MPI Word Count

Istruzioni per l'esecuzione

```
# Nella directory principale del progetto
# Esecuzione con la directory "input" come argomento
make wordCount
mpirun -allow-run-as-root -np [2-24] -oversubscribe src/./wordCount.out input
make clean
```

Problema

MPI Word Count è un programma per il conteggio parallelo delle occorrenze delle parole all'interno dei file.

Il programma prende in input il path di una directory e produce in output un file con estensione .csv contenente tutte le parole trovate e le relative occorrenze in ordine decrescente.

Esecuzione in 10 punti

L'esecuzione del programma è così riassumibile:

1. inizializzazione e controlli sull'input e sul numero di processi;
2. creazione dei datatype per le parole e le porzioni dei file;
3. creazione della lista dei file presenti nella directory fornita in input;
4. suddivisione dei file tra i processi;
5. elaborazione dei task assegnati;
6. ricezione dei risultati;
7. aggregazione dei risultati;
8. ordinamento delle parole in base alle occorrenze trovate;
9. generazione del file .csv;
10. distruzione dei datatype.

Codice

La soluzione implementata prevede l'utilizzo di:

- MPI Datatypes per parole e porzioni dei file;
- liste per la gestione dei file;
- liste per la gesione delle parole.

Strutture dati

Analizziamo nel dettaglio le strutture dati utilizzate:

```
typedef struct filePart
{
    double startPoint;
    double endPoint;
    char filePath[300];
} filePart;
```

> N.B. La lunghezza massima del path è basata su un arrotondamento per eccesso della massima lunghezza del nome di un file nei principali filesystem (Fonte: Wikipedia).

```
typedef struct word
{
    int occurrencies;
    char text[50];
} word;
```

> N.B. La lunghezza massima delle parole è basata su un arrotondamento per eccesso della lunghezza della quarta parola più lunga per la lingua inglese (Fonte: Wikipedia).

Le precedenti strutture dati, presenti rispettivamente nei file *fileManagement.h* e *wordManagement.h* sono state utilizzate rispettivamente per l'invio dei job ai processi e per l'invio dei risultati elaborati al processo master.

```
typedef struct fileNode
{
    char *path;
    int size;
    struct fileNode *next;
} fileNode;

typedef struct linkedFileList
{
    int size;
    struct fileNode *head;
} linkedFileList;
```

```
typedef struct node
{
    char *text;
    int occurrencies;
    struct node *next;
```

```
} node;

typedef struct linkedList
{
    int size;
    node *head;
} linkedList;
```

Le precedenti strutture definiscono rispettivamente le liste per i file e per le parole, presenti nei file *linkedList.h* e *linkedFileList.h*.

Analisi della soluzione

Analizziamo nel dettaglio ogni punto della lisa precedente.

1-2: inizializzazione, controlli e creazione dei datatypes

```
MPI_Init(&argc, &argv);
int tasks, myrank;
MPI_Status status;
MPI_Comm_size(MPI_COMM_WORLD, &tasks);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Datatype filePartDatatype, wordDatatype;
newFilePartDatatype(&filePartDatatype);
newWordDatatype(&wordDatatype);

if (argc < 2 || tasks < 2)
{
    if(myrank == MASTER)
    printf("Usage: mpirun -allow-run-as-root -np [2-24] -oversubscribe ./%s
\"/path/to/folder\"\n", argv[0]);
    MPI_Finalize();
    return 0;
}
```

Una volta inizializzato MPI e creati i datatype, viene controllato se sia stata fornito un path in input e se i processi sono almeno 2. In caso di esito negativo, il master stampa le istruzioni per l'esecuzione e il programma viene arrestato.

3: creazione della lista dei file

```
//File: main.c
double allFilesSize = 0;
linkedFileList *fileList = createFileList(argv[1], &allFilesSize);

//File: fileManagement.c
linkedFileList *createFileList(char *filePaths, double *allFileSize)
{
```

```
    linkedFileList *fileList = newFileList();
    DIR *dir;
    struct dirent *dirent;
    char absolutePath[300];
    char sep1 = '/', sep2 = '\0';

    dir = opendir(filePaths);
    if (dir != NULL)
    {
        while ((dirent = readdir(dir)) != NULL)
        {
            if (dirent->d_type != DT_DIR)
            {
                //Creo il path assoluto del file
                [...]

                fileNode *info = createFileNode(absolutePath);
                *allFileSize += info->size;
                addFileListEntry(fileList, info);
            }
        }
        closedir(dir);
        //Controllo se ci sono errori
        [...]
    }
    return fileList;
}
```

Una volta aperta la directory, si compone il path assoluto per ogni file e si creano i relativi nodi da inserire nella lista.

4: suddivisione dei file tra i processi

```
int portion = allFilesSize / tasks;
int rest = (int)allFilesSize % tasks;

//Creo alcune variabili necessarie per la suddivisione
[...]

MPI_Request *requests;
MPI_Alloc_mem(sizeof(MPI_Request) * (tasks - 1), MPI_INFO_NULL, &requests);
MPI_Alloc_mem(sizeof(filePart) * foundFiles, MPI_INFO_NULL, &(jobs[currentTask]));

fileNode *n = fileList->head;
for (int i = 0; i < foundFiles; i++)
{
    assignedBytes = 0;
    notAssignedBytes = n->size;

    while (notAssignedBytes > 0)
    {
```

```
        if (bytesToAssign == 0)
        { // Se non ho altri byte da assegnare al task corrente...
            if (filePartCounter != 0)
            { //... se ho almeno una parte di file che deve essere inviata...
                if (currentTask != MASTER)
                { //... e se il processo per cui sto calcolando cosa inviare non è
il MASTER, allora invio.
                    MPI_Isend(jobs[currentTask], filePartCounter,
filePartDatatype, currentTask, 0, MPI_COMM_WORLD, &(requests[currentTask - 1]));
                }
                else
                { // Altrimenti salvo il numero di parti da analizzare
                    masterFilePartCount = filePartCounter;
                }
                //Ripristino le variabili e alloco lo spazio per il job successivo
                [...]
            }
        }

        //Assegno tutto il file o una porzione di esso
        [...]
    }
}
```

Una volta calcolati i byte da assegnare ai processi, per ogni file si verifica se la dimensione di quest'ultimo sia minore o uguale dei byte da assegnare. In caso di esito positivo, si assegna il file, altrimenti se ne assegna una porzione di dimensione pari ai byte rimanenti.

Non appena i byte disponibili per il processo si esauriscono, se sono state assegnate parti viene effettuato l'invio o, nel caso si tratti del master, viene salvato il numero di porzioni assegnate.

5: elaborazione dei task

Master

```
//File: main.c
//Elaborazione del master
int masterCountedWords = 0;
filePart *masterFilePart = jobs[MASTER];
word *masterWordArr = getWordOccurrencies(masterFilePart, masterFilePartCount,
&masterCountedWords);

//Resto del programma
[...]

//Elaborazione degli slave
int partNum = 0, countedWords = 0;
filePart *fileList = checkMessage(filePartDatatype, &partNum, status);
word *wordArr = getWordOccurrencies(fileList, partNum, &countedWords);
MPI_Send(wordArr, countedWords, wordDatatype, MASTER, 0, MPI_COMM_WORLD);
```

```
//File: wordManagement.c
word *getWordOccurrencies(filePart *parts, int partNum, int *countedWords)
{
    //Creo le variabili necessarie e una lista
    [...]

    for (int i = 0; i < partNum; i++)
    {
        //Apro il file e verifico se mi trovo all'inizio, se ci sono parole
troncate e, in tal caso, leggo
        [...]

        int prevCharIsWordTerminator = 0;
        while (ftell(file) < (fp.endPoint))
        {
            currWord[wordPtr] = fgetc(file);

            if ((isalpha(currWord[wordPtr]) || isdigit(currWord[wordPtr])) &&
!prevCharIsWordTerminator)
            {
                wordPtr++;
            }
            else if (isWordTerminator(currWord[wordPtr]) &&
!prevCharIsWordTerminator)
            {
                currWord[wordPtr] = '\0';

                for (int n = 0; currWord[n]; n++)
                {
                    currWord[n] = tolower(currWord[n]);
                }

                addOccurrency(list, currWord, &wordCount);
                wordPtr = 0;
                prevCharIsWordTerminator = 1;
            }
            else if (isWordTerminator(currWord[wordPtr]) &&
prevCharIsWordTerminator)
            {
            } // Ignoro i doppi terminatori
            else if ((isalpha(currWord[wordPtr]) || isdigit(currWord[wordPtr])) &&
prevCharIsWordTerminator)
            {
                wordPtr++;
                prevCharIsWordTerminator = 0;
            }
        }
    }
    wordArr = getWordArrayFromList(list, countedWords);
    return wordArr;
}
```

L'elaborazione dei task prevede come output un array di parole ed occorrenze, nonchè il numero di parole diverse trovate.

Analizziamo il protocollo implementato per il word counting, supponendo che ci siano due processi A e B che lavorano su porzioni diverse dello stesso file e che ci sia una parola "spezzata" tra le due porzioni:

- Il processo A analizza la propria porzione fino alla fine. Dato che l'ultima parola è incompleta (non trova un terminatore), allora non la inserisce nell'array.
- Il processo B verifica di non trovarsi ad inizio file e che il carattere precedente al byte da cui iniziare a leggere è una lettera o un numero. B procede a leggere all'indietro fino al primo terminatore, dopodichè comincia a leggere ed inserire le parole nel suo array. In questo modo, B riesce a leggere la parola "spezzata" tra le due porzioni.

6-7: ricezione e aggregazione dei risultati

```
//File: main.c
if (i != MASTER)
{
    MPI_Probe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &recvStatus);
    MPI_Get_count(&recvStatus, wordDatatype, &count);
    MPI_Alloc_mem(sizeof(word) * count, MPI_INFO_NULL, &wordArr);
    MPI_Recv(wordArr, count, wordDatatype, recvStatus.MPI_SOURCE, 0,
MPI_COMM_WORLD, &recvStatus);
    updateMasterList(list, wordArr, count);
}
else
{
    updateMasterList(list, masterWordArr, masterCountedWords);
    MPI_Free_mem(masterWordArr);
}

//File: wordManagement.c
void updateMasterList(linkedList *list, word *wordArr, int wordNum)
{
    for (int i = 0; i < wordNum; i++)
    {
        updateListEntry(list, wordArr[i].text, wordArr[i].occurrencies);
    }
}
```

8-9-10: ordinamento delle parole, generazione del file e distruzione dei datatype

```
//File: main.c
word *orderedWordArr = getWordArrayFromList(list, &wordArrLength);
sortByCount(orderedWordArr, 0, wordArrLength);
printOutputCSV(orderedWordArr, wordArrLength);
```

```
for (int i = 0; i < tasks - 1; i++)
{
    MPI_Free_mem(jobs[i]);
}
MPI_Free_mem(jobs);

freeFileList(fileList);
free(orderedWordArr);
MPI_Free_mem(wordArr);

//Codice degli slave
[...]

MPI_Type_free(&filePartDatatype);
MPI_Type_free(&wordDatatype);
MPI_Finalize();

//File: wordManagement.c
word *getWordArrayFromList(linkedList *list, int *wordArrLength)
{
    word *wordArr;
    MPI_Alloc_mem(sizeof(word) * list->size, MPI_INFO_NULL, &wordArr);
    *wordArrLength = list->size;

    int i = 0;
    for (node *n = list->head; n != NULL; n = n->next, i++)
    {
        strncpy(wordArr[i].text, n->text, 50);
        wordArr[i].occurrencies = n->occurrencies;
    }

    return wordArr;
}

void printOutputCSV(word *wordArr, int wordArrLength)
{
    FILE *csvFile = fopen("results.csv", "w+");
    fprintf(csvFile, "Word, Occurrencies\n");
    for (int i = (wordArrLength - 1); i >= 0; i--)
    {
        fprintf(csvFile, "%s, %d\n", wordArr[i].text, wordArr[i].occurrencies);
    }
    char cwd[300];
    getcwd(cwd, sizeof(cwd));
    printf("File %s/results.csv generated successfully!\n", cwd);
}
```

L'ordinamento delle parole avviene attraverso l'algoritmo Merge Sort su un array generato a partire dalla lista in cui sono stati aggregati i risultati.

Le parole vengono ordinate in ordine crescente, quindi per ottenere un ordine decrescente all'interno del file, basta scrivere le parole in ordine inverso all'interno del file.