Ana Clara Mueller Miranda - RA 148806
Bruno Pires Moreira Silva - RA 139892
Igor Ribeiro Ferreira de Matos - RA 140492
Rafaela Cristine dos Santos Uchôas - RA 140351

Relatório Projeto 2 - MapReduce

1. Sobre o projeto

Para o segundo projeto foi proposto o desenvolvimento de uma aplicação que contasse palavras em documentos baseada no MapReduce, um modelo que divide a lógica em dois conceitos: O mapeamento e a redução.

No desenvolvimento da nossa lógica consideramos como input um arquivo de texto. Inicialmente, foi feito uma lógica que processava todo o arquivo para depois fazer a inserção no hash, entretanto, isso diminuía o desempenho do projeto já que o MapReduce ajuda justamente no processamento de altas quantidades de dados.

Para resolver esse problema utilizamos a syscall “stat” para pegar informações do arquivo de input e poder tratá-lo na função “tratamento”.

- tamanho(): Função que utiliza um struct stat e um buffer para pegar a quantidade de bits do documento, utilizando uma variável global nome para função.

```
int tamanho(){
    stat(nome, &sb);
    int t = sb.st_size;
    return t;
}
```

iguais para todos os códigos

- tratamento(int inicio, int fim): Função que recebe o inicio e o fim de um bloco de bytes que serão processados, e faz o tratamento checando se esse bloco começa ou termina no meio de uma palavra, e caso ocorra a função corrige para que o bloco comece e termine sem cortar palavras ao meio.

```c
void *tratamento(int inicio, int fim){
    FILE *auxiliar=fopen(nome,"r");
    if(inicio != 1){
        fseek(auxiliar, inicio-1, SEEK_SET);
        int letra =fgetc(auxiliar);
        while(letra!=32){
            if(letra==10){
                break;
            }
            fseek(auxiliar, inicio-1, SEEK_SET);
            letra=fgetc(auxiliar);
            inicio--;
        }
    }
    if(fim!=tamanho()){
        fseek(auxiliar, fim-1, SEEK_SET);
        int letra=fgetc(auxiliar);
        while(letra!=32){
            fseek(auxiliar, fim-1, SEEK_SET);
            letra=fgetc(auxiliar);
            fim--;
        }
    }
}
```

iguais para todos os códigos

- novo(char string []): Função que recebe uma string (uma palavra) e cria um novo elemento (do tipo que vai ser inserido na lista) e retorna esse novo elemento criado.

```c
TLista *novo(char string[]){
    TLista *novo = (TLista*) malloc(sizeof(TLista));
    novo->prox = NULL;
    novo->valor=1;
    strcpy(novo->string, string);
    return novo;
}
```

iguais para todos os códigos

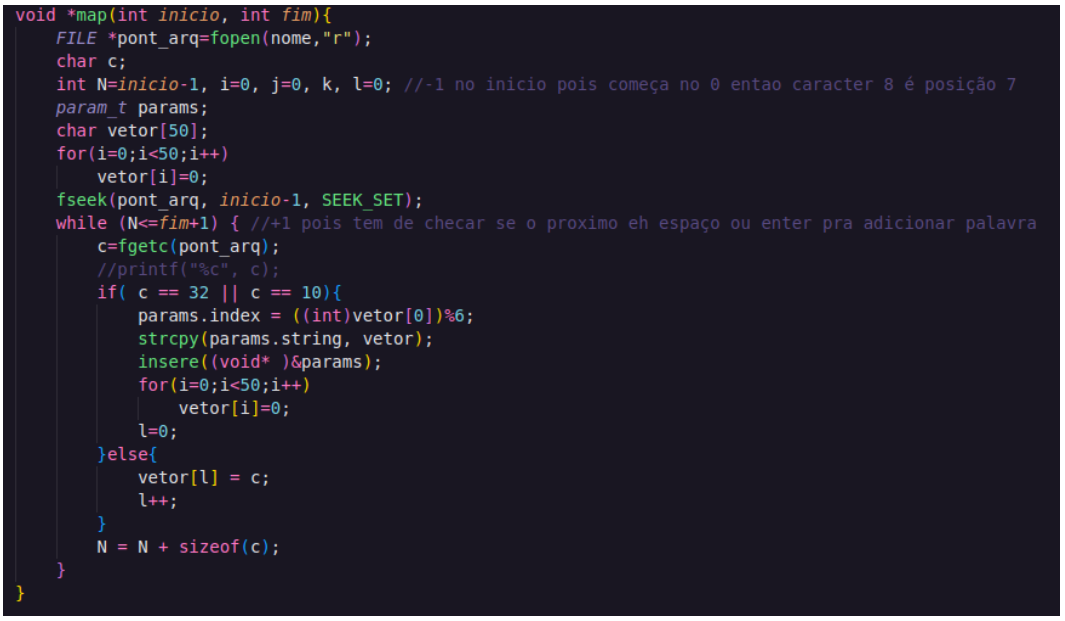
- insere(void* args): Função que recebe o elemento a ser adicionado à lista.

```c
void *insere(void* args){
    param_t castedArgs = *(param_t *)args;
    TLista *Lista = Hash[castedArgs.index];
    if (Lista==NULL){
        Hash[castedArgs.index] = novo(castedArgs.string);
        return NULL;
    }
    while(Lista->prox!=NULL)
        Lista = Lista->prox;

    Lista->prox = novo(castedArgs.string);
}
```

iguais para todos os códigos

- map(int inicio, int fim): Dado um bloco de texto, a função map percorre o arquivo e com um vetor auxiliar, que é iniciado com o caracter nulo (caracter 0). Com o primeiro caracter devidamente localizado, é iniciado um loop que itera a cada caracter do bloco de forma que os caracteres que delimitam uma string é checado, e se esse for o caso, ele é inserido na tabela hash.



```
void *map(int inicio, int fim){
    FILE *pont_arq=fopen(nome,"r");
    char c;
    int N=inicio-1, i=0, j=0, k, l=0; //-1 no inicio pois começa no 0 entao caracter 8 é posição 7
    param_t params;
    char vetor[50];
    for(i=0;i<50;i++)
        vetor[i]=0;
    fseek(pont_arq, inicio-1, SEEK_SET);
    while (N<=fim+1) { //+1 pois tem de checar se o proximo eh espaço ou enter pra adicionar palavra
        c=fgetc(pont_arq);
        //printf("%c", c);
        if( c == 32 || c == 10){
            params.index = ((int)vetor[0])%6;
            strcpy(params.string, vetor);
            insere((void* )&params);
            for(i=0;i<50;i++)
                vetor[i]=0;
            l=0;
        }else{
            vetor[l] = c;
            l++;
        }
        N = N + sizeof(c);
    }
}
```

Código n°1

```
void *map(void* args){
    //int inicio, int fim
    //void* args
    pthread_mutex_lock(&mutex_map);
    bloco bytes = *(bloco *)args;
    int inicio = bytes.inicio;
    int fim = bytes.fim;
    FILE *pont_arq=fopen(nome,"r");
    char c;
    int N=inicio-1, i=0, j=0, k, l=0; //-1 no inicio pois come?a no 0 entao caracter 8 ? posi??o 7
    param_t params[MAX_T];
    char vetor[50];
    for(i=0;i<50;i++)
        vetor[i]=0;
    fseek(pont_arq, inicio-1, SEEK_SET);
    while (N<=fim+1) { //+1 pois tem de checar se o proximo eh espa?o ou enter pra adicionar palavra
        c=fgetc(pont_arq);
        //printf("%c", c);

        if( c == 32 || c == 10 || c==-1){
            params[j].index = ((int)vetor[0])%6;
            strcpy(params[j].string, vetor);
            if(strlen(params[j].string)!=0){
                insere((void* )&params[j]);
            }
            j++;
            for(i=0;i<50;i++)
                vetor[i]=0;
            l=0;
        }else{
            vetor[l] = c;
            l++;
        }
        N = N + sizeof(c);
    }
    pthread_mutex_unlock(&mutex_map);
}
```

Código n°3

- divide(): Função que é utilizada para pegar a entrada do usuário de quantas maps ele deseja que abram ao mesmo tempo para tratar o arquivo, e dada essa quantidade calcular os intervalos, chama as threads cada thread para 1 intervalo que irá primeiro tratar o intervalo e depois mapear.

```
void divide(){
    int t = tamanho();
    int qnt_blocos= t/maps, i, j=0;
    bloco blocos[qnt_blocos];
    printf("%d / %d = %d", t, maps, qnt_blocos);
    for(i=1;i<=t;i=i+qnt_blocos){
        blocos[j].inicio=i; //inicio do bloco
        blocos[j].fim=i+qnt_blocos-1; //fim do bloco
        //printf("\nbloco: %d %d %d", j, blocos[j].inicio,  blocos[j].fim);
        //pthread_create(&mapear_tid[j],NULL, tratamento, (void*)&blocos[j]);
        //pthread_create(&mapear_tid[j],NULL, map, (void*)&blocos[j]);
        j++;
    }
    if(blocos[j-1].fim>t){
        blocos[j-1].fim=t;
    }
}
```

Código n° 1

```c
void divide(){
    int t = tamanho();
    int qnt_blocos= t/maps, i, j=0;
    bloco blocos[qnt_blocos];
    printf("%d / %d = %d\n", t, maps, qnt_blocos);
    for(i=1;i<=t;i=i+qnt_blocos){
        blocos[j].inicio=i; //inicio do bloco
        blocos[j].fim=i+qnt_blocos-1; //fim do bloco
        //printf("\nbloco: %d %d %d", j, blocos[j].inicio,  blocos[j].fim);
        pthread_create(&tratar_tid[j],NULL, tratamento, (void*)&blocos[j]);
        j++;
    }

    for(i=1;i<=t;i=i+qnt_blocos)
        pthread_join(&tratar_tid[i],NULL);


    for(i=1;i<=t;i=i+qnt_blocos)
        pthread_create(&mapear_tid[j],NULL, map, (void*)&blocos[j]);
    for(i=1;i<=t;i=i+qnt_blocos)
        pthread_join(&mapear_tid[i],NULL);


    if(blocos[j-1].fim>t){
        blocos[j-1].fim=t;
    }
}
```

Código n°3

- printList(): Função que recebe uma lista e imprime ela.

```c
void printList(TLista *Lista){
    while(Lista!=NULL){
        printf("%s ",Lista->string);
        Lista=Lista->prox;
    }
    printf("\n");
}
```

iguais para todos os códigos

- getHash(): Função que que recebe um hash e imprime o hash.

```c
void getHash(){
    int i=0;
    for(i=0;i<6;i++){
        printf("%d ",i);
        printList(Hash[i]);
    }
}
```

iguais para todos os códigos

- printar_resultado(): Função que imprime o resultado efetuado no reduce, ou seja a palavra e a quantidade de vezes em que ela ocorre.

```c
void printar_resultado(TLista *Lista){
    while(Lista!=NULL){
        if(strcmp(Lista->string, "CHECADO")!=0){
            printf("%s--------%d\n",Lista->string, Lista->valor);
        }
        Lista=Lista->prox;
    }
}
```

iguais para todos os códigos

- reduce(): Função que recebe um index, que é qual posição do Hash ele vai trabalhar. Dada essa posição (que é uma lista) ele passa por toda a lista checando palavra por palavra. Foram definidos dois ponteiros, o “Aux” percorre a lista e compara para que sejam contadas quantas palavras a lista tem e o ponteiro “Principal” indica qual palavra vai ser comparada. Caso as palavras sejam iguais, ou seja, a comparação é 0, o valor daquela chave é incrementado, e anota as palavras que já foram checadas.

```c
void *reduce(void* args){
    int index = *(int *)args;
    TLista *aux=Hash[index];
    TLista *principal=Hash[index];
    while(principal!=NULL){
        while(aux!=NULL){
            aux=aux->prox;
            if(aux==NULL){
                break;
            }
            if(strcmp(principal->string,  aux->string)==0){
                principal->valor++;
                strcpy(aux->string, "CHECADO");
            }
        }
        principal=principal->prox;
        aux=principal;
    }
}
```

iguais para todos os códigos

- mep_reduce(): Nosso Hash sempre possui 6 posições, ou seja 6 listas, essa função criar 6 threads para trabalhar no reduce de cada lista e depois printar o resultado obtido.

```c
void map_reduce(){
    int i;
    for(i=0;i<6;i++){
        pthread_create(&reduce_tid[i],NULL, reduce, (void*)&nums[i]);
    }
    for (i=0; i<6; i++){
        pthread_join(reduce_tid[i], NULL);
    }
    for(i=0;i<6;i++){
        printar_resultado(Hash[i]);
    }
}
```

iguais para todos os códigos

- struct TLista : Estrutura utilizada para cada uma das seis listas da tabela hash, com a string do item, o valor que será somado na fase reduce, que é 1, e um ponteiro de encadeamento, como o padrão de listas.s
- struct param_t : Estrutura utilizada para a passagem de argumentos para as funções chamadas em threads, mais especificamente, na função de inserção das listas.

```c
typedef struct TLista{
    char string[MAX_CHAR];
    int valor;
    struct TLista *prox;
}TLista;
typedef struct {
    int index;
    char string[MAX_CHAR];
}param_t;
```

iguais para todos os códigos

Um problema que foi encontrado no desenvolvimento foi em questão a sincronização para inserir no Hash utilizando a quantidade de threads que foi escolhida pelo usuário. No código da lógica aplicada inicialmente, mostrado abaixo, a inserção funcionava normalmente, pois não era utilizada as ideias de múltiplas threads trabalhando ao mesmo tempo no map.

```
void *map(int inicio, int fim){
    FILE *pont_arq=fopen(nome,"r");
    char c;
    int N=inicio-1, i=0, j=0, k, l=0; //-1 no inicio pois começa no 0 entao caracter 8 é posição 7
    param_t params;
    char vetor[50];
    for(i=0;i<50;i++)
        vetor[i]=0;
    fseek(pont_arq, inicio-1, SEEK_SET);
    while (N<=fim+1) { //+1 pois tem de checar se o proximo eh espaço ou enter pra adicionar palavra
        c=fgetc(pont_arq);
        //printf("%c", c);
        if( c == 32 || c == 10){
            params.index = ((int)vetor[0])%6;
            strcpy(params.string, vetor);
            insere((void* )&params);
            for(i=0;i<50;i++)
                vetor[i]=0;
            l=0;
        }else{
            vetor[l] = c;
            l++;
        }
        N = N + sizeof(c);
    }
}
```

Entretanto, quando alteramos para nova lógica que lê os blocos definidos pela função de tratamento, começamos a ter problema com essa sincronização.

Para fazer a nova função divide() dividimos o processo em duas partes, primeiro abrimos as threads que fazem todo o tratamento e depois que isso é finalizado,

```
void divide(){
    int t = tamanho();
    int qnt_blocos= t/maps, i, j=0;
    bloco blocos[qnt_blocos];
    printf("%d / %d = %d\n", t, maps, qnt_blocos);
    for(i=1;i<=t;i=i+qnt_blocos){
        blocos[j].inicio=i; //inicio do bloco
        blocos[j].fim=i+qnt_blocos-1; //fim do bloco
        //printf("\nbloco: %d %d %d", j, blocos[j].inicio,  blocos[j].fim);
        pthread_create(&tratar_tid[j],NULL, tratamento, (void*)&blocos[j]);
        j++;
    }

    for(i=1;i<=t;i=i+qnt_blocos)
        pthread_join(&tratar_tid[i],NULL);

    for(i=1;i<=t;i=i+qnt_blocos)
        pthread_create(&mapear_tid[j],NULL, map, (void*)&blocos[j]);
    for(i=1;i<=t;i=i+qnt_blocos)
        pthread_join(&mapear_tid[i],NULL);

    if(blocos[j-1].fim>t){
        blocos[j-1].fim=t;
    }
}
```

```c
void *map(void* args){
    //int inicio, int fim
    //void* args
    pthread_mutex_lock(&mutex_map);
    bloco bytes = *(bloco *)args;
    int inicio = bytes.inicio;
    int fim = bytes.fim;
    FILE *pont_arq=fopen(nome,"r");
    char c;
    int N=inicio-1, i=0, j=0, k, l=0; //-1 no inicio pois come[illegible]a no 0 entao caracter 8 [illegible] posi[illegible]o 7
    param_t params[MAX_T];
    char vetor[50];
    for(i=0;i<50;i++)
        vetor[i]=0;
    fseek(pont_arq, inicio-1, SEEK_SET);
    while (N<=fim+1) { //+1 pois tem de checar se o proximo eh espa[illegible]o ou enter pra adicionar palavra
        c=fgetc(pont_arq);
        //printf("%c", c);

        if( c == 32 || c == 10 || c==-1){
            params[j].index = ((int)vetor[0])%6;
            strcpy(params[j].string, vetor);
            if(strlen(params[j].string)!=0){
                insere((void* )&params[j]);
            }
            j++;
            for(i=0;i<50;i++)
                vetor[i]=0;
            l=0;
        }else{
            vetor[l] = c;
            l++;
        }
        N = N + sizeof(c);
    }
    pthread_mutex_unlock(&mutex_map);
}
```

Para a função de Map() temos a diferença da checagem de diferentes caracteres que geram problemas com threads, com base na tabela ASCII. Os problemas que ocorreram geraram códigos que funcionam parcialmente, porém com detalhes que não conseguimos resolver para rodar 100%. Temos 1 código que funciona porém não abre múltiplas threads para tratar o map (arquivo texto.c), outro que usa threads no map porém tem problemas de sincronização (arquivo texto3.c) e outro que funciona porém corta palavras as vezes e em outras não corta palavras (arquivo texto2.c). Anexamos os 3 arquivos pois julgamos importante para ver que tentamos de vários jeitos chegar no resultado final desejado.

2. Organização do grupo

Após discutirmos em sala nas aulas de laboratório como o projeto seria desenvolvido acabamos dividindo as funções da seguinte forma, baseado no que cada um tinha mais facilidade:

Leitura de Arquivo/Tratamento/Main: Ana Clara e Rafaela

Map: Bruno

Reduce: Ana Clara e Igor

Relatório: Todos

3. Auto-Avaliação do grupo

Inicialmente houveram alguns problemas de entendimento da lógica, em que ao juntar as funções que foram desenvolvidas percebemos que os integrantes haviam entendido a lógica discutida de forma errada. Com o auxílio do professor foi possível que a lógica fosse alinhada para que o desenvolvimento do projeto não fosse prejudicado. Todos os integrantes foram proativos de forma que a carga do trabalho não ficasse muito pesada para ninguém.