Progetto	Nome e Cognome	Data di consegna
WordCount	Mario Santoro	01/06/2020

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

```c
#include <stdio.h>
#include <mpi.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/dir.h>
#include <dirent.h>
#include <sys/param.h>
#include <time.h>
#define LENGTH 120000 //al massimo 120000 parole
#define LBYTE 1000
#define CHARLENGTH 20 //non esistono parole più lunghe di 20 caratteri
char PATHNAME[MAXPATHLEN];
//struttura che identifica la singola parola e la sua frequenza (o ricorrenza)
typedef struct Word
{
    char name[CHARLENGTH];
    int frequenza;
} Word;
//Struttura con il nome del file e la sua dimensione
typedef struct SizeByte
{
    char file[CHARLENGTH];
    long sizeByte;
} SizeByte;
/*struttura che per un dato rank identifica da quale byte partire(start)
 e dove finire (end) in un dato file(nameFile)*/
typedef struct ByteSplit
{
    int rank;
    char nameFile[CHARLENGTH];
    long start;
    long end;
} ByteSplit;

//funzione che crea il file csv con i risultati
void creaCSV(Word w[LENGTH], int size)
{
    FILE *fpcsv;
    fpcsv = fopen("/home/pcpc/risultati.csv", "w+"); //in locale solo risultati.csv
    fprintf(fpcsv, "OCCORRENZA,PAROLA");
    for (int t = 0; t < size; t++)
    {
        fprintf(fpcsv, "\n%s,%d", w[t].name, w[t].frequenza);
    }
    fclose(fpcsv);
}

/*funzione che calcola la frequenza delle parole in una porzione di array
(row è la dimensione dell'array) inserendo nella struttura Word la parola
```

```c
 e la sua frequenza infine restituisce la dimensione della struttura*/
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;
}
/*funzione che calcola in una struttura SizeByte la dimensione di ogni file
e restituisce la dimensione (di byte) totale per tutti i file*/
int 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;
}
```

```c
/*funzione che in input ha la taglia totale dei byte (di tutti i file) e la
suddivide equamente per ogni processo (p processi totali) in un array partitioning
(di long passato per argomento) in cui ogni posizione identifica quanti byte
dovrà "consumare" il processo i-esimo*/
void partitioning(long taglia, int p, long *partitioning)
{
    //dimensione array modulo numero di processi otteniamo il resto
    int resto = taglia % p;
    //inizializzo l'array con i valori (iniziali) uguali per tutti
    for (int i = 0; i < p; i++)
    {
        partitioning[i] = taglia / p;
    }
    //se il resto non è 0
    if (resto != 0)
    {

        int temp = 0;
        /*finchè il resto non è zero viene smistato il valore tra le varie porzioni*/
        while (resto != 0)
        {
            /*il tmp deve essere modulo p (se resto > di p si
             andrebbe in un elemento inesistente)*/
            partitioning[temp % p] = partitioning[temp % p] + 1;
            temp++;
            resto--;
        }
    }
}

/*funzione che prende in input:
- array di SizeByte la struttura contente il nome dei file e la loro dimensione;
- long taglia contente la dimensione totale (in Byte) di tutti i file;
- array di ByteSplit una struttura che verrà riempita con le informazioni necessarie
ai singoli processi per lo split dei byte tra i file divide equamente le parole
dell'array tra i processi;
- partitioning array di long contente la somma dei byte che ogni singolo processo deve
consumare;
- l'array di interi send è utile per calcolare quante celle della struttura ByteSplit
dovrà essere inviato a ogni singolo processo, poichè ogni cella identifica es:
index struttura 0 processo 0 "file1.txt" da byte 0 a 100, index struttura 1 processo
0 "file2.txt" da byte 0 a 30, quindi send[0] sarà uguale a 2*/
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;
}

/*completa e unifica i risultati di due struttura Word (parola e frequenza)
in un unica struttura: w2, restituisce la dimensione della struttura w2*/
int unisciResult(Word w1[LENGTH], int sizeW1, Word w2[LENGTH], int sizeW2)
{
    int bool = 0;

    for (int i = 0; i < sizeW1; i++)
    {
        for (int j = 0; j < sizeW2; j++)
        {
            /*se la parola della prima struttura è già presente nella seconda
            allora viene soltanto fatta l'addizione delle 2 frequenze*/
            if (strcmp(w1[i].name, w2[j].name) == 0)
            {
                w2[j].frequenza = w2[j].frequenza + w1[i].frequenza;
                bool = 1;
            }
        }
        /*se il flag è 0 (e la parola non deve essere vuota) la parola non è presente
         nella seconda struttura quindi viene aggiunta la parola
         e la frequenza trovata*/
        if (bool == 0 && strcmp(w1[i].name, "") != 0)
        {
            strcpy(w2[sizeW2].name, w1[i].name);
            w2[sizeW2].frequenza = w1[i].frequenza;
            sizeW2++;
        }
```

```
        bool = 0;
    }
    return sizeW2;
}
/*
funzione che prende in input:
- un array bidimensionale di caratteri che conterrà le parole che dovrà computare il
processo;
- ByteSplit la struttura che per un dato rank identifica da quale byte partire (start)
e dove finire (end) in un dato file(nameFile);
- start è da dove iniziare a scorrere l'indice dell'array, passato per argomento
poichè la funzione può essere rieseguita più volte dal processo(tante quante
sono i file da cui attingono i byte) e quindi poter aggiungere allo stesso array
altre parole per l'iterazione successiva.
Viene restituito lo stesso start che è incrementato durante la funzione e diventa la
nuova size dell'array
*/
int riempioArray(char array[LENGTH][CHARLENGTH], ByteSplit sp, int start)
{
    FILE *fp;
    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/");
    strcpy(path, PATHNAME);
    //concateno con il nome del file
    strcat(path, sp.nameFile);
    if ((fp = fopen(path, "rt")) == NULL)
    {
        printf("Errore nell'apertura del file'");
        exit(1);
    }
    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;
}

int main(int argc, char *argv[])
{
    int myRank, p;
    MPI_Status status;
    MPI_Request req;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    int tag = 0;
    //Creo la struttura ByteStruct come nuovo tipo MPI per passarlo con send e recive
    MPI_Datatype st, oldtypes[3];
    int blockcounts[3];
    MPI_Aint offsets[3];
    //definisco i valori della nuovo tipo MPI
```

```
    offsets[0] = offsetof(ByteSplit, rank);
    oldtypes[0] = MPI_INT;
    blockcounts[0] = 1;

    offsets[1] = offsetof(ByteSplit, nameFile);
    oldtypes[1] = MPI_CHAR;
    blockcounts[1] = 20;

    offsets[2] = offsetof(ByteSplit, start);
    oldtypes[2] = MPI_LONG;
    blockcounts[2] = 2;
    //per creare la struttura come nuovo tipo
    MPI_Type_create_struct(3, blockcounts, offsets, oldtypes, &st);
    MPI_Type_commit(&st);

    //Creo la struttura Word come nuovo tipo MPI per passarlo con send e recive
    MPI_Datatype w, oldtypes1[2];
    int blockcounts1[2];
    MPI_Aint offsets1[2];
    //definisco i valori del nuovo tipo MPI
    offsets1[0] = offsetof(Word, name);
    oldtypes1[0] = MPI_CHAR;
    blockcounts1[0] = 20;

    offsets1[1] = offsetof(Word, frequenza);
    oldtypes1[1] = MPI_INT;
    blockcounts1[1] = 1;
    //per inviare la struttura come nuovo tipo
    MPI_Type_create_struct(2, blockcounts1, offsets1, oldtypes1, &w);
    MPI_Type_commit(&w);

    SizeByte byte[LBYTE];
    ByteSplit sp[LBYTE];
    //avvio calcolo tempo
    clock_t begin = clock();
    if (myRank == 0)
    {
        /*"size" contiene la lunghezza totale in byte dei file, mentre
        in "byte" le informazioni: nome file e size in Byte*/
        long size = CalcolaByte(byte);
        long part[p];
        int row[p];
        //suddivisione della dimensione totale dei byte per i processi
        partitioning(size, p, part);

        //riempio la struttura ByteSplit sp con le informazioni da inviare ai processi
        int siz = splitByte(byte, size, sp, part, row);
        char array[LENGTH][CHARLENGTH];
        int index = row[0];
        for (int i = 1; i < p; i++)
        {
```

```
            //prima send che contiene la lunghezza della struttura in invio
            MPI_Send(&row[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];
        }
        int dim = 0;
        for (int i = 0; i < row[0]; i++)
        {
            /*riempio l'array tante volte quanti sono i file
            da cui il processo 0 deve attingere i byte*/
            dim = riempioArray(array, sp[i], dim);
        }
        Word words[LENGTH];
        Word tmp[LENGTH];
        /*calcola la frequenza delle parole contenute
        nella porzione di array assegnatogli*/
        int sizeStructWord = calcolaFrequenza(array, dim, words);
        for (int i = 1; i < p; i++)
        {
            int sizeRecive;
            //ricevo la dimensione della struttura in arrivo alla recive successiva
            MPI_Recv(&sizeRecive, 1, MPI_INT, i, tag, MPI_COMM_WORLD, &status);
            //ricevo la struttura calcolata da ciuascun processo
            MPI_Recv(&tmp, sizeRecive, w, i, tag + 1, MPI_COMM_WORLD, &status);
            /*unisco i risultati mandati dal processo i con i risultati globali
             contenuti nella struttura del processo 0*/
            sizeStructWord = unisciResult(tmp, sizeRecive, words, sizeStructWord);
        }
        //creao file csv
        creaCSV(words, sizeStructWord);
        //calcolo e stampo il tempo di computazione
        clock_t end = clock();
        double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
        printf("execution time = %lf\n", time_spent);
    }
    else
    {
        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);
        char array[LENGTH][CHARLENGTH];
        int index = 0;
        for (int i = 0; i < siz; i++)
        {
            /*riempio l'array tante volte quanti sono i file da cui
             il processo corrente deve attingere i byte*/
```

```
            index = riempioArray(array, sp[i], index);
        }
        Word words[LENGTH];
        /*calcola la frequenza delle parole contenute nella
        porzione di array inviatogli*/
        int size2 = calcolaFrequenza(array, index, words);
        /*mando la dimensione della struttura
        che verrà inviata alla prossima send*/
        MPI_Send(&size2, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
        /*mando la struttura contenente le parole e
        la frequenza calcolate dal processo corrente*/
        MPI_Send(&words, size2, w, 0, tag + 1, MPI_COMM_WORLD);
    }
    //libero la memoria e chiudo i processi
    MPI_Type_free(&st);
    MPI_Type_free(&w);
    MPI_Finalize();
    return 0;
}
```

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 in evidenza dell'algoritmo sono spiegate nei commenti nel codice soprastante.

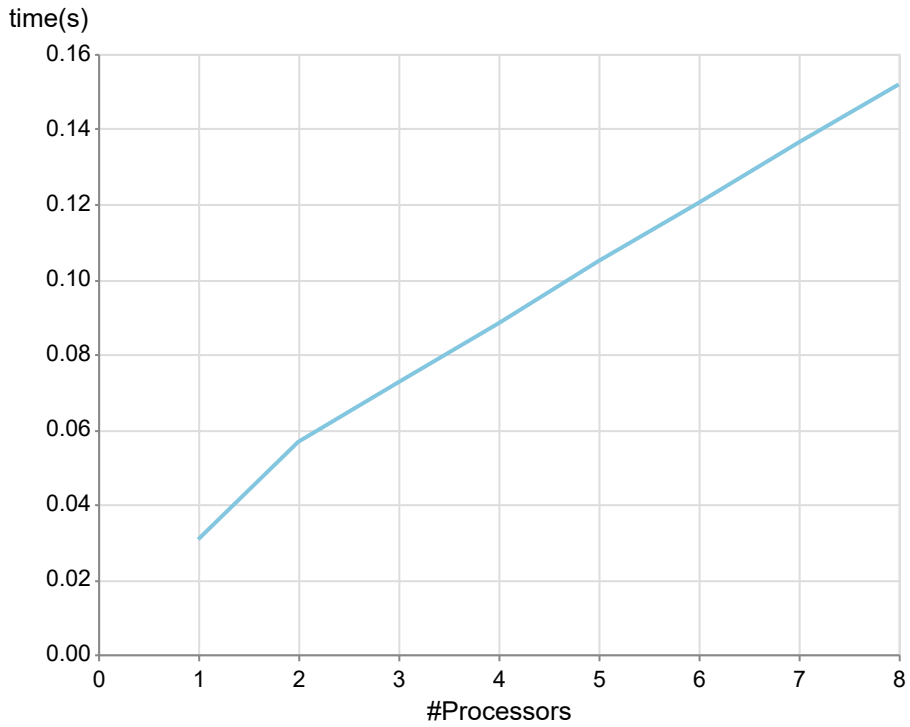
Nei file le parole sono seguite da '\n' per semplificare la lettura nell'implementazione.

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 utilizzando la gestione della memoria dinamica, e quindi puntatori, essa creava problemi nell'utilizzo delle send e recive.

Risultati

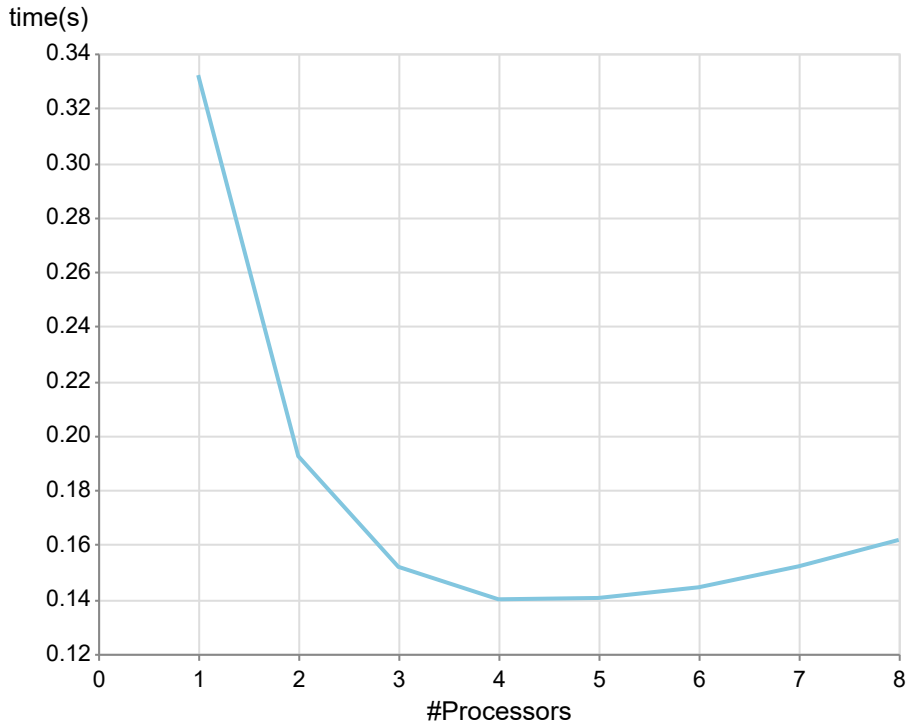
Scalabilità forte

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.