Progetto	Nome e Cognome	Data di consegna
WordCount	Mario Santoro	02/06/2020

Descrizione della soluzione

Il **Word Count** è il calcolo dell'occorrenza (o frequenza) delle parole in uno (o più) documenti (file ".txt"). Il Word Count può essere necessario quando è richiesto un testo per rimanere all'interno di un numero specifico di parole.

La soluzione trovata si compone nelle fasi seguenti:

- il nodo MASTER legge l'elenco dei file da una cartella chiamata "file", che calcolerà la dimensione in byte di ogni file (e la dimensione totale). Una volta letto i file, il MASTER calcolerà quanti byte dovrà "consumare" ogni singolo processo, in una struttura vengono definite le informazioni da passare ai processi, del tipo quali file aprire e da dove iniziare e dove finire la lettura nei file che gli sono stati assegnati. Nel frattempo il MASTER legge la porzioni di byte dai file assegnati salvando le parole trovate in un array, poi calcola, della propria porzione di array, le parole e la loro occorrenza salvondaole in una struttura.
- Una volta che un processo SLAVE ha ricevuto le informazioni dal MASTER, inizia la lettura nei file delle porzioni di byte che gli sono stati affidati inserendo le parole in un array, di cui ne calcolerà la frequenza che ricorre ogni parola trovata, infine l'apposita struttura vine inviata al MASTER.
- Il MASTER riceve le strutture calcolate dai processi e combina i risultati. Ad esempio, la parola "World" potrebbe essere contata in più processi e quindi somma tutte queste occorrenze in un unica struttura. Infine crea un file CSV con i risultati (coppia parola e frequenza).

Codice

Per prima cosa il processo MASTER chiama la funzione *CalcolaByte(SizeByte byte[LBYTE])* per inserire nell'array di *SizeByte* (una struttura creata) il nome del file e la sua dimensione in byte. Restituisce la dimensione totale dei byte (tutti i file).

```c
/*funzione che calcola in una struttura SizeByte la dimensione di ogni file
e restituisce la dimensione (di byte) totale per tutti i file*/
long CalcolaByte(SizeByte byte[LBYTE])
{
    char ch;
    int j = 0;
    FILE *fp;
    DIR *dirp = NULL;
    struct dirent *dp;
    char path[MAXPATHLEN];

    /*getwd restituisce un percorso file assoluto che rappresenta
     la directory di lavoro corrente.*/
    if (!getwd(PATHNAME))
    {
        printf("Error getting path\n");
        exit(0);
    }
    /*concateno il percorso della directory corrente con /file
    per indicare dove recuperare i file*/
    strcat(PATHNAME, "/file/");

    /*prendo i file dalla cartella e apro i file*/
    dirp = opendir(PATHNAME);
    long size = 0;
    while (dirp)
    {
        if ((dp = readdir(dirp)) != NULL)
        {
            if (strcmp(dp->d_name, ".") != 0 && strcmp(dp->d_name, "..") != 0)
            {
                /*recupero i nomi dei file dalla cartella e lo concateno
                al percorso assoluto trovato, oltre che inserire il
                nome del file nella struttura*/
                char *ptr = dp->d_name;
                strcpy(byte[j].file, ptr);
                strcpy(path, PATHNAME);
                strcat(path, ptr);

                if ((fp = fopen(path, "rt")) == NULL)
                {
                    printf("Errore nell'apertura del file'");
                    exit(1);
                }
                /*mi posiziono alla fine del file e con ftell calcolo la dimensione
                 del file per inserirla nella struttura */
                fseeko(fp, 0, SEEK_END);
                long tmp = ftell(fp);
                byte[j].sizeByte = tmp;
                size += tmp;
```

```
                j++;
            }
        }
        else
        {
            closedir(dirp);
            dirp = NULL;
        }
    }
    fclose(fp);
    closedir(dirp);
    return size;
}
```

Successivamente una funzione *partitioning(long taglia, int p, long *partitioning)* divide la taglia totale ottenuta dalla funzione precedente per il numero di processi per stabilire quanti byte ogni processo dovrà "consumare", in caso di resto assegna equamente a ogni processo la somma rimasta in questo modo:

```
int temp = 0;
while (resto != 0)
      {
          partitioning[temp % p] = partitioning[temp % p] + 1;
          temp++;
          resto--;
      }
```

A questo punto il MASTER chiama la funzione *splitByte(SizeByte byte[LBYTE], long taglia, ByteSplit sp[LBYTE], long *partitioning, int *send)*, questa funzione si occupa di riempire un array di *ByteSplit*, una nuova struttura, contenente le informazioni da inviare agli SLAVE, la struttura in questione, è strutturata come nell'esempio che segue:

rank	nameFIle	start	end
0	file.txt	0	899
1	file.txt	900	1300
1	file1.txt	0	499
2	file1.txt	500	1000
...	...	...	...

start indica dove il rank deve iniziare a "consumare" byte nel file ed end dove finire.

La funzione riempirà anche un array di interi che ogni posizione inidca il processo, mentre il contenuto indica quante righe della struttura mandare al processo i-esimo, es: send[1]= 2 perchè al rank 1 dovranno essere inviate due righe della struttura.

La funzione è implementata nel modo seguente (spiegazioni nei commenti):

```c
int splitByte(SizeByte byte[LBYTE], long taglia, ByteSplit sp[LBYTE], long
*partitioning, int *send)
{
    int i = 0; //indice che scorre struttura SizeByte byte
    int j = 0; //indice che scorre struttura ByteSplit sp
    /*variabile che sarà settata nell'array di interi send,
    inidica quanti record della struttura ByteSplit dovrà utilizzare
    il processo k-esimo*/
    int count = 0;
    int rank = 0;   //identifica il processo che dovrà utilizzare quelle informazioni
    long start = 0; // da dove il processo inizierà a "consumare i byte" in un file
    long end = 0;   // da dove il processo finirà di "consumare i byte" in un file
    /*variabile che somma per ogni iterazione il consumo di byte dei processi,
    quando sono stati assegnati tutti i byte e la variabile è uguale a taglia
    (totale byte di tutti i file) si può uscire dal while*/
    int res = 0;
    do
    {
        //setto da dove il processo deve iniziare a consumare byte
        sp[j].start = start;
        sp[j].rank = rank; //setto il rank corrente
        /*se ciò che deve consumare il processo è minore o uguale
        //alla dimensione totale del file i-esimo*/
        if (partitioning[rank] <= byte[i].sizeByte)
        {
            //setto in rank le righe totali della struttura utilizzati dal processo
            send[rank] = count + 1;
            //copio il nome del file nella struttura sp
            strcpy(sp[j].nameFile, byte[i].file);
            /*l'end è l'inizio + il valore che doveva
            consumare il processo (-1)*/
            end = start + partitioning[rank] - 1;
            //setto dove il processo deve finire di consumare byte
            sp[j].end = end;
            /*modifico il valore della dimensione di byte del file sottraendo
            il valore già consumato dal processo corrente*/
            byte[i].sizeByte -= partitioning[rank];
            //incremento il valore dei byte consumati in questa iterazione
            res += partitioning[rank];
            /*si incrementa rank, si passa al processo successivo
            poichè ha consumato i byte che doveva*/
            rank++;
            /*lo start start del processo successivo diventa l'end (quindi dove
            si è fermato a consuamare byte) del processo corrente*/
            start = end;
            //count torna a 0 per il processo successivo
            count = 0;
            /* se  il file è rimasto a 0 byte
            devo passare al file successivo*/
            if (byte[i].sizeByte==0)
```

```
            {
                i++;
            }
        }
        /*se ciò che deve consumare il processo è strettamente
         maggiore alla dimensione totale del file i-esimo*/
        else
        {
            /*end sarà start (inizio del consumo di byte)
             + (dimensione di byte rimasti nel file- 1)*/
            end = start + byte[i].sizeByte - 1;
            sp[j].end = end; //setto l'end
            /*il valore che deve consumare il processo corrente viene
            scalato per i byte "consumati" nel file corrente*/
            partitioning[rank] -= byte[i].sizeByte;
            //incremento il valore dei byte consumati in questa iterazione
            res += byte[i].sizeByte;
            //lo start trona a 0 perchè si passa a un file successivo
            start = 0;
            /*il processo avrà bisogno di consumare da un altro file quindi
            si incrementa il contatore delle riga della struttura per processo*/
            count++;
            //setto nella struttura il nome del file
            strcpy(sp[j].nameFile, byte[i].file);

            i++; //passo al file successivo
        }
        j++; //incremento la struttura ByteSplit
    } while (res != taglia);

    return j;
}
```

Adesso il MASTER è pronto a inviare agli SLAVE le informazioni, inviandogli col tag 0 la dimensione della struttura in arrivo, mentre la seconda send con tag 1 la locazione di memoria che è stata assegnata per essere computata da quel processo:

```
int index = send[0];
        for (int i = 1; i < p; i++)
        {
            //prima send che contiene la lunghezza della struttura in invio
            MPI_Send(&send[i], 1, MPI_INT, i, tag, MPI_COMM_WORLD);
            //seconda send con la struttura, quanto inviare è stabilito in row[i]
            MPI_Send(&sp[index], row[i], st, i, tag + 1, MPI_COMM_WORLD);
            //incremento index per il processo successivo
            index += row[i];
        }
```

Gli slave ricevono le informazioni con due recive:

```
int siz;
        //ricevo la dimensione della struttura in arrivo nella recive succssive
        MPI_Recv(&siz, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
        /*ricevo la struttura contente le informazioni
        necessarie per consuamre byte nei file*/
        MPI_Recv(&sp, siz, st, 0, tag + 1, MPI_COMM_WORLD, &status);
```

A questo punto sia il MASTER che gli SLAVE richiamano una funzione (tante volte quanti sono i file da cui attingono i byte) per recuperare le parole loro assegnate, la funzione è *riempioArray(char array[LENGTH][CHARLENGTH], ByteSplit sp, int start)*, in questa funzione viene aperto il file contenuto nella struttura passata come argomento. Dopodichè ci si posiziona sul byte di partenza con la *fseeko* e si inizia a scorrere il file carattere per carattere, saltando (solo se non si parte dal byte 0) la prima parola destinata al processo precedente, infatti un flag viene settato a 1 solo se si arriva al \n della prima parola, mentre per la fine dal while si può uscire quando si è arrivato al byte di fine e senza spezzare la parola quindi continua fino al \n. Ecco la parte saliente della funzione:

```
int j = 0; //scorre i caratteri
/*posizionarsi su un byte specifico all'interno del
file per iniziare a consumare byte*/
fseeko(fp, sp.start, SEEK_SET);
char ch;
int flag = 0;
long sizech= sp.end; //dove terminare nel file
int count = sp.start; //dove iniziare
/*Il while deve continuare finchè il valore di inizio "count" arriva
all'end "sizech" o finchè è diverso \n per non avere parole tagliate*/
while (count <= sizech || ch != '\n')
{
    ch = fgetc(fp); //lettura del carattere nel file
    /*se siamo all'inizio del file settiamo il flag a 1 per fargli leggere
     la parola, altrimenti la prima parola è lasciata al processo precedente*/
    if (count == 0)
    {
        flag = 1;
    }
    if (flag == 1) // se il flag è 1 iniziamo a riempire l'array con le parole
    {
        if (ch != '\n') //se è diverso da \n allora è ancora la stessa parola
        {
            array[start][j] = ch;
            j++;
        }
        /*se è \n aggiungo il tappo per concludere parola e vado avanti
        alla riga successiva (nuova parola) azzerando il contatore dei caratteri*/
        if (ch == '\n')
        {
            array[start][j] = '\0';
            start++;
            j = 0;
        }
    }
    /*quando non ci troviamo all'inizio del file lasciamo che la prima parola
    al processo precedente, quindi se il flag è 0 (inizio della lettura) e
    arriviamo al \n quindi alla fine della prima parola, settiamo il flag
    a 1 e all'iterazione successiva iniziamo a prendere le parole*/
    if (flag == 0 && ch == '\n')
    {
        flag = 1;
    }

    count++;
}
return start;
```

Una volta riempito l'array con le parole i processi possono calcolare le parole e le loro occorrenze. Compito assegnato alla funzione *calcolaFrequenza(char array[LENGTH][CHARLENGTH], int row, Word words[LENGTH])* che riempie l'array di *Word*, struttura per immagazzianare le parole e le occorrenze, nel seguente modo:

```
int calcolaFrequenza(char array[LENGTH][CHARLENGTH], int row, Word words[LENGTH])
{
    //setto flag a 0
    int bool = 0;
    //inserisco primo elemento dell'array nella struttura con frequenza 1
    strcpy(words[0].name, array[0]);
    words[0].frequenza = 1;
    int size = 1;
    //scorro tutto l'array parola per parola
    for (int i = 1; i < row; i++)
    {
        //scorro la struttura
        for (int j = 0; j < size; j++)
        {
            /*controllo se è gia presente la parola nella struttura,
            nel caso aumento la frequenza della parola e setto il flag a 1,
             infine posso uscire dal ciclo che scorre la struttura*/
            if (strcmp(array[i], words[j].name) == 0)
            {
                bool = 1;
                words[j].frequenza++;
                break;
            }
        }
        /*se il flag è ancora a 0 non ho trovato l'occorrenza della parola
         nella struttura quindi si aggiunge una nuova parola con
         frequenza 1 nella struttura*/
        if (bool == 0)
        {
            strcpy(words[size].name, array[i]);
            words[size].frequenza = 1;
            size++;
        }
        bool = 0;
    }

    return size;
}
```

A questo punto i porcessi SLAVE inviano la struttura *Word* calcolata al MASTER, quest'ultimo riceve le strutture e via via fonde le strutture ottenute con quella da lui calcolata con la funzione

unisciResult(Word w1[LENGTH], int sizeW1, Word w2[LENGTH], int sizeW2) che, dati due array di Word e le loro dimensioni, fonde i risultati nella struttura w2.

Infine il processo MASTER crea un file CSV con i risultati ottenuti dalla struttura Word finale.

Note sulla compilazione

Copiare il codice della sezione precedente in un file ".c". La cartella "file" deve essere nello stesso percorso del file. I file al suo interno possono avere qualsiasi nome, ma nell'implementazione è imposto un limite superiore per la lunghezza del nome (20 caratteri) e quanti file è possibile creare (1000).

Per procedere alla compilazione, utilizzare *mpicc* come segue:

```
$ mpicc nome_file.c -o nome_eseguibile
```

e poi avviarlo con:

```
$ mpirun -np p nome_eseguibile
```

Specificando in p il numero di processi coinvolti nell'esecuzione.

Note sull'implementazione

Le parti salienti del codice sono descritte nelle sezioni precedenti, è possibile vedere l'intero codice sorgente dal file "*WordCounter.c*" allegato, dove ogni sezione è ampiemente descritta con commenti.

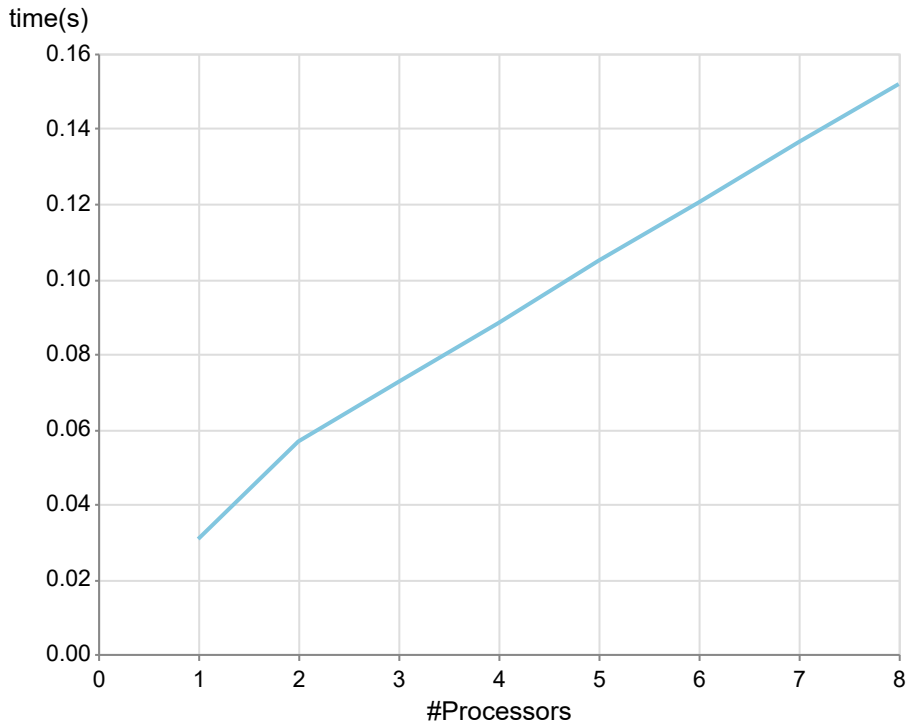
Nei file le parole sono seguite da '\n' per semplificare la lettura nell'implementazione.

Il limite imposto nell'algoritmo proposto sono parole lunghe al massimo 20 caratteri (nell'alfabeto italiano non esistono parole così lunghe), mentre il numero di parole totali che è possibile esaminare (con 1 processo) è 120.000 poiché è il limite imposto alla dimensione dell'array, per incrementare cambiare il valore nella *define* iniziale.

Risultati

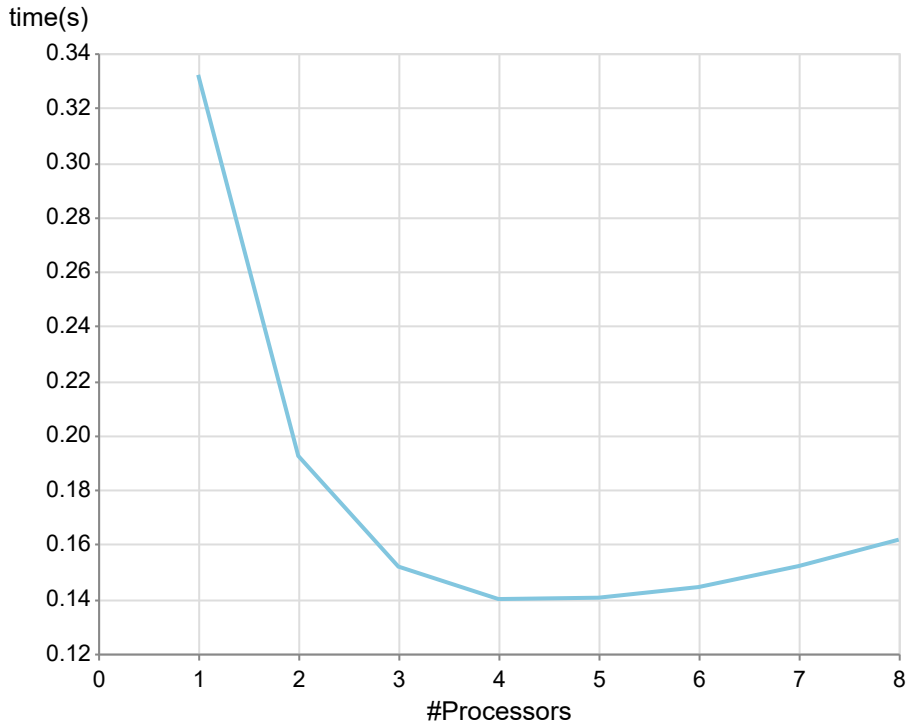
Scalabilità forte

P:	1	2	3	4	5	6	7	8
N:	10.000	20.000	30.000	40.000	50.000	60.000	70.000	80.000



Scalabilità debole

Per N= 110.000



Descrizione risultati

Sono stati illustrati i risultati ottenuti con numero di processori da 1 a 8, lanciando il programma dalla macchina MASTER con il seguente comando:

```
$ mpirun -np <number_of_instance> --host <ip_instance1>,<ip_instance2>,...,
<ip_instance_n>
```

Il primo grafico mostra l'andamento della *scalabilità forte*, la dimensione del problema aumenta allo stesso ritmo del numero di processori, mantenendo uguale la quantità di lavoro per processore. Nonostante la taglia per numero di processori rimane invariata, i tempi di esecuzione sembrano subire un rallentamento costante.

Nel secondo grafico viene illustarto l'andamento della *scalabilità debole* cioè la velocità con cui diminuisce l'efficenza all'aumentare dei processori, con dimensione fissa del'input. In questo caso l'efficienza del programma migliora fino a 4 processori per poi subire un incremento di inefficienza.

In ogni caso i tempi non variano eccessivamente , il che implica che la soluzione fornita dovrebbe essere in grado di risolvere problemi con input maggiori in lassi di tempo accettabili.

Conclusioni

Dopo aver analizzato l'algoritmo sia in termini di scalabilità forte che in termini di scalabilità debole si evince che l'algoritmo raggiunge il tempo di esecuzione migliore su 4 processori, dopodichè influenzato dall'overhead della comunicazione tra i processi il tempo di esecuzione inizia a peggiorare.