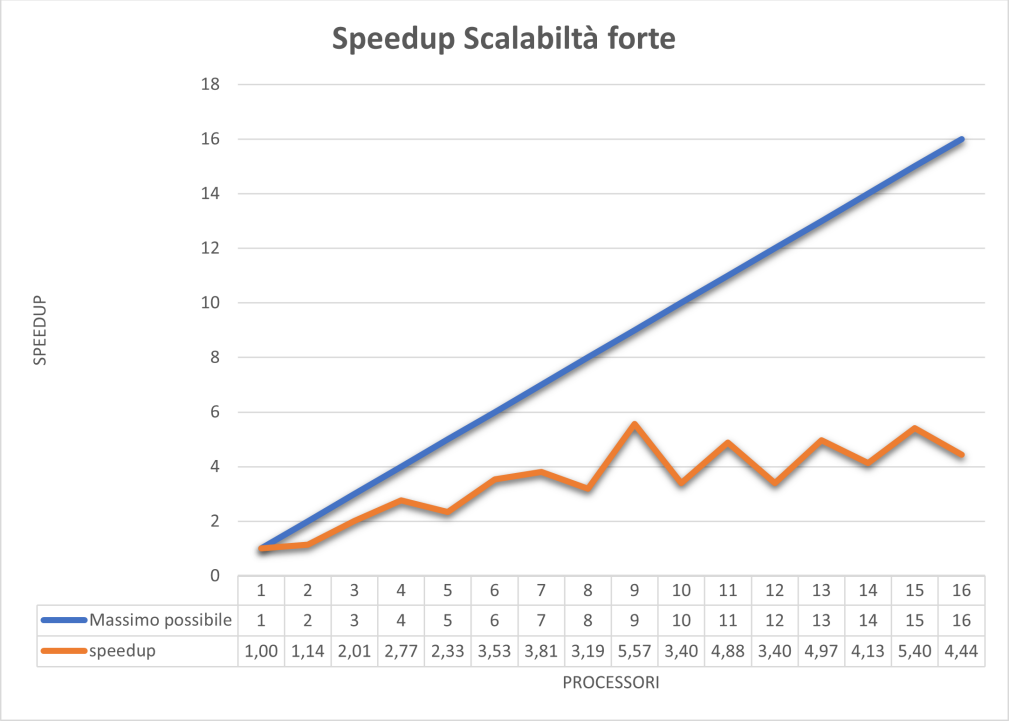
Per quanto rigurda l'input, come detto prima rimane lo stesso al variare del numero di nodi. Sono stati utilizzati 13 file che hanno la rispettiva taglia:

- Alice's adventures in wonderland (147 KB)
- PETER PAN (256 KB)
- The Wonderful Wizard of Oz (203 KB)
- file1.txt (255 KB)
- file2.txt (255 KB)
- file3.txt (255 KB)
- file4.txt (255 KB)
- file5.txt (255 KB)
- file6.txt (255 KB)

- file7.txt (255 KB)
- file8.txt (255 KB)
- file9.txt (255 KB)
- file10.txt (255 KB)

I risultati ottenuti vengono mostrati nei seguenti grafici:

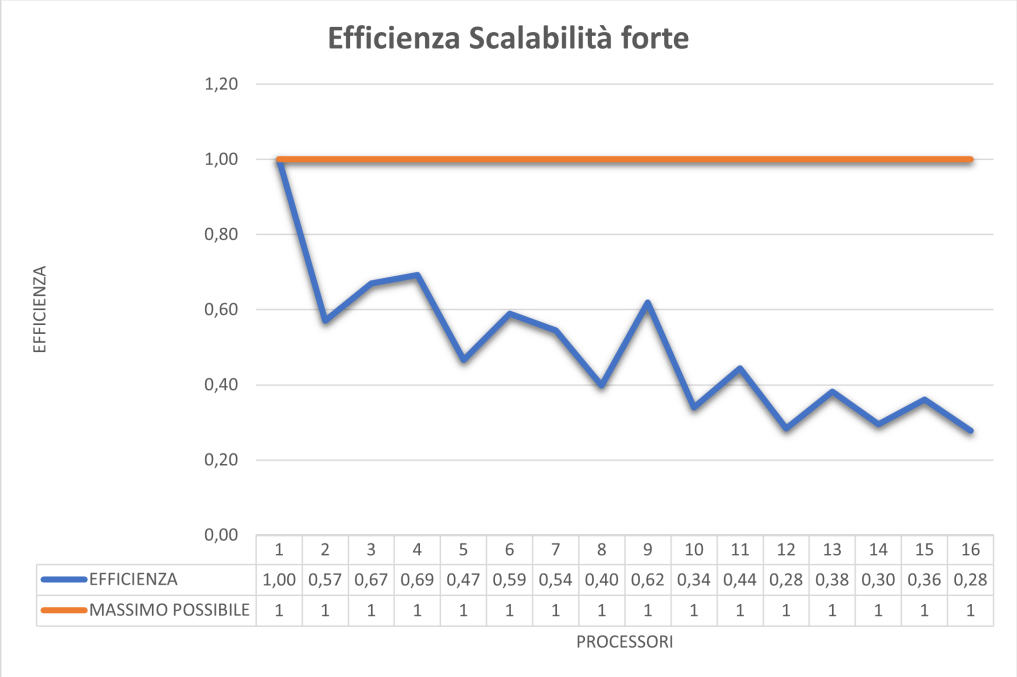
Figura 3.1: Seedup Scalabilità forte



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Massimo possibile	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
speedup	1,00	1,14	2,01	2,77	2,33	3,53	3,81	3,19	5,57	3,40	4,88	3,40	4,97	4,13	5,40	4,44

Dalla figura si può vedere che con 16 vCPU si è ottenuto uno speedUp di 4,44, un risultato per niente ottimale. Probabilmente il risultato è dovuto non dal tempo speso per la comunicazione ma per il tempo speso a computare, sopratutto nella parte iniziale e finale del programma, dove vengono effettuate le fasi di partizionamento del lavoro e raggruppamento delle occorrenze.

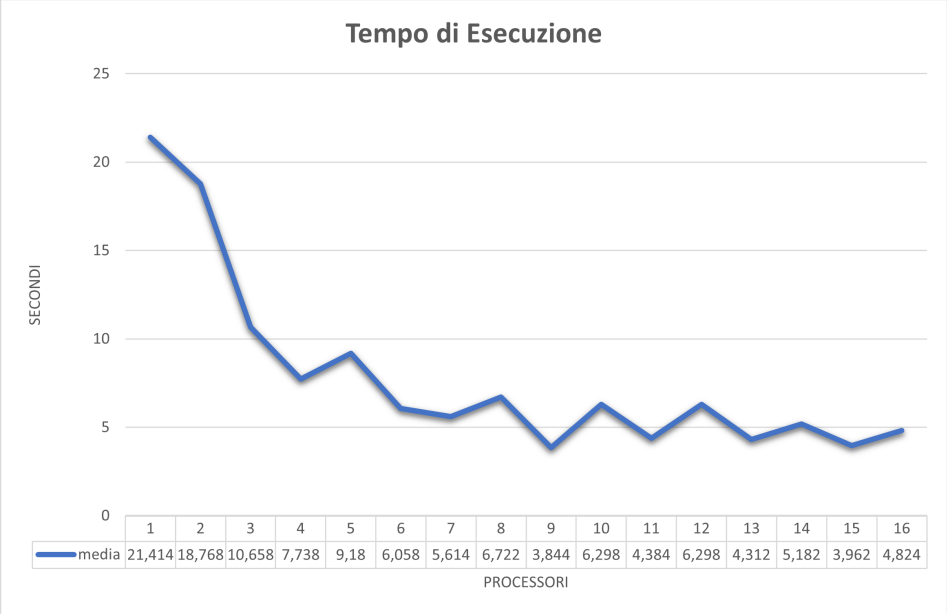
Figura 3.2: Efficienza Scalabilità forte



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
EFFICIENZA	1,00	0,57	0,67	0,69	0,47	0,59	0,54	0,40	0,62	0,34	0,44	0,28	0,38	0,30	0,36	0,28
MASSIMO POSSIBILE	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Nella seguente figura è possibile vedere che l'efficienza migliore con l'aumentare delle vCPU si raggiune con 4vCPU. Con l'aumentare dei nodi l'efficienza cala drasticamente, questo fa capire che aumentando il numero di nodi l'overhead della comunicazione in questo caso può comportare ad un calo delle prestazioni.

Figura 3.3: Tempo di esecuzione



La figura mostra i tempi di esecuzione al variare delle vCPU. Ovviamente il tempo di esecuzione dovrebbe essere minore aggiungendo unità di elaborazione. In questo caso si ha che con 16 vCPU il programma diventa 5 volte più veloce.

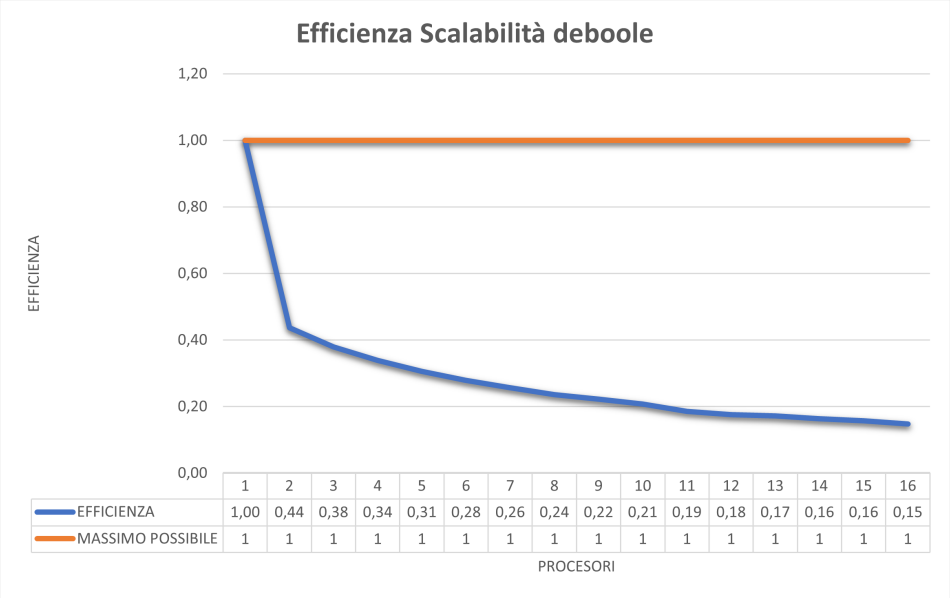
3.3 Scalabilità debole

In questo tipo di test l'input viene incrementato in maniera costante al variare dei nodi che vengono utilizzati. Questo test ci aiuta a capire se il programma richiede molta memoria, un elevato numero di risorse e l'impatto della comunicazione. Si ottiene scalabilità debole lineare se il tempo di esecuzione rimane costante mentre il carico di lavoro viene incrementato con il numero di nodi. Nel test è stata utilizzata solo la metrica di efficienza e sono stati effettuati anche questa volta 5 esperimenti per numero di istanza. L'efficienza nella scalabilità debole è calcolata nel seguente modo:
Sia n il numero di nodi, t1 il tempo di esecuzione per 1 nodo e tn il tempo per n nodi, allora la Weak Scalability Efficiency è :

$$Eweak = (\frac{t1}{tn}) \tag{3.3}$$

Per quanto riguarda l'input è stato utilizzato un solo file, cioè "PETER PAN" (256 KB) che veniva copiato in maniera costante al variare delle vCPU utilizzate. Ad esempio utilizzando 5 vCPU l'input è composto da 5 file "PETER PAN". I risultati ottenuti sono i seguenti:

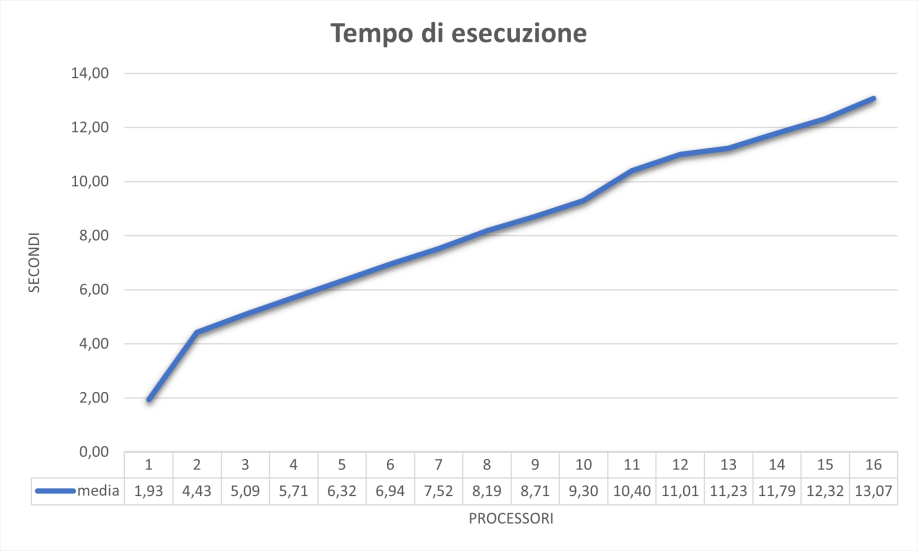
Figura 3.4: Efficienza Scalabilità debole



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
EFFICIENZA	1,00	0,44	0,38	0,34	0,31	0,28	0,26	0,24	0,22	0,21	0,19	0,18	0,17	0,16	0,16	0,15
MASSIMO POSSIBILE	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Dalla figura si può notare come il programma si comporta male in termini di efficienza per quanto riguarda la scalabilità debole. Il calo è drastico sopratutto dopo 4vCPU, probabilmente anche in questo caso la computazione iniziale e finale generano molto overhead sopratutto in termini di utilizzo di risorse e memoria.

Figura 3.5: Tempo di esecuzione



Nella figura si può vedere come il tempo di esecuzione aumenta con l'aumentare del carico di lavoro e numero di vCPU. Un risultato aspettato, siccome da come si è visto in precedenza, in termini di efficienza il programma non si comportava bene. Infatti con 16 vCPU il tempo di esecuzione è 7 volte più lento, da considerare però che il carico passa da 250 KB a 4MB.

CAPITOLO 4

CONCLUSIONI

Nei capitoli precedenti è stato presentato il problema Word Count e una soluziona distribuita scritta in c con MPI su un cluster AWS. Il Cluster era composto da 8 istanze ognuna con 2 vCPU, è stata testata la soluzione in termini di scalabilità forte e scalabilità debole. Si è cercato di capire tramite i test che impatto aveva l'aumento del numero di nodi con input fisso e con input in aumento costante insieme ad un incremento dei nodi. Dai risultati abbiamo visto che il programma soffre molto in termini di scalabilità debole. Anche per quanto riguarda la scalabilità forte i test non sono stati ottimi. Probabilmente uno dei problemi principali è l'overhead computazionale che impatta su entrambi i test, con gli scarsi risultati della scalabilità debole si può evincere che l'overhead dovuto alla comunicazione impatta duramente, probabilmente insieme ad un elevato uso delle risorse e della memoria.