WordCount - Raffaele Marino - 02/06/2020

Descrizione della soluzione

Il problema WordCount consiste nel contare le occorrenze di parole all'interno di uno o più file di testo. Il problema consiste in tre fasi, la prima fase consiste nel master di andare a calcolare il numero di byte dei file, per poter suddividere questo valore con tutti i processi in maniera equa, successivamente a questa fase vi è l'invio ai vari slave delle informazioni a loro relative, ovvero quali file prendere e quali porzioni di file, in quanto poi dovranno andare nei file e calcolare per loro le occorrenze, successivamente nella terza ed ultima fase vi è l'invio da parte degli slave, della propria porzione calcolata, al master che provvederà a calcolare il tutto e a restituire il file csv. In questa versione vedremo un riadattamento del map-reduce con OpenMPI. Più nel dettaglio il programma si sviluppa nelle 3 fasi seguenti:

- Il *master* legge una lista di file, o da una cartella, ogni file quanto "pesa" in byte, ne calcola la somma e divide, per i vari processi, questo valore. Una volta fatta questa divisione il master provvederà ad inviare una struttura agli slave, tale struttura è formata da 4 campi, una che contiene il valore rank, quindi identifica a chi deve andare quella sezione, il secondo campo contiene il valore di inizio in byte, il terzo valore contiene il valore di fine in byte, e il quarto valore contiene il nome del file stesso, cosi sappiamo a quale file fanno riferimento inizio e fine.
- La seconda, dopo l'invio di questa struttura ai relativi processi, prevede il calcolo delle occorrenze delle parole nei relativi processi, in quanto andranno prima ad aprire il file, leggerne le parole e successivamente a calcolarne le occorrenze, il tutta grazie a una struttura ad hoc per questa operazione, che è composta da "parola" e "frequenza".
- L'ultima fase consiste nel mettere insieme, da parte del *master*, i risultati ottenuti dai vari slave, e scrivere il tutto in un file CSV formattato con "*parola*" - "*frequenza*".

Codice

```
#include <fcntl.h>
#include <dirent.h>
#include <sys/stat.h>
#include <time.h>
```

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <ctype.h>
#include <mpi.h>
//questo valore è statico ed indica quante parole
//sono contenute nei vari file, visto che il riallocamento
//dinamico del C non prevede una consistenza di memoria
//e con le varie malloc e realloc mi ritrovavo caratteri non corretti
#define row 400000

#define cols 16 //massima lunghezza della parola, anch'essa statica

#define splitprocesso 100 //grandezza struttura di split processo
                          //da aumentare se aumentano i processi
#define numfile 10 //numero dei file

typedef struct {
    char parola[cols];
    int frequenza;
}Word;

typedef struct {
    int rank;
    long start;
    long end;
    char nomefile[16];
}SplitPerProcesso2;

typedef struct{
    long size;
    char nomefile[16];
}SizeFile;

void BPP2(SplitPerProcesso2 * s , long * bytePerProcesso , SizeFile * bytePerFile, int
p){
    int i=0; //processi
    int j=0; //struttura
    int k=0; //file
    int n=0; //num file in struttura
    long rimanenza=0;
    long tot = 0;
    int flag=0;

    while(i<p){
        s[j].rank=i;
        s[j].start=rimanenza;
        signed long differenza = bytePerProcesso[i]-bytePerFile[k].size;
        if(differenza>=0 && k<10){   //se la taglia del processo è abbastanza grande
```

```
                                    //da prendere in input tutta la grandezza del file
            bytePerProcesso[i]=bytePerProcesso[i]-bytePerFile[k].size;
            s[j].end=bytePerFile[k].size+rimanenza;
            strcpy(s[j].nomefile,bytePerFile[k].nomefile);
            rimanenza=0;
            j++;
            k++;
        }else{
            signed long diff2=bytePerFile[k].size-bytePerProcesso[i];
            if(diff2<0){//quando k = 10, ovvero sono all'ultimo file
                        //e la differenza è < di 0 (non dovrebbe accadere)
                        //quindi riempio per la taglia rimanente del file il processo
                s[j].end=bytePerFile[k].size;

            }else{  //nel caso in cui la size del processo rimanente è minore della
size del file
                    //quindi vado a riempire per quanto rimane, dicendo il nome del
file,
                    //e passo al processo successivo, rimanendo però con quel file
                bytePerFile[k].size=bytePerFile[k].size-bytePerProcesso[i];
                rimanenza=bytePerProcesso[i]+1;
                s[j].end=bytePerProcesso[i];
                bytePerProcesso[i]=0;
            }

            //vado ad aumentare la fine del file, in quanto se un file è tagliato
            //il processo che riceverà la seconda parte, salterà la prima parola(come
da implementazione)
            //cio implicherebbe che un processo si va andrebbe saltare una parola
            //quindi apro il file, mi metto nella posizione end calcolata, e aumento
la end per quel processo
            //fino a che non trovo un \n
            strcpy(s[j].nomefile,bytePerFile[k].nomefile);
            char file1[20]="/";
            char ch;
            char pf[PATH_MAX];
            //get path of current directory
            if (getcwd(pf, sizeof(pf)) != NULL) {
                //printf(" Current working dir: %s\n", pf);
            } else {
                perror("getcwd() error");
            }
            if(strcmp(s[j].nomefile,"")!=0){
                strcat(file1,s[j].nomefile);
                strcat(pf,file1);
                FILE *in_file;
                in_file = fopen(pf, "r");
                // test for files not existing.
                if (in_file == NULL)
                {
                    printf("Error! Could not open file\n");
```

```
                    exit(-1);
                }
                fseek(in_file, s[j].end, SEEK_SET);
                ch = fgetc(in_file);
                while(ch!='\n' && isalpha(ch)){
                    ch = fgetc(in_file);
                    s[j].end++;
                    flag=1;
                }
                if(flag==1){
                    s[j].end++;
                    flag=0;
                }
                fclose(in_file);
            }
            n=0;
            i++;
            j++;
            s[j].start=rimanenza;
        }
    }
}

void contaOccorrenzeCSV(Word *parole, int lunghezza){
    FILE *fpcsv;
    int num=0;
    fpcsv=fopen("/home/pcpc/occorrenze.csv","w+"); //apro in scrittura(se già esiste
sovrascrive) file csv
    fprintf(fpcsv,"OCCORRENZA,PAROLA"); //prima riga file csv
    //per ogni parola presente nella struttura dobbiamo contare la frequenza di questa
    for(num = 0 ; num < lunghezza ; num++){
        //qui controllo se la parola che sto analizzando non sia vuota, questo perchè
        //se quando trovo una corrispondenza, io quella parola non devo più
analizzarla, e quindi
        //quando la trovo le imposto il valore " "
        if(strcmp(parole[num].parola," ")!=0){
            //questo for mi consente di analizzare dalla parola immediatamente
successiva a quella che ho
            //con il resto della struttura
            for(int a = num+1; a < lunghezza; a++){
                //se trova la corrispondenza allroa vado ad aumentare la frequenza di
tale parola
                //ovviamente deve continuare a cercare nel caso in cui trova altre
parole uguali
                if(strcmp(parole[a].parola,parole[num].parola)==0){
                    parole[num].frequenza=parole[num].frequenza+parole[a].frequenza;
                    //quando trova la parola, imposta nella struttura tale parola a "
"
                    //cosi il for principale, trova la parola " " non andrà a fare
questo for
                    //risparmiando tempo. Come se in qualche modo mi segno che ho già
```

```
analizzato questa parola
                    strcpy(parole[a].parola," ");
                }
            }
            if(strcmp(parole[num].parola,"")!=0){
                fprintf(fpcsv,"\n%s,%d",parole[num].parola, parole[num].frequenza);
//scrivo in file csv
            }
        }
    }
}

void contaOccorrenze(Word *parole, int lunghezza){
    int num=0;
    //per ogni parola presente nella struttura dobbiamo contare la frequenza di questa
    for(num = 0 ; num < lunghezza ; num++){
        //qui controllo se la parola che sto analizzando non sia vuota, questo perchè
        //se quando trovo una corrispondenza, io quella parola non devo più
analizzarla, e quindi
        //quando la trovo le imposto il valore " "
        if(strcmp(parole[num].parola," ")!=0){
            //questo for mi consente di analizzare dalla parola immediatamente
successiva a quella che ho
            //con il resto della struttura
            for(int a = num+1; a < lunghezza; a++){
                //se trova la corrispondenza allroa vado ad aumentare la frequenza di
tale parola
                //ovviamente deve continuare a cercare nel caso in cui trova altre
parole uguali
                if(strcmp(parole[a].parola,parole[num].parola)==0){
                    //essendo parole[a].frequenza = 1 è come se facessi +1
                    parole[num].frequenza=parole[num].frequenza+parole[a].frequenza;
                    //quando trova la parola, imposta nella struttura tale parola a "
"
                    //cosi il for principale, trova la parola " " non andrà a fare
questo for
                    //risparmiando tempo. Come se in qualche modo mi segno che ho già
analizzato questa parola
                    strcpy(parole[a].parola," ");
                }
            }
        }
    }
}

void ripartizioneElementi( long * arrayAppoggio,  long sizeFile, int p){
    int modulo=sizeFile%p;
    //è probiabile che per solo per alcuni processi avrò più elementi
    //controllo se il modulo mi da resto > 0
    if(modulo!=0){
        //se da > 0 allroa vado a inizializzare un array di appoggio nel quale
```

```
        //inserisco inizialmente quanti elementi devono ricevere ogni processo
        for(int i=0;i<p;i++){
            arrayAppoggio[i]=sizeFile/p;
        }
        int temp=0;
        //faccio un while decremetnando ogni volta il valore di modulo,
        //in quanto aggiungo un elemento per volta a ogni processo,
        //cosi da avere una distribuzione equa
        while(modulo!=0){
            //uso una variabile temp che mi serve solo per andare ogni volta
            //ad aggiungere un elemento per il relativo processo
            if(temp<p){
                arrayAppoggio[temp]=arrayAppoggio[temp]+1;
                temp++;
            }else{
                temp=0;
                arrayAppoggio[temp]=arrayAppoggio[temp]+1;
                temp++;
            }
            modulo--;
        }
    }else{
        //altrimenti riempio normalmente questo array di appoggio con i valori della
divisione,
        //esendo uguali per tutti i processi
        for(int i=0;i<p;i++){
            arrayAppoggio[i]=sizeFile/p;
        }
    }
}

//prendo gradndezza file
double stat_filesize(const char *filename){
    struct stat statbuf;
    if (stat(filename, &statbuf) == -1){
        printf("failed to stat %s\n", filename);
        exit(EXIT_FAILURE);
    }
    return statbuf.st_size;
}

//calcolo la dimensione dei singoli file, mettendoli in una struttura (nome file /
grandezza)
//e ritorno la dimensione totale dei file
long dimTotaleFile(SizeFile * sizePerFile){
    DIR *dir;
    struct dirent *ent;
    long sizeTotFile=0;
    char cwd[PATH_MAX];
    //get path of current directory
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
```

```
        //printf("Current working dir: %s\n", cwd);
    } else {
       perror("getcwd() error");

    }
    //add path folder
    strcat(cwd,"/file/");
    if ((dir = opendir(cwd)) != NULL) {
        int i=0;
        while ((ent = readdir (dir)) != NULL) {
            if(strcmp(ent->d_name,".")!=0 && strcmp(ent->d_name,"..")!=0){
                FILE *in_file;
                char stringa[30]="file/";
                strcat(stringa,ent->d_name);
                sizePerFile[i].size=stat_filesize(stringa);
                strcpy(sizePerFile[i].nomefile,stringa);
                sizeTotFile+=sizePerFile[i].size;
                i++;
            }
        }
    }
    return sizeTotFile;
}


int creaStrutturaParole(Word *parole, SplitPerProcesso2 * s, int count){
    int i=0; //contatore per righe
    int j=0; //contatore per colonne
    char ch;
    char separatore='\n';
    char terminatore='.';

    for(int p = 0; p<count; p++){
        char cwd[PATH_MAX];
        //get path of current directory
        if (getcwd(cwd, sizeof(cwd)) != NULL) {
            //printf("Current working dir: %s\n", cwd);
        } else {
            perror("getcwd() error");
        }
        //add path folder
        long conta = s[p].start;
        char file1[20]="/";
        strcat(file1,s[p].nomefile);
        strcat(cwd,file1);

        FILE *in_file;
        in_file = fopen(cwd, "r");
        // test for files not existing.
        if (in_file == NULL){
            printf("Error! Could not open file\n");
```

```
            exit(-1); // must include stdlib.h
        }

        //se non parte dall'inizio del file si salta la parte iniziale fino a che non
arriva al primo \n
        fseek(in_file, conta, SEEK_SET);
        if(conta!=0){
            while(ch!='\n'){
                conta++;
                ch = fgetc(in_file);
            }
        }

        //finchè non arriva alla fine che doveva fare e non trova /n,
        //cosi se viene tagliato il file, e so che il processo successivo
        //che prenderà quel file non considera la prima parola
        //(se il file non parte da 0) verrà considerata nel processo corrente
        long finefile=s[p].end;
        fseek(in_file, conta, SEEK_SET);
        while(conta<finefile){
            ch = fgetc(in_file);
            if(ch==separatore){
                parole[i].frequenza=1;
                i++;
                j=0;
            }else{
                if(isalpha(ch)){
                    parole[i].parola[j]=ch;
                    j++;
                }
            }
            conta++;
        }
        //devo cercare di fare che se il file è taglaito devo prendermi fino a fine
stringa,
        //visto che il processo che si prende il file tagliato partirà da dopo quella
stringa
        //printf("per %d , sono arrivato a %ld, dovevo arrivare a %ld, ho trovato %d
parole\n",s[p].rank,conta,s[p].end,i);
        fclose(in_file);
    }
    return i;
}

int main (int argc, char *argv[]){

    int numtasks, rank, source=0, tag=1;
    Word parole[row]={" ",0};
    MPI_Status stat;

    MPI_Init(&argc,&argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Request request = MPI_REQUEST_NULL;

    int count=3;
    MPI_Datatype wordtype,filePerProcType,oldtypes[count],oldtypes1[2];
    int blockcounts[count],blockcounts1[2];
    MPI_Aint offsets[count],offsets1[2],lb,extent;

    offsets[0] = offsetof(SplitPerProcesso2, rank);
    oldtypes[0] = MPI_INT;
    blockcounts[0] = 1;
    MPI_Type_get_extent(MPI_INT,&lb,&extent);

    offsets[1] = offsetof(SplitPerProcesso2, start);
    oldtypes[1] = MPI_LONG;
    blockcounts[1] = 2;
    MPI_Type_get_extent(MPI_LONG,&lb,&extent);

    offsets[2] = offsetof(SplitPerProcesso2, nomefile);
    oldtypes[2] = MPI_CHAR;
    blockcounts[2] = 16;
    MPI_Type_get_extent(MPI_CHAR,&lb,&extent);

    MPI_Type_create_struct(count, blockcounts, offsets, oldtypes, &filePerProcType);
//creo il tipo struttura
    MPI_Type_commit(&filePerProcType);

    offsets1[0] = 0;
    oldtypes1[0] = MPI_CHAR;
    blockcounts1[0] = cols;

    MPI_Type_get_extent(MPI_CHAR, &lb, &extent); //lo uso per dire quanto spazio c'è
fra il primo campo della struttura e il secondo
    offsets1[1] = offsetof(Word, parola); //appunto l'offset di quanti valori dal
primo campo
    oldtypes1[1] = MPI_INT;
    blockcounts1[1] = 1;

    MPI_Type_create_struct(2, blockcounts1, offsets1, oldtypes1, &wordtype); //creo il
tipo struttura
    MPI_Type_commit(&wordtype);

    long bytePerProcesso[rank];
    long sizeTotFile=0;
    SizeFile sizePerFile[numfile];
    SplitPerProcesso2 s[splitprocesso];

    if(rank==0){
        printf("per processi #%d\n",numtasks);
        clock_t Kbegin = clock();
```

```
        //parte costante del programma essendo eseguita solo dal processo 0
        sizeTotFile=dimTotaleFile(sizePerFile);
        ripartizioneElementi(bytePerProcesso,sizeTotFile,numtasks);
        BPP2(s,bytePerProcesso,sizePerFile,numtasks);
        int k=0; //indice di dove mi trovo all'interno della struttura
        int startper0=0;
        int grandezzaperzero=0;

        //celle da passare a processo 1...
        while(s[k].rank==0){
                k++;
                startper0++;
            }

        int q=startper0; //da dove devo partire
        for (int i=1; i<numtasks; i++){
            int j=0; //quanti elementi
            while(s[k].rank==i){
                k++;
                j++;
            }
            MPI_Send(&s[q], j , filePerProcType, i, tag, MPI_COMM_WORLD);
            q=k;
        }
        //MPI_Wait(&request, &stat);
        //fine parte costante
        clock_t Kend = clock();
        double Ktime_spent = (double)(Kend - Kbegin) / CLOCKS_PER_SEC;
        printf("parte costante tempo impegato = %lf\n",Ktime_spent);

        //inizio conteggio parole, parte che varia a seconda dei processi
        clock_t Pbegin = clock();
        grandezzaperzero=creaStrutturaParole(parole,s,startper0);
        contaOccorrenze(parole,grandezzaperzero);
        int grandezzaprocessi=0;
        int start2=grandezzaperzero;
        int quant=row-start2;

        for(int p = 1; p < numtasks; p++){
            MPI_Recv(&parole[start2], quant, wordtype, p, tag, MPI_COMM_WORLD, &stat);
            MPI_Get_count(&stat, wordtype, &grandezzaprocessi);
            start2=start2+grandezzaprocessi;
            quant=row-start2;
        }
        contaOccorrenzeCSV(parole,start2);
        clock_t Pend = clock();
        double Ptime_spent = (double)(Pend - Pbegin) / CLOCKS_PER_SEC;
        printf("parte parallela tempo impegato = %lf\n",Ptime_spent);
        //fine

        double TimeTot=Ktime_spent+Ptime_spent;
```

```
        printf("tempo totale = %lf\n",TimeTot);
        printf("\n");
    }else{
        int count=0;
        int grandezzaStruttura=0;
        MPI_Recv(s, splitprocesso , filePerProcType, source, tag, MPI_COMM_WORLD,
&stat);
        MPI_Get_count(&stat, filePerProcType, &count);
        grandezzaStruttura=creaStrutturaParole(parole,s,count);
        contaOccorrenze(parole,grandezzaStruttura);
        MPI_Ssend(parole, grandezzaStruttura, wordtype, 0, tag, MPI_COMM_WORLD);
    }
    MPI_Type_free(&wordtype);
    MPI_Type_free(&filePerProcType);
    MPI_Finalize();
    return 0;
}
```

Descrizione dei punti salienti del codice

```
for(num = 0 ; num < lunghezza ; num++){
        if(strcmp(parole[num].parola," ")!=0){
             for(int a = num+1; a < lunghezza; a++){
                 if(strcmp(parole[a].parola,parole[num].parola)==0){
                    parole[num].frequenza=parole[num].frequenza+parole[a].frequenza;
                    strcpy(parole[a].parola," ");
                 }
             }
        }
    }
```

Aggiornamento frequenza

Questa parte di codice prevede di scorrere la struttura per tutta la sua lunghezza, e contare la frequenza delle parole. Per evitare di contare più volte la stessa parola, ed anche di fare più volte un for che altrimenti sarebbe inutile e aumenterebbe il tempo di esecuzione, ho previsto che nel caso trovi la corrispondenza della parola, deve andare a riscrivere la parola con " ", così facendo, il primo *if* mi dirà se fare o no il secondo *for*. Ovviamente se trovo la corrispondenza andrò poi ad aggiornare la frequenza della parola con quella analizzata, così da avere la frequenza aggiornata.

```
    //se non parte dall&#39;inizio del file si salta la parte iniziale fino a che non
arriva al primo \n
        fseek(in_file, conta, SEEK_SET);
        if(conta!=0){
            while(ch!=&#39;\n&#39;){
                conta++;
                ch = fgetc(in_file);
            }
        }
```

Se start != 0

In questa parte di codice, nella funzione contaOccorrenze, mi serve per dire che se trovo che l'inizio del file è != da 0, quindi non parto da inizio file, vado avanti un byte alla volta fin quando non arrivo al carattere \n, questo perchè, se il mio file viene troncato bruscamente, da dove vado a leggere poi è probabile che la mia parola sia tagliata e il conteggio non sia poi corretto. Una conseguenza però a questo codice c'è, e verrà analizzata al passo seguente.

```
                fseek(in_file, s[j].end, SEEK_SET);
                ch = fgetc(in_file);
                while(ch!=&#39;\n&#39; && isalpha(ch)){
                    ch = fgetc(in_file);
                    s[j].end++;
                    flag=1;
                }
                if(flag==1){
                    s[j].end++;
                    flag=0;
                }
                fclose(in_file)
```

Se il file viene troncato

Cosa accadrebbe se il file venisse troncato e questa funzione non esisterebbe? Non verrebbe contata una parola (nel caso migliore!) questo mi ha portato a scrivere questa porzione di codice che prevede di andare ad incrementare, all'interno della struttura "SplitProcesso2" la end, ovvero la porzione finale del file. Questo perchè come da codice descritto precedentemente, se il un processo trova l'inizio del file che è diverso da 0, allora non conterà la prima parola in quanto potrebbe essere tagliata.

Note sulla compilazione

Per poter eseguire il codice in locale, basta fare un clone della repository, per poi andare a compilare con il seguente comando:

```
mpicc WordCountByte.c -o WordCountBye
```

per poi andare a eseguirlo con il seguente comando:

```
mpirun -np N WordCountByte >> output.txt
```

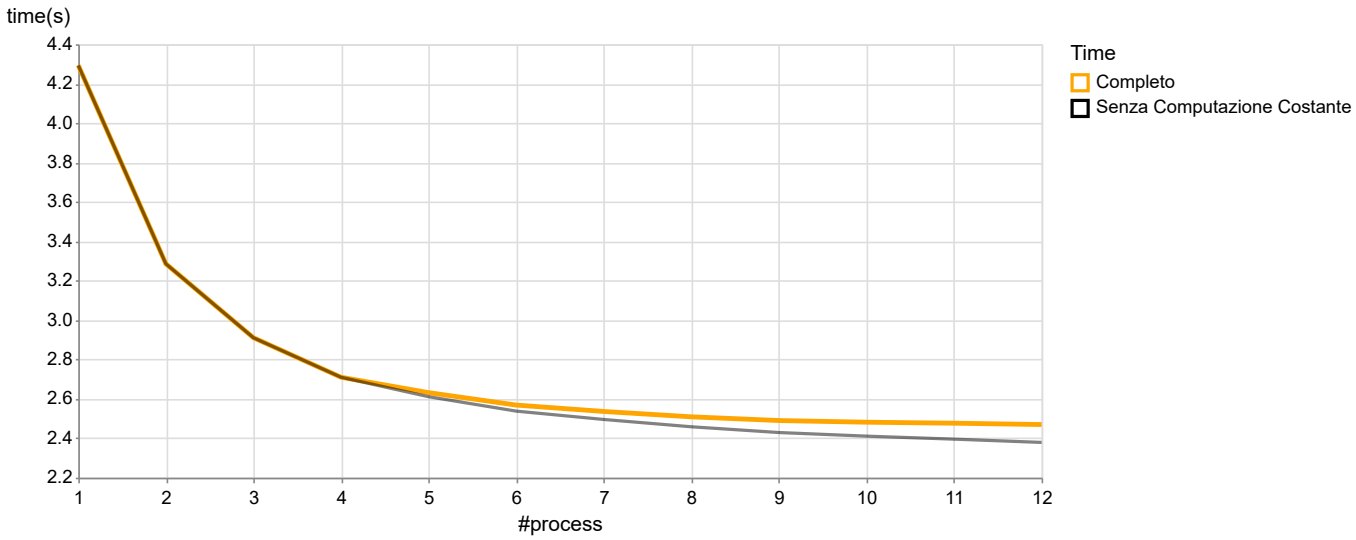
dove **N** è il numero di processi da utilizzare, mentre l'output è stato ridiretto a un file chiamato **"output.txt"**, questo per rendere più comoda la lettura dell'output ottenuto dal codice, in quanto al suo interno ho utilizzato dei comandi per prendere il tempo di esecuzione delle varie fasi del codice.

Note sull'implementazione

Per una corretta esecuzione del codice bisogna andare a modificare il numero di righe (#define row) all'interno del codice, questo perchè il c non permette di avere una consistenza di memoria tale da poter riallocare grandi sezioni di memoria dinamicamente senza introdurre caratteri non corretti. Anche il numero di colonne (#define cols) ovvero la massima grandezza della parola che posso avere, è dichiarata staticamente.

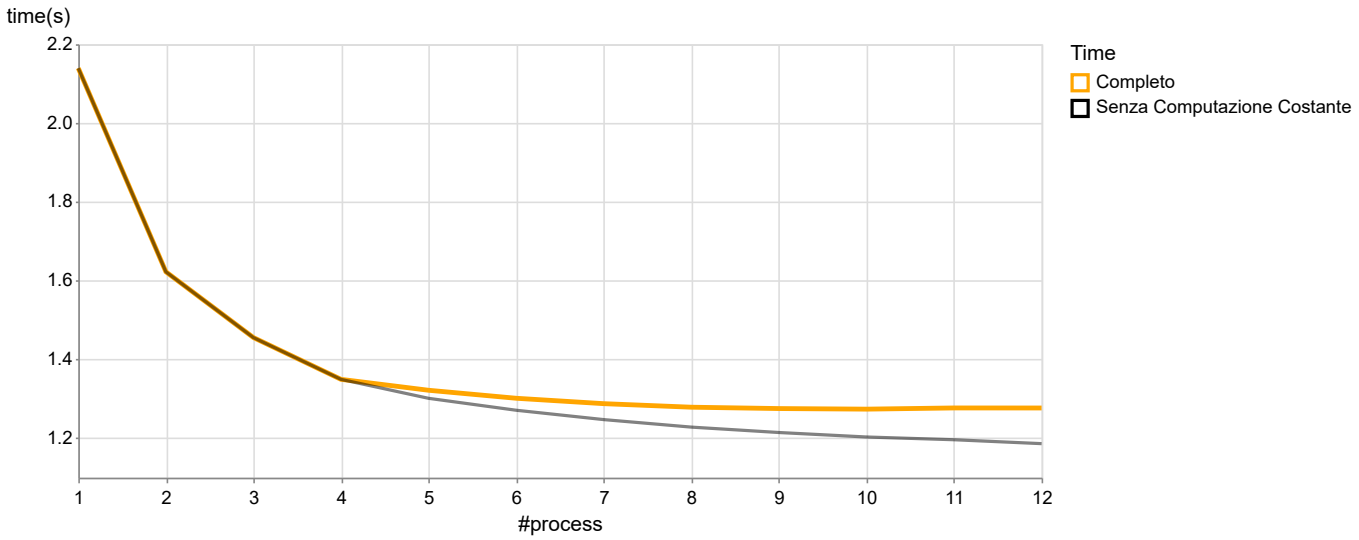
Risultati

400.000 parole



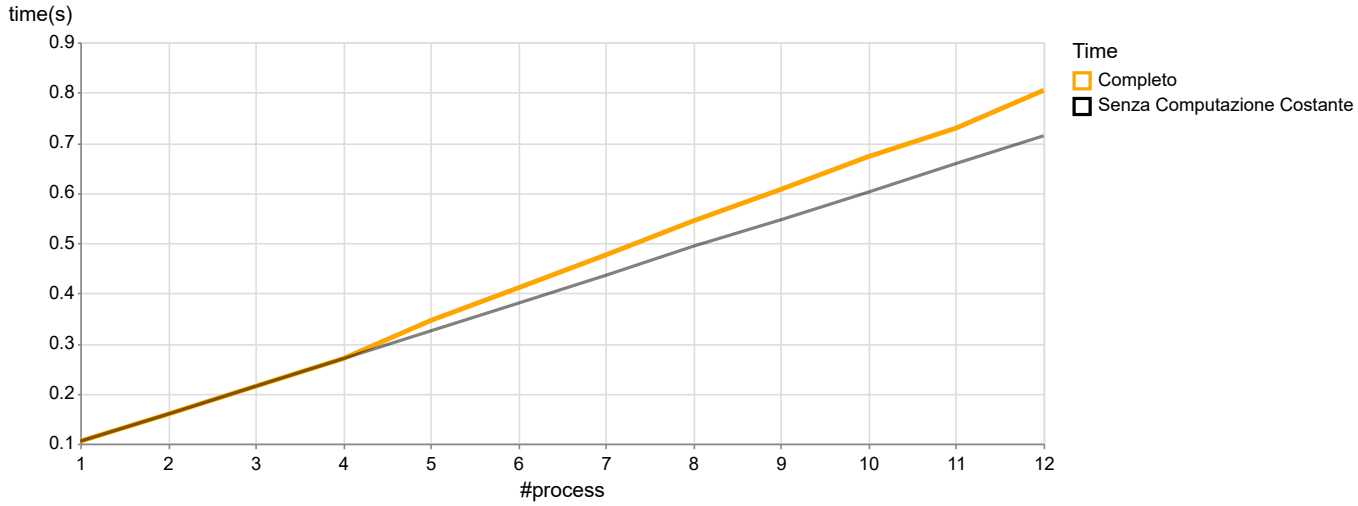
In questo grafico vi sono i tempi di esecuzione del codice da 1 processo a 12 processi, tenendo costante il numero di parole a 400.000 parole. Nel grafico sono rappresentate due linee che rappresentano (in nero) il tempo di esecuzione dell'algoritmo senza contare il tempo usato per la parte costante dello stesso, ovvero la parte che viene fatta a prescindere dai processi che comprende la lettura dei file e la suddivisione nella struttura, e le send ai vari processi; mentre la linea color arancio rappresenta il tempo totale per l'esecuzione dell'algoritmo, quindi aggiungendo la parte parallela, ovvero il tempo usato dal processo con rank = 0 per il calcolo delle occorrenze della sua parte, il tempo di attesa dei vari rank, e il calcolo totale delle occorrenze. Si può evincere dal grafico come il tempo impiegato dall'algoritmo cali drasticamente per i primi step, quindi dall'utilizzo di un solo processo fino a 4, successivamente si può vedere come la linea del tempo di esecuzione continui a calare, anche se con escursioni di tempo minori, ma costantemente fino al 12esimo processo. Si può inoltre notare come la componente costante inizi, dopo i 4 processi, a diventare sempre più invasiva, e porti ad un "rumore" sempre maggiore.

200.000 parole



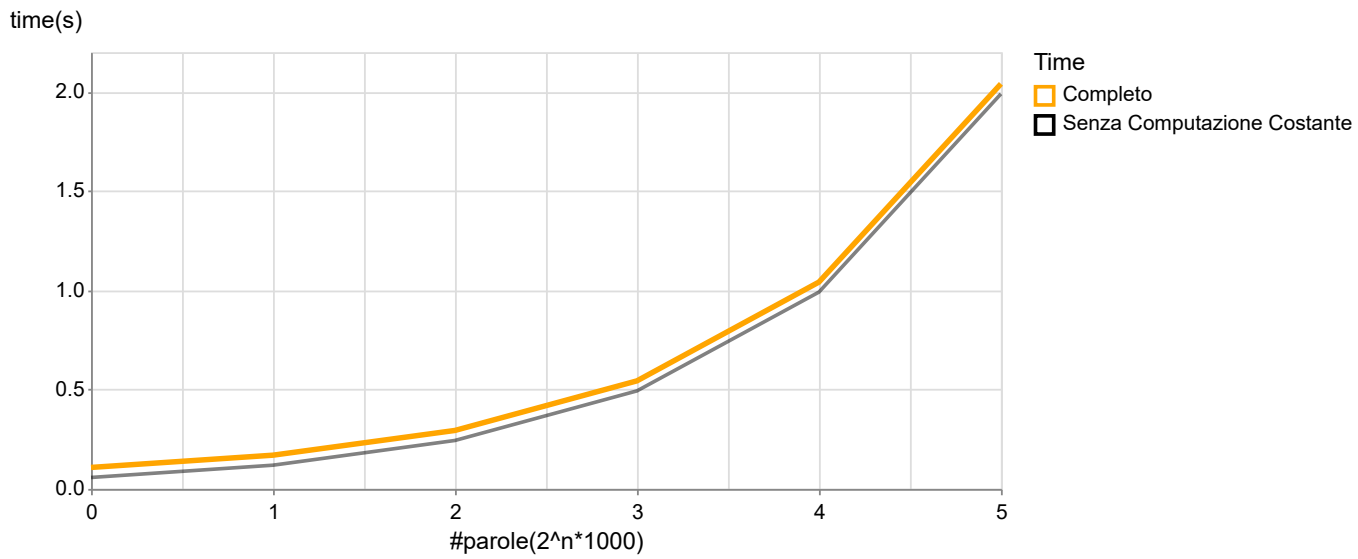
In questo grafico vi sono i tempi di esecuzione del codice da 1 processo a 12 processi, tenendo costante il numero di parole a 200.000 parole. La rappresentazione grafica è la stessa usata per il grafico precedente (per le 400.000 parole). Si può evincere dal grafico come il tempo impiegato dall'algoritmo cali drasticamente per i primi step, quindi dall'utilizzo di un solo processo fino a 4, successivamente si può vedere come la linea del tempo di esecuzione continui a calare, anche se con escursioni di tempo minori, ma costantemente fino al 12esimo processo. Si può inoltre notare come la componente costante inizi, dopo i 4 processi, a diventare sempre più invasiva, e porti ad un "rumore" sempre maggiore.

1	2	3	4	5	6	7	8	9	10	11	12
10k	20k	30k	40k	50k	60k	70k	80k	90k	100k	110k	120k



In questo grafico abbiamo una rappresentazione di esecuzione di tempo che si rifà a un test dove ho provveduto ad incrementare in maniera lineare il numero di parole e il numero di processsi (come da tabella soprastante). La rappresentazione grafica è la medesima dei precedenti, e si può evincere come il tempo di esecuzoine cresca in maniera lineare. Sempre invasiva la componente costante che, come da grafico, dopo il quarto processo, quindi dal quinto in poi, si faccia sempre più presente e invasiva portando a un aumento di tempo rispetto a ciò che potrebbe essere l'algoritmo senza di essa.

P	8	8	8	8	8	8
#parole	10k	20k	40k	80k	160k	320k



In questo grafico abbiamo una rappresentazione di esecuzione tenendo costante il numero di processore, ma andando ad aumentare, in potenze di 2, il numero di parole utilizzate, quindi partendo da 10.000 parole (2^0) fino ad arrivare a 320.000 parole (2^5). Si può notare da grafico come il tempo di esecuzione cresca in maniera esponenziale al crescere esponeziale del numero di parole così come dovrebbe essere, senza picchi di tempo di esecuzion,, e la parte costante invece rimanga semrpe costante nella sua "rumorosità", quindi non particolarmente incisiva ai fini della dimostrazione grafica.

Note sui Risultati

File usati per i test

Per i file di test sono stati formattati nel modo seguente:

parola1

parola2

parola3

.

.

.

Quindi ogni parola è "divisa" dall'altra con un semplice \n, quindi per suddividere le parole basta andare a capo. Mentre per la diversità delle parole usate sono state prende un set di 1000 parole, cosi per quando poi ho dovuto controllare la corretta conta delle occorrenze mi è stato più facile capire se fosse stata fatta in maniera corretta o meno.

Conclusioni

Dall'analisi dei grafici si può evincere come l'algoritmo si comporti bene, riesca a scalare correttamente senza avere picchi di complessità con un determinato numero di processi, o senza avere picchi di complessità con un determinato numero di parole, e dei risultati corretti in base alle parole date, sia per varietà che per numero di parole.