Word Count

Progetto del corso d'esame ***Programmazione Concorrente e Parallela su Cloud***, Anno Accademico 2020/2021 curriculum ***Cloud Computing***.

Studente: Liguorino Vincenzo
Matricola: 0522500840

Dettagli per la compilazione e l'esecuzione
Dato che il programma non presenta librerie e non è composto da molteplici file file la compilazione si esegue con un semplice comando.

```
mpicc wordCount..c -o esameLiguorino
```

L'esecuzione avviene tramite **mpirun**, specificando l'hostfile precedentemente creato

```
mpirun -np [N_PROCESSORI] –hostfile hfile  ./esameLiguorino
```

In fase di run del programma verrà chiesto all'utente il numero di file da inserire nella lista e successivamente il nome del file, nel mio caso all'interno del repository ci sono anche i tre file che sono stati utilizzati come test.

Sommario

1. Descrizione del problema

Con ***Word Count*** intendiamo il numero di parole presenti in un documento o messaggio di testo.

Questo concetto è comunemente utilizzato dai traduttori per determinare il prezzo che una persona deve pagare per un certo lavoro di traduzione, ma può essere utilizzato anche per calcolare leggibilità e velocità di digitazione e lettura. Inoltre, contare le parole può essere necessario, soprattutto in ambito accademico o legale, giornalistico e pubblicitario, quando si vuole che un testo rimanga entro un numero di parole specifico

La soluzione che andremo ad analizzare ha come obiettivo proprio quello di realizzare un conteggio del numero di parole presenti in una serie di file passati in input, tramite l'utilizzo del linguaggio di programmazione ***C*** e della libreria ***MPI (Message Passing Interface)***.

2. Descrizione dettagliata del progetto

Il progetto in questione si può dividere concettualmente in 4 parti:

- Calcolo medio delle parole da leggere
- Lettura delle parole nei file e creazione dell'array contenente parola e frequenza
- Invio e ricezione degli istogrammi locali al processo Master
- Creazione del file .csv

Adesso andiamo ad analizzare nel dettaglio punto per punto:

Calcolo medio delle parole da leggere

Il primo punto che analizziamo riguarda il calcolo medio delle parole da leggere per ogni singolo processore. Il Master prende in input un numero N di file da leggere, l'utente scrive il nome dei file che vuole prendere in considerazione e li inserisce all'interno di un array che mediante istruzione

```
MPI_Bcast(list_file,MAX_NUMBER_FILE,typeNameFile,0,MPI_COMM_WORLD);
```

invia a tutti gli ***slave***.
All'interno di esso, così come specificato dalla traccia, avviene il conteggio del numero di parole massimo che bisogna prendere in considerazione. Tutto questo viene ottenuto mediante la somma del numero di parole in ogni file, dividendolo poi per il numero di processori.
Nella mia soluzione ho fatto si che nel caso di parole pari, queste vengono divise in parti uguali tra i vari processori mentre nel caso di parole dispari ci sarà sempre un processore tra tutti che ne leggerà qualcuna in più e nel mio caso ho scelto di affidare questo compito all'ultimo processore della lista a meno che non si verifica una situazione particolare.
L'informazione relativa al numero di parole massimo che uno slave può leggere e il resto della divisione tra processori, che sarà uguale a 0 se le parole sono pari oppure N se le parole sono dispari, il Master la invia mediante istruzione MPI_Bcast come si evince dall'immagine seguente:

```
char totalWord[50];
int maxW = 0;
//The first operation is count how many words must you read
int size = sizeof(list_file)/sizeof(list_file[0]);
for (int i = 0; i < size; ++i) {
    FILE* fp = fopen(list_file[i], "r");
    // checking if the file exist or not
    if (fp != NULL) {
        while (fscanf(fp, "%s", totalWord) != EOF){
            word_count++;
        }
        fclose(fp);
    }
}

maxWords = (maxW+=word_count)/(common_size-1);
maxResto = (word_count)%(common_size-1);

MPI_Bcast(&maxWords,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(&maxResto,1,MPI_INT,0,MPI_COMM_WORLD);
```

La situazione particolare può avvenire quando un processore intermedio arriva al numero massimo di parole e gli manca solo l'ultima da leggere, in questo caso faccio si che legga anche l'ultima parola del file stesso spostando il cursore in avanti e di conseguenza l'ultimo leggerà ovviamente una parola in meno.

Lettura delle parole nei file e creazione dell'array contenente parola e frequenza

Il secondo punto che andiamo ad analizzare è l'analisi parola per parola all'interno dei file che troviamo nella lista inviata dal ***Master***.
Una volta entrati nel flusso di esecuzione degli slave il primo controllo che viene effettuato, precedente alla fase di analisi delle parole è quello di capire su quale processore siamo perché nel caso in cui siamo quello con ***rank 1***, si va ad aprire il file che si trova all'inizio dell'array; mentre nel caso in cui siamo in un processore diverso dal primo si prendono in considerazione dei parametri aggiuntivi come ad esempio:

- ***startOfAnotherPosition***: valore intero che può assumere i valori 0 e 1 ed indica al processore se bisogna partire da una posizione diversa rispetto all'inizio del file
- ***indexFile***: come spiega il nome della variabile ci indica la posizione del file da aprire che può essere la stessa del processore precedente oppure quella successiva
- ***currentPositionInFile***: contiene la posizione del cursore all'interno del file

Nel caso in cui il valore dello ***startOfAnotherPosition*** è uguale ad 1 vuol dire che il processore precedente ha letto il numero massimo di valori senza però arrivare alla fine del file e di conseguenza, si usa l'istruzione

```
fseek(fp, currentPositionInFile, SEEK_SET);
```

per riaprire lo stesso file ma da una posizione diversa da quella iniziale.

Una volta effettuato il controllo spiegato precedentemente sul processore 1 si va ad effettuare l'analisi delle parole contenute all'interno del file fino a quando non si arriva alla fine del file oppure fino a quando la variabile ***count_read_word***, che conta il numero di parole lette, non arriva alla soglia massima.

```c
//Read the word in text file and create the array containing word and occurrence
while (fscanf(fp, "%s", word) != EOF && count_read_word<maxParole ){

      //Position of curson in file
      currentPositionInFile = ftell(fp);
      count_read_word++;

      //Function that lowercase all words
      strlwr(word);

      //Removes the point if present next to the word
      len = strlen(word);
      if (ispunct(word[len - 1]) != 0){
          word[len - 1] = '\0';
      }
        // Check if word exits in list of all distinct words
        isUnique = 1;
        for (i=0; i<index && isUnique; i++) {
            if (strcmp(words[i], word) == 0)
                isUnique = 0;
        }

        // If word is unique then add it to distinct words list
        // and increment index. Otherwise increment occurrence
        // count of current word.
        if (isUnique) {
            strcpy(words[index], word);
            count[index]++;

            index++;
        } else {
            count[i - 1]++;
        }
}
```

Come si evince dall'immagine, all'interno del ciclo di analisi viene aggiornato il valore della posizione del cursore da poi eventualmente inviare al processo successivo; viene richiamata una funzione ***strlwr*** alla quale si passa in input la parola e se *maiuscola* la si trasforma in *minuscolo*; viene utilizzara la funzione ***ispunct*** che rimuove il punto se presente dopo la parola; vengono costruiti gli array ***words*** e ***count*** che conterranno i valori utili a produrre l'istogramma locale del processo (parole e rispettive frequenze).

Invio e ricezione degli istogrammi locali al processo Master

Come descritto nel punto precedente i vari processori producono i loro istogrammi locali e una volta prodotti li inviano al Master che effettua un merge fra tutti gli istogrammi per produrre poi quello finale che conterrà tutte le parole dei vari processori e la loro frequenza totale.
Gli array finali che esso ottiene vengono passati ad una funzione ***sortArray*** che ordina le parole da quella con frequenza più alta a quella con frequenza più bassa.

Creazione del file .csv

Dopo aver effettuato l'ordinamento degli array, l'ultima operazione è quella di creare un file .csv contenente tutte queste informazioni, mediante le seguenti operazioni:

```
fpt = fopen("FinalHistograms.csv", "w+");
```

per aprire il file in lettura e scrittura;

```
fprintf(fpt, " Words, Frequency\n");
```

per indicare il titolo delle colonne ed infine

```
for (i=0; i<index; i++) {
        if(finalCountWord[i] !=0) {
                fprintf(fpt,"%s, %d\n", finalWord[i], finalCountWord[i]);
        }
}
```

per stampare all'interno del file il contenuto degli array.

3. Analisi di Performance

Le misurazioni in termini di performance sono state effettuate su di un cluster ***t2.xlarge*** EC2 di AWS avente ogni macchina 4 vCPU e 16GB di memoria.
Sono stati effettuati in totale tre test per valutare la ***Strong*** e la ***Weak Scalability***, andando nel dettaglio i test effettuati sono:

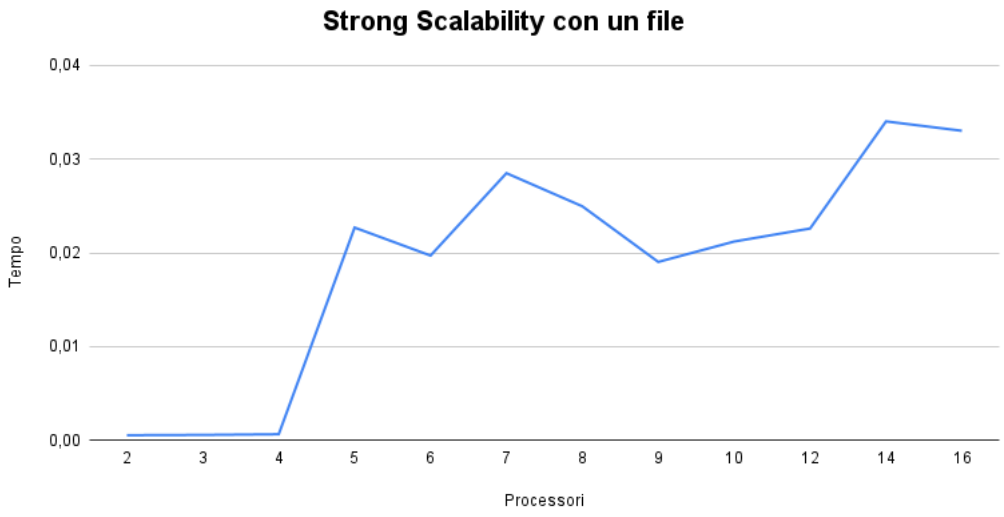
- ***1 file*** dato in input su rispettivamente **2,3,4,5,6,7,8,9,10,12,14,16 vCPUs**, con due ripetizioni ognuno, con lo scopo di misurare la ***Strong Scalability***.
- ***2 file*** dati in input su rispettivamente **2,3,4,5,6,7,8,9,10,12,14,16 vCPUs**, con due ripetizioni ognuno, con lo scopo di misurare la ***Strong Scalability***.
- Per quel che riguarda la ***Weak Scalability*** si possono utilizzare alcuni dei risultati ottenuti precedentemente con l'aggiunta di un ulteriore test.

Come si può notare le misurazioni partono dall'utilizzo di un numero minimo di 2 processori perché all'interno della computazione il ***MASTER*** ha il ruolo solo di raccoglitore di informazioni.

Strong Scalability

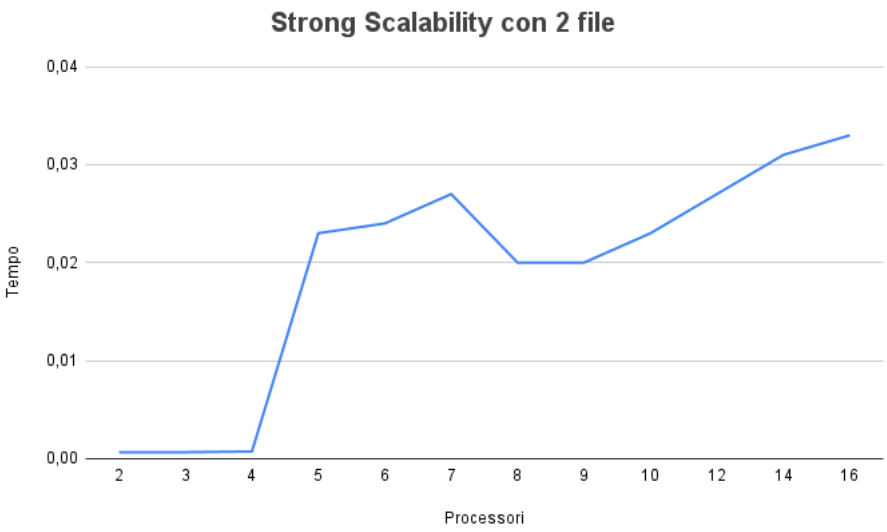
Test con un solo file

I test sono stati ripetuti più volte per ogni incremento di processori, i grafici risultanti sono una media.
Di seguito i risultati in termini di tempo al variare del numero di processori:



Test con due file

I test sono stati ripetuti più volte per ogni incremento di processori, i grafici risultanti sono una media.
Di seguito i risultati in termini di tempo al variare del numero di processori

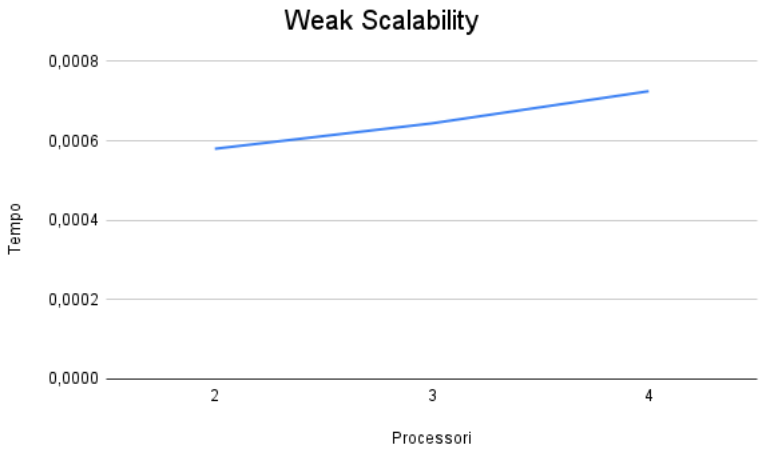


Weak Scalability

Come detto in precedenza per questo tipo di analisi si potevano utilizzare anche tempistiche ottenute in alcune casistiche della Strong Scalability. In aggiunta è stato provato un altro scenario di test

- Test con un file su 2 processori
- Test con due file su 3 processori
- Test con tre file su 4 processori

Di seguito i risultati e i grafici ottenuti:



4. Conclusioni

Come si evince dai risultati che ho ottenuto dalla mia soluzione al problema, la parallelizzazione ha i suoi pro e i suoi contro, nel mio caso viste le operazioni di separazione dei file tra i vari processi e recupero eventualmente del punto preciso da dove riprendere il conteggio, con l'aumentare dei processori le operazioni aumentano e ci sta un degrado non troppo eccessivo che però rende più prestazionale l'utilizzo di un set minimo di processori rispetto ad uno più ampio.