UNIVERSITÀ DEGLI STUDI DI MESSINA

Dipartimento di Scienze Matematiche e Informatiche,
Scienze Fisiche e Scienze della Terra

CORSO DI LAUREA TRIENNALE IN INFORMATICA L-31

LABORATORIO DI RETI E SISTEMI DISTRIBUITI

Relazione Progetto

Progettazione di un sistema MapReduce per il WordCount in C

Gabriele Franchina 547619

Lorenzo Pio Virecci Fano 547047

Sommario

Capitolo 1 - Problem Statement

Introduzione al problema

L'elaborazione di grandi dataset testuali richiede architetture capaci di gestire in maniera efficiente il calcolo distribuito e il parallelismo, garantendo risultati corretti e tempi di esecuzione contenuti. In questo contesto, il paradigma MapReduce rappresenta un approccio consolidato per suddividere e aggregare il lavoro in maniera scalabile, consentendo di sfruttare più processi o macchine per l'elaborazione dei dati.

Descrizione del problema e obbiettivi finali

Il problema affrontato consiste nel calcolare il conteggio totale delle parole di un testo molto lungo come "Il Signore degli Anelli", in maniera efficiente e parallelizzata, simulando un ambiente distribuito su una singola macchina.

La scelta del linguaggio C, pur richiedendo una gestione più attenta della memoria e dei processi rispetto a linguaggi di più alto livello, permette un controllo granulare sulle risorse di sistema e prestazioni ottimali.

L'architettura del sistema è basata su una suddivisione in più processi cooperanti:

- Master: un unico processo coordinatore del sistema e responsabile della lettura del file di input e della distribuzione dei dati;
- Mapper: più processi incaricati dell'elaborazione dei chunk di testo;
- Reducer: un processo dedicato all'aggregazione finale dei risultati.

La comunicazione tra i processi avviene tramite meccanismi di Inter-Process Communication (IPC) su ambiente locale, mentre il Reducer adotta un modello multithreaded con utilizzo di mutex per garantire la corretta sincronizzazione nell'accesso concorrente alle strutture dati condivise.

L'obiettivo principale è dimostrare la realizzabilità di un framework MapReduce semplificato in C, analizzandone il comportamento in termini di parallelismo, correttezza e prestazioni al variare del numero di Mapper e delle condizioni di esecuzione.

Capitolo 2 - Stato dell'arte

Paradigma di MapReduce

MapReduce è un paradigma di programmazione distribuita introdotto da Google nel 2004 per elaborare grandi dataset in modo scalabile e fault-tolerant. La sua architettura può esser suddivisa in 3 fasi fondamentali:

- Fase di Map: si occupa dell'elaborazione parallela, distribuendo il carico su più worker paralleli. Nello specifico, processa un sottoinsieme di dati e genera coppie (chiave, valore).
- Fase di Shuffle&sort: in questa fase vengono aggregati tutte le coppie chiave valore per chiave in modo tale da raggruppare insieme tutti i valori associati alla stessa chiave.
- Fase di Reduce: aggrega tutte le coppie con la stessa chiave per produrre il risultato finale.

Nello specifico, MapReduce si suddivide in due funzioni fondamentali: la funzione Map che trasforma l'input in coppie (chiave, valore) in modo indipendente, permettendo parallelismo tra più processi e la funzione Reduce che raccoglie tutte le coppie dai Mapper, raggruppa per chiave e calcola i valori aggregati. Per il WordCount, il Reduce somma i valori di ciascuna parola per ottenere il conteggio totale.

Vantaggi e limitazioni

MapReduce è utilizzato per l'elaborazione di grandi dataset, come l'analisi di testi e log, l'indicizzazione di pagine web e il calcolo di statistiche e aggregazioni.

I principali vantaggi di MapReduce includono:

- Scalabilità: possibilità di aumentare il numero di Mapper e Reducer
- Parallelismo: esecuzione contemporanea dei Mapper su chunk separati
- Semplicità concettuale: il programmatore si concentra solo su Map e Reduce

Tuttavia, anche tale paradigma soffre di alcune limitazioni. Tra queste troviamo:

- Overhead di comunicazione tra Master, Mapper e Reducer
- Necessità di strutture dati efficienti per il Reduce
- Implementazioni su singola macchina non sfruttano pienamente il potenziale di distribuzione

Capitolo 3 - Metodologia ed implementazione

Panoramica architetturale

Il progetto è stato sviluppato implementando un framework MapReduce semplificato in linguaggio C, eseguibile su un'unica macchina tramite localhost. L'obiettivo principale è calcolare il conteggio delle parole nel testo "Il Signore degli Anelli", suddividendo l'elaborazione in più processi concorrenti (Mapper) e aggregando i risultati in un unico processo Reducer, sotto il coordinamento di un processo Master.

L'architettura del sistema prevede tre tipologie di processi:

- Master – Coordinatore principale, responsabile della suddivisione del file di input, dell'avvio dei Mapper e del Reducer, e della gestione della comunicazione iniziale.
- Mapper – Processi worker che leggono porzioni del file di testo, calcolano il conteggio locale delle parole e inviano i risultati al Reducer.
- Reducer – Processo che raccoglie i risultati da tutti i Mapper, aggrega i conteggi per ciascuna parola e produce l'output finale.

La comunicazione tra i processi avviene tramite meccanismi di Inter-Process Communication (IPC) in ambiente locale, consentendo di simulare il parallelismo e il modello distribuito tipici di MapReduce pur operando su una singola macchina.

Configurazione del progetto

Il file di input utilizzato nel progetto è lotr.txt, contenuto nella cartella data/. Questo file rappresenta il testo del "Il Signore degli Anelli" e costituisce il dataset su cui viene eseguito il calcolo del WordCount, permettendo di verificare l'efficienza e la correttezza del framework MapReduce implementato.

Il progetto è organizzato in quattro componenti principali contenuti nella cartella src/, ognuno con un ruolo specifico nella pipeline MapReduce:

Common.h

common.h contiene tutte le definizioni condivise tra i vari componenti del progetto. Le principali responsabilità del file `common.h` sono:

- definire il numero di processi Mapper tramite la costante `NUM_MAPPERS`;
- specificare i percorsi delle FIFO utilizzate per la comunicazione inter-process, collocate nella directory `/tmp`;
- dichiarare costanti di utilità, come la lunghezza massima di una parola;
- definire la struttura dati utilizzata per rappresentare i chunk ottenuti dalla suddivisione del file di input.

```
#ifndef COMMON_H
#define COMMON_H

#define NUM_MAPPERS 4

#define MASTER_TO_MAPPER_FIFO "/tmp/master_to_mapper_%d"
#define MAPPER_TO_REDUCER_FIFO "/tmp/mapper_%d_to_reducer"

#define MAX_WORD_LEN 64
#define MAX_LINE_LEN 1024

typedef struct {
    long offset;
    long size;
} ChunkData;

#endif
```

Questo file permette di modificare facilmente la configurazione del sistema senza dover intervenire su tutti i file sorgente.

MASTER.C

Il processo *Master* rappresenta il coordinatore principale del framework. Le sue responsabilità fondamentali sono:

- calcolare la suddivisione del file `lotr.txt` in chunk di dimensioni equivalenti, assegnandone uno a ciascun Mapper;
- avviare i processi Mapper e Reducer mediante le primitive di sistema `fork` ed `exec`;
- trasmettere a ogni Mapper le informazioni relative al chunk assegnato (offset e dimensione);
- gestire la sincronizzazione complessiva dei processi, attendendo il completamento dell'elaborazione.

In questa architettura, il Master svolge esclusivamente funzioni di coordinamento e orchestrazione, senza intervenire direttamente sull'elaborazione del contenuto del file di input.

Avvio del processo Reducer

```
pid_t pid = fork();
if (pid == 0) {
    execl("./reducer", "reducer", NULL);
    perror("execl reducer");
    exit(EXIT_FAILURE);
    }
```

Nel framework MapReduce, il Reducer deve essere attivo prima dei Mapper, poiché è il componente incaricato di raccogliere i risultati parziali prodotti da questi ultimi.

Nello specifico, il Master crea un nuovo processo tramite `fork()` e, nel processo figlio, sostituisce l'immagine di processo con il programma reducer tramite `execl()`. Il Reducer viene così eseguito come processo indipendente, pronto a ricevere i dati provenienti dai Mapper attraverso meccanismi di comunicazione interprocesso basati su FIFO (named pipes).

Questa scelta consente di sincronizzare correttamente il flusso di esecuzione: i Mapper possono inviare i propri risultati solo quando il Reducer è già in ascolto, evitando perdite di dati e garantendo la corretta aggregazione finale.

Avvio dei processi Mapper

```
for (int i = 0; i < NUM_MAPPERS; i++) {
    pid = fork();
    if (pid == 0) {
        char id[2];
        sprintf(id, "%d", i);
        execl("./mapper", "mapper", id, NULL);
        perror("execl mapper");
        exit(EXIT_FAILURE);
    }
}
```

I *Mapper* rappresentano i worker paralleli del sistema MapReduce: ciascun Mapper elabora una porzione distinta del file di input, permettendo di sfruttare il parallelismo a livello di processo.

Il processo Master crea un numero di processi pari a `NUM_MAPPERS`. A ogni Mapper viene passato un identificativo numerico (*id*) come argomento della riga di comando, che consente di determinare le FIFO di comunicazione da utilizzare e di ricevere successivamente dal Master le informazioni relative al *chunk* di file da elaborare (offset e dimensione).

In questo modo, ciascun Mapper opera in modo indipendente sugli stessi dati di input, producendo un conteggio locale delle parole che viene poi inviato al Reducer tramite comunicazione interprocesso basata su FIFO, contribuendo all'aggregazione finale dei risultati.

Distribuzione dei chunk

```
for (int i = 0; i < NUM_MAPPERS; i++) {
snprintf(fifo_path, sizeof(fifo_path), MASTER_TO_MAPPER_FIFO, i);

int fifo_fd = open(fifo_path, O_WRONLY);
if (fifo_fd < 0) {
    perror("open master_to_mapper FIFO");
    continue;
}

    ChunkData chunk;
    chunk.offset = i * chunk_size;
    chunk.size = (i == NUM_MAPPERS - 1)
                    ? (file_size - chunk.offset)
                    : chunk_size;

    ssize_t bytes_written = write(fifo_fd, &chunk, sizeof(ChunkData));
    if (bytes_written != sizeof(ChunkData)) {
        perror("write chunk to FIFO");
    }

    close(fifo_fd);

    if (i == 0) printf("\n");
```

```
    printf("[Master] Inviato chunk %d: offset= %ld, size= %ld\n", i,
    ↪  chunk.offset, chunk.size);
    }

printf("\n");
```

La fase di distribuzione dei dati rappresenta un elemento chiave del paradigma MapReduce. Il processo Master non invia direttamente il contenuto del file, ma solamente le informazioni necessarie per accedervi, evitando duplicazioni inutili dei dati e riducendo l'overhead di comunicazione.

Per ciascun Mapper, il Master calcola l'offset iniziale del *chunk* e ne determina la dimensione (assegnando all'ultimo Mapper eventuali byte residui). Tali informazioni vengono quindi trasmesse attraverso meccanismi di comunicazione interprocesso basati su FIFO, consentendo a ogni Mapper di conoscere con precisione la porzione di file da elaborare.

In questo modo, ciascun Mapper legge esclusivamente il segmento di input assegnato, senza sovrapposizioni con gli altri processi, garantendo una suddivisione corretta del lavoro e preservando la coerenza del conteggio finale delle parole.

Sincronizzazione e terminazione

```
for (int i = 0; i < NUM_MAPPERS + 1; i++) {
        wait(NULL);
    }

    cleanup_fifos();

    printf("[Master] Esecuzione completata\n");
    return 0;
}
```

Un corretto coordinamento dei processi è essenziale nei sistemi distribuiti. Il Master deve assicurarsi che tutti i Mapper e il Reducer abbiano terminato prima di concludere l'esecuzione.

Il Master utilizza `wait()` per attendere la terminazione di tutti i processi figli (Mapper + Reducer). Solo dopo la loro conclusione viene stampato un messaggio finale che segnala il completamento dell'intero job MapReduce.

MAPPER.C

Il Mapper rappresenta il componente di elaborazione parallela del framework MapReduce. Il suo compito è leggere una porzione del file di input assegnata dal Master, eseguire il calcolo locale del WordCount e inviare i risultati parziali al Reducer.

Ogni Mapper opera in maniera indipendente, consentendo di sfruttare il parallelismo a livello di processo.

Struttura dati per il WordCount locale

```c
typedef struct WordCount {
    char word[MAX_WORD_LEN];
    int count;
    struct WordCount *next;
} WordCount;

WordCount *head = NULL;
```

Durante la fase Map, ogni Mapper deve mantenere un conteggio locale delle parole incontrate, quindi è necessaria una struttura dati che consenta di memorizzare le coppie (parola, conteggio) e aggiornarle dinamicamente.

Il Mapper utilizza una lista collegata (linked list), dove ogni nodo rappresenta una parola distinta e il numero di occorrenze trovate nel chunk assegnato. La scelta di una lista collegata garantisce semplicità di implementazione e flessibilità nella gestione dinamica delle parole.

Ricerca e aggiornamento delle parole

```c
WordCount *find_word(const char *word) {
    WordCount *curr = head;
    while (curr) {
        if (strcmp(curr->word, word) == 0)
            return curr;
        curr = curr->next;
    }
    return NULL;
}

void add_word(const char *word) {
    WordCount *wc = find_word(word);
    if (wc) {
        wc->count++;
    } else {
        wc = malloc(sizeof(WordCount));
        strncpy(wc->word, word, MAX_WORD_LEN);
        wc->word[MAX_WORD_LEN - 1] = '\0';
        wc->count = 1;
        wc->next = head;
        head = wc;
    }
```

```
}
```

Per implementare il WordCount, il Mapper deve essere in grado di verificare se una parola è già presente nel conteggio locale ed valutare se incrementare il contatore o inserirla come nuova.

La funzione `find_word` scorre la lista collegata alla ricerca di una parola già presente. La funzione `add_word` utilizza `finword` per determinare se incrementare il contatore o creare un nuovo nodo. Questo approccio garantisce la correttezza del conteggio locale prima della fase di Reduce.

Normalizzazione delle parole

```c
void normalize_buffer(char *buffer) {
    for (int i = 0; buffer[i]; i++) {
        if (!isalnum((unsigned char)buffer[i]))
            buffer[i] = ' ';
        else
            buffer[i] = tolower(buffer[i]);
    }
}
```

Nel WordCount è fondamentale trattare parole in equo modo indipendentemente da maiuscole/minuscole, punteggiatura.

La funzione `normalize` converte tutti i caratteri in minuscolo e rimuove i caratteri non alfanumerici, garantendo uniformità tra parole indipendentemente dalle loro caratteristiche di scrittura.

Ricezione e lettura del chunk

```c
    snprintf(fifo_path, sizeof(fifo_path), MASTER_TO_MAPPER_FIFO,
    ↪  mapper_id);

    int fifo_fd = open(fifo_path, O_RDONLY);
    if (fifo_fd < 0) {
        perror("open master_to_mapper FIFO");
        exit(EXIT_FAILURE);
    }

    ChunkData chunk;
    if (read(fifo_fd, &chunk, sizeof(ChunkData)) != sizeof(ChunkData)) {
        perror("read chunk");
        close(fifo_fd);
        exit(EXIT_FAILURE);
    }
    close(fifo_fd);

    while (ftell(fp) < chunk_end && fgets(buffer, sizeof(buffer), fp)) {
        normalize_buffer(buffer);

        char *token = strtok(buffer, " ");
        while (token) {
            if (strlen(token) > 0)
```

```
                add_word(token);
            token = strtok(NULL, " ");
        }
    }
```

Ogni Mapper riceve dal Master le informazioni necessarie per identificare la porzione di file da elaborare. Il Mapper utilizza il proprio identificativo per determinare quale FIFO utilizzare e legge da essa i dati relativi al chunk, ossia l'offset iniziale e la dimensione del segmento assegnato.

Grazie a queste informazioni, il Mapper accede esclusivamente al segmento di file designato, evitando sovrapposizioni con altri Mapper e garantendo che ogni processo lavori in modo indipendente sul proprio chunk. Durante la lettura, eventuali parole parziali all'inizio o alla fine del chunk vengono gestite in modo da non corrompere il conteggio complessivo delle parole.

Invio al Reducer

```
    snprintf(fifo_path, sizeof(fifo_path), MAPPER_TO_REDUCER_FIFO,
    ↪  mapper_id);
    fifo_fd = open(fifo_path, O_WRONLY);
    if (fifo_fd < 0) {
        perror("open mapper_to_reducer FIFO");
        exit(EXIT_FAILURE);
    }

    FILE *stream = fdopen(fifo_fd, "w");
    if (!stream) {
        perror("fdopen");
        close(fifo_fd);
        exit(EXIT_FAILURE);
    }

    WordCount *curr = head;
    while (curr) {
        fprintf(stream, "%s %d\n", curr->word, curr->count);
        curr = curr->next;
    }

    fclose(stream);
```

Una volta completato il conteggio locale delle parole, ciascun Mapper deve trasmettere i propri risultati al Reducer. Per farlo, il Mapper costruisce il percorso della FIFO corrispondente al proprio identificativo, apre la FIFO in modalità scrittura e associa un flusso di I/O tramite `fdopen()`, permettendo di utilizzare le funzioni standard di scrittura su file.

Successivamente, il Mapper scorre la propria lista locale di parole e frequenze, scrivendo su FIFO ciascuna parola seguita dal rispettivo conteggio, riga per riga. Al termine dell'invio dei dati, il flusso viene chiuso con `fclose()`, garantendo la corretta trasmissione delle informazioni e la sincronizzazione con il Reducer, che potrà così aggregare i conteggi di tutti i Mapper in modo sicuro ed efficiente.

REDUCER.C

Il Reducer rappresenta la fase di aggregazione finale del paradigma MapReduce. Il suo compito è raccogliere i risultati parziali prodotti dai Mapper, sommare i conteggi relativi a ciascuna parola e produrre l'output finale del WordCount.

Struttura dati globali e mutex

```c
typedef struct WordCount {
    char word[MAX_WORD_LEN];
    int count;
    struct WordCount *next;
} WordCount;

WordCount *global_head = NULL;
pthread_mutex_t list_mutex = PTHREAD_MUTEX_INITIALIZER;
int total_words = 0;

typedef struct {
    int mapper_id;
} ThreadArgs;
```

Per gestire l'aggregazione concorrente dei risultati provenienti dai diversi Mapper, il Reducer utilizza una struttura dati basata su lista concatenata. La struttura `WordCount` rappresenta ciascun nodo della lista e contiene la parola analizzata, il relativo conteggio delle occorrenze e un puntatore al nodo successivo, permettendo una gestione dinamica dell'insieme di parole individuate.

Il puntatore globale `global_head` identifica l'inizio della lista condivisa, mentre il mutex `list_mutex` garantisce l'accesso mutuamente esclusivo alla struttura durante gli aggiornamenti concorrenti effettuati dai thread del Reducer. La variabile `total_words` mantiene inoltre il conteggio complessivo delle parole elaborate.

Infine, la struttura `ThreadArgs` consente di passare a ciascun thread informazioni specifiche, come l'identificativo del Mapper associato, facilitando la gestione parallela della ricezione e dell'elaborazione dei dati.

Ricerca e aggiornamento delle parole

```c
void add_word_safe(const char *word, int count) {
    pthread_mutex_lock(&list_mutex);

    WordCount *wc = find_word_unsafe(word);
    if (wc) {
        wc->count += count;
    } else {
        wc = malloc(sizeof(WordCount));
        strncpy(wc->word, word, MAX_WORD_LEN);
        wc->word[MAX_WORD_LEN - 1] = '\0';
        wc->count = count;
        wc->next = global_head;
        global_head = wc;
    }
```

```
    total_words += count;  // somma tutte le occorrenze
    pthread_mutex_unlock(&list_mutex);
}
```

La funzione `add_word_safe` implementa il meccanismo di aggiornamento concorrente della struttura dati condivisa utilizzata dal Reducer per l'aggregazione delle parole. All'inizio della funzione viene acquisito il mutex `list_mutex`, così da garantire l'accesso mutuamente esclusivo alla lista concatenata globale ed evitare condizioni di race tra i thread.

Successivamente viene ricercata la parola tramite una funzione di ricerca. Se la parola è già presente, il relativo contatore viene incrementato del valore ricevuto; in caso contrario, viene allocato dinamicamente un nuovo nodo `WordCount`, inizializzato con la parola e il conteggio corrente, e inserito in testa alla lista globale.

La variabile `total_words` viene quindi aggiornata sommando tutte le occorrenze elaborate, mantenendo una visione complessiva del numero totale di parole processate dal Reducer. Infine, il mutex viene rilasciato, consentendo ad altri thread di accedere in sicurezza alla struttura condivisa.

Thread per processare i dati da un Mapper

```
void *process_mapper(void *arg) {
    ThreadArgs *args = (ThreadArgs *)arg;
    int mapper_id = args->mapper_id;
    char fifo_path[256];

    snprintf(fifo_path, sizeof(fifo_path), MAPPER_TO_REDUCER_FIFO,
    ↪  mapper_id);

    int fifo_fd = open(fifo_path, O_RDONLY);
    if (fifo_fd < 0) {
        perror("open mapper_to_reducer FIFO");
        free(args);
        pthread_exit(NULL);
    }

    FILE *stream = fdopen(fifo_fd, "r");
    if (!stream) {
        perror("fdopen");
        close(fifo_fd);
        free(args);
        pthread_exit(NULL);
    }

    char word[MAX_WORD_LEN];
    int count;
    while (fscanf(stream, "%s %d", word, &count) == 2) {
        add_word_safe(word, count);
    }

    fclose(stream);
```

```
    free(args);
    pthread_exit(NULL);
}
```

La funzione `process_mapper` si occupa di gestire i dati provenienti da uno specifico Mapper.

Successivamente apre la FIFO in modalità lettura e la converte in uno stream standard tramite `fdopen`, così da poter utilizzare operazioni di input.

Il thread legge quindi in modo iterativo le coppie *parola–conteggio* inviate dal Mapper e, per ciascuna di esse, invoca la funzione `add_word_safe`, che aggiorna la struttura dati globale del Reducer.

Al termine della lettura, lo stream viene chiuso, la memoria degli argomenti del thread viene liberata e il thread termina la propria esecuzione in modo pulito tramite `pthread_exit`.

Questo meccanismo consente al Reducer di gestire simultaneamente più flussi di dati provenienti dai diversi Mapper, sfruttando il parallelismo a livello di thread per aggregare in modo efficiente i conteggi parziali delle parole.

Funzione main

```c
int main() {
    pthread_t threads[NUM_MAPPERS];

    /* Crea thread per ogni mapper */
    for (int i = 0; i < NUM_MAPPERS; i++) {
        ThreadArgs *args = malloc(sizeof(ThreadArgs));
        args->mapper_id = i;
        if (pthread_create(&threads[i], NULL, process_mapper, args) != 0)
        ↪ {
            perror("pthread_create");
            free(args);
            exit(EXIT_FAILURE);
        }
    }

    /* Attende tutti i thread */
    for (int i = 0; i < NUM_MAPPERS; i++) pthread_join(threads[i], NULL);

    /* Stampa risultati finali */
    print_sorted_results();

    /* Libera memoria */
    WordCount *curr = global_head;
    while (curr) {
        WordCount *next = curr->next;
        free(curr);
        curr = next;
    }

    pthread_mutex_destroy(&list_mutex);
    return 0;
}
```

La funzione `main` del Reducer rappresenta il punto di coordinamento dell'intera fase di aggregazione dei risultati.

Inizialmente viene creato un insieme di thread pari al numero di Mapper (`NUM_MAPPERS`), assegnando a ciascun thread l'identificativo del Mapper da cui dovrà ricevere i dati.

Successivamente, il Reducer attende la terminazione di tutti i thread mediante `pthread_join`, garantendo che l'aggregazione dei conteggi parziali sia completamente conclusa prima di procedere. Una volta completata la fase di raccolta, vengono stampati i risultati finali ordinati attraverso la funzione `print_sorted_results`.

Infine, la funzione provvede alla deallocazione della memoria dinamica utilizzata per la lista globale delle parole e alla distruzione del mutex di sincronizzazione, assicurando una chiusura pulita delle risorse prima della terminazione del processo.

Capitolo 4 - Risultati

Data l'implementazione aggiornata del sistema MapReduce descritta nei capitoli precedenti, in questa sezione vengono presentati i risultati ottenuti durante la fase di testing e sperimentazione del progetto. L'obiettivo principale delle prove è stato verificare il corretto funzionamento dell'algoritmo di WordCount in un contesto di esecuzione concorrente basato su comunicazione interprocesso tramite FIFO e aggregazione multithread nel Reducer, valutando inoltre il corretto comportamento del sistema..

I test sono stati eseguiti utilizzando come input il file `lotr.txt`, contenente il testo del romanzo "Il Signore degli Anelli", scelto per la sua dimensione significativa e per la presenza di un vocabolario ampio e variegato, caratteristiche ideali per valutare l'efficacia dell'algoritmo di conteggio delle parole e la correttezza della fase di aggregazione concorrente.

Ambiente di test

Le sperimentazioni sono state effettuate in ambiente Linux tramite WSL, eseguendo il sistema (Master, Mapper e Reducer) sulla stessa macchina. La comunicazione tra i processi avviene attraverso FIFO (named pipes) create nel file system locale, mentre all'interno del Reducer l'aggregazione dei risultati provenienti dai diversi Mapper è gestita tramite thread sincronizzati mediante mutex, simulando un modello di elaborazione parallela pur rimanendo su un singolo nodo fisico.

Comandi di esecuzione e testing

Una volta posizionati nella directory principale del progetto, i test vengono eseguiti secondo la seguente procedura.

```
make
```

Il comando compila tutti i moduli del sistema (master, mapper e reducer) producendo i relativi eseguibili.

Successivamente si procede cosi all'avvio del sistema MapReduce

```
./master
```

L'esecuzione del master avvia automaticamente il processo reducer e i processi mapper, in numero definito dalla costante `NUM_MAPPERS`

Il master si occupa inoltre di suddividere il file di input in chunk e di inviare a ciascun mapper l'offset e la dimensione della porzione di file da elaborare.

Al termine dell'esecuzione, il reducer stampa a schermo il risultato finale del WordCount nel formato:

```
[Master] File size: 3262595 bytes, chunk size: 815648

Avvio reducer...
```

```
Avvio mapper 0...
Avvio mapper 1...
Avvio mapper 2...
Avvio mapper 3...


[Master] Inviato chunk 0: offset= 0, size= 815648
[Master] Inviato chunk 1: offset= 815648, size= 815648
[Master] Inviato chunk 2: offset= 1631296, size= 815648
[Master] Inviato chunk 3: offset= 2446944, size= 815651

Numero parole contate in totale: 582115 parole
Numero parole uniche contate: 14085 parole

Report counter in report.txt

[Master] Esecuzione completata
```

```
...
struggling -> 10
earthward -> 1
waxed -> 4
burn -> 22
i -> 6997
signs -> 41
aright -> 3
as = 4149
it = 7855
was = 7061
eventually = 11
under = 597
bilbo = 923
to = 12084
gandalf = 1308
[Master] Esecuzione completata
```

In tutte le configurazioni testate, il sistema ha prodotto un conteggio corretto delle occorrenze delle parole presenti nel file lotr.txt.

Le parole risultano normalizzate in minuscolo e prive di caratteri non alfanumerici. Inoltre, risultano correttamente aggregate dal reducer.

Questo conferma il corretto funzionamento del sistema MapReduce implementato su un algortimo di WordCount.

Capitolo 5 - Conclusione e sviluppi futuri

Considerazioni finali e sviluppi futuri

Il progetto ha avuto come obiettivo la progettazione e l'implementazione di un sistema MapReduce semplificato in linguaggio C, capace di eseguire l'algoritmo di WordCount su un dataset testuale di grandi dimensioni rappresentato dal testo "Il Signore degli Anelli".

Dal punto di vista didattico e progettuale, il lavoro ha consentito di approfondire concetti fondamentali della programmazione concorrente e dei sistemi distribuiti.

Per quanto riguarda gli sviluppi futuri, il progetto potrebbe essere esteso in diverse direzioni. Una possibile evoluzione consiste nell'introduzione di un meccanismo di bilanciamento dinamico del carico, capace di adattare automaticamente la dimensione dei chunk assegnati ai Mapper in base alle prestazioni di esecuzione. Un ulteriore miglioramento potrebbe prevedere l'esecuzione del sistema su macchine fisicamente distribuite, superando il vincolo della singola macchina e avvicinandosi maggiormente a uno scenario di rete reale. Inoltre, l'adozione di strutture dati più efficienti per la gestione del WordCount e l'ottimizzazione della fase di Reduce potrebbero ridurre ulteriormente l'overhead computazionale. Infine, l'integrazione di meccanismi di fault tolerance, tipici dei framework MapReduce reali, permetterebbe di rendere il sistema più robusto e più vicino a soluzioni utilizzate in ambito industriale.