Projeto 2 – Ordenação de vetor e Multithreads

Sistemas Operacionais TT304 A

Grupo Trio da Depressão

Alice Mantovani 193539

Laura Margaritelli 200978

Viviane Moraes 207152

Limeira – SP

Junho/2018

Objetivo e especificações:

O trabalho deveria utilizar multithreads para a ordenação de um vetor de mais de 99.999 valores. Esses valores deveriam ser lidos de um arquivo de entrada e, após a ordenação, um arquivo de saída deveria ser criado contendo o vetor ordenado. Para a ordenação do vetor, um sort rápido deveria ser utilizado.

A quantidade de elementos a serem ordenados, a quantidade de threads que seriam executadas, e os nomes do arquivo de entrada e arquivo de saída deveriam ser especificados na entrada do programa pelo próprio usuário.

O projeto visa a análise do desempenho do programa criado pelo grupo na execução de 2, 4, 8 e 16 threads.

Solução do problema:

Para solucionar o problema, resolvemos alocar dentro da própria thread, um determinado trecho do vetor principal. Os trechos seriam divididos em tamanhos iguais, ou então os primeiros trechos iriam receber uma posição a mais do que os trechos finais do vetor.

Para a ordenação desses trechos, nosso grupo decidiu usar um heapsort, e um merge para a ordenação entre os trechos já prontos.

Código:

Bibliotecas utilizadas, função para limpar o buffer e declarações globais de variáveis:

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#define TAM 20

void flush_in (){
        int ch;
        while((ch = fgetc(stdin)) != EOF && ch != '\n'){}
}

typedef struct argumentos{
        int tam;
        int posi;
}thread;
struct argumentos *argthread;

int *varquivo;
```

A biblioteca *pthread* (POSIX thread) é necessária para a utilização de multithreads.

Declaramos o vetor como um ponteiro *varquivo*.

A struct *argumento* contém os dados que serão passados como argumento na criação das threads.

A variável *tam* armazena o tamanho que o vetor alocado *v* que determinada thread terá. Já a variável *posi* guarda a posição inicial onde o vetor alocado *v* se encontrará em relação ao vetor principal *varquivo*. Se *posi* para determinado vetor *v* for 5, por

exemplo, então ele se encontrará a partir da sexta posição do vetor *varquivo* (considerando a primeira posição como zero)

```
20 void merge (int *varquivo, int inicio, int meio, int fim){
21         int *temp, p1, p2, tamanho, i, j, k;
22         int fim1 = 0, fim2 = 0;
23         tamanho = fim-inicio+1;
24         p1 = inicio;
25         p2 = meio+1;
26         temp = (int *) malloc(tamanho*sizeof(int));
27         if (temp != NULL){
28                 for(i = 0; i < tamanho; i++){
29                         if(!fim1 && !fim2){
30                                 if(varquivo[p1] < varquivo[p2])
31                                         temp[i] = varquivo[p1++];
32                                 else
33                                         temp[i] = varquivo[p2++];
34                                 if(p1 > meio)
35                                         fim1 = 1;
36                                 if(p2 > fim)
37                                         fim2 = 1;
38                         }
39                         else{
40                                 if(!fim1)
41                                         temp[i] = varquivo[p1++];
42                                 else
43                                         temp[i] = varquivo[p2++];
44                         }
45                 }
46                 for(j = 0, k = inicio; j < tamanho; j++, k++)
47                         varquivo[k] = temp[j];
48         }
49         free(temp);
50 }
```

A função *merge* é utilizada para ordenar trechos ordenados do vetor, em pares.

Cada thread irá ordenar um trecho, e a função *merge* irá unir esses trechos ordenados.

```c
void criaHeap (int *v, int i, int f){
        int aux = v[i];
        int j = i*2 + 1;
        while(j <= f){
                if(j < f){
                        if(v[j] < v[j+1])
                                j += 1;
                }
                if(aux < v[j]){
                        v[i] = v[j];
                        i = j;
                        j = 2*i + 1;
                }
                else{
                        j = f + 1;
                }
        }
        v[i] = aux;
}

void heapsort (int *v, int N){
        int i, aux;
        for(i = (N-1)/2; i >= 0; i--){
                criaHeap(v, i, N-1);
        }
        for(i = N-1; i >= 1; i--){
                aux = v[0];
                v[0] = v[i];
                v[i] = aux;
                criaHeap(v, 0, i-1);
        }
}
```

A função *heapsort* e *criaheap* são executadas pelas threads, que irão então ordenar trechos do vetor principal *varquivo*.

```c
void *tfunc (void *t_arg){
        struct argumentos *t;
        t = (thread *) t_arg;

        int *v, tam, ini, i, p;

        tam = t->tam;
        ini = t->posi;

        v = (int *)malloc(tam * sizeof(int));

        p = ini;
        for(i = 0; i < tam; i++){
                v[i] = varquivo[p];
                p++;
        }

        heapsort(v, tam);

        p = ini;
        for(i = 0; i < tam; i++){
                varquivo[p] = v[i];
                p++;
        }

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

**tfunc* é a função da thread, onde ocorre a alocação de um vetor menor *v* de tamanho *tam* (dado recebido pelos argumentos da thread). O primeiro *for* dessa função é onde o vetor *v* receberá os valores de determinada posição do vetor *varquivo* (que é determinado pelo dado posi recebido como argumento).

O segundo *for* da função **tfunc* acontece após a ordenação do vetor *v*, e é onde o vetor *varquivo* receberá os valores já ordenados do trecho determinado por *ini* (o argumento *posi*).

Depois de terminada, a memória é liberada (*t_arg*), e a thread é finalizada (*pthread_exit*).

```
117 int main (){
118         int N, i, p, T, fim, ini, *v;
119
120         printf("Quantidade de valores a serem ordenados : ");
121         scanf("%d", &N);
122
123         if(N <= 99999){
124                 printf("\nO arquivo deve conter mais de 99.999 valores.\n");
125                 printf("Insira novamente a quantidade: ");
126                 scanf("%d", &N);
127         }
128
129         printf("Quantidade de threads : ");
130         scanf("%d", &T);
131
132         flush_in();
133
134         printf("Nome do arquivo de entrada : ");
135         char ne[TAM];
136         gets(ne);
137
138         printf("Nome do arquivo de saída : ");
139         char ns[TAM];
140         gets(ns);
```

As entradas do programa são: quantidade de valores (*N*), quantidade de threads (*T*), nome do arquivo de entrada (*ne*) e nome do arquivo de saída (*ns*).

Como especificado na proposta do projeto, o programa deveria ordenar mais de 99.999 valores lidos de um arquivo de entrada. Para tanto, o programa possui uma condição para caso o valor digitado pelo usuário seja igual ou menor a 99.999, pedindo novamente a inserção do valor *N*.

```
142         varquivo = (int *)malloc(N * sizeof(int));
143
144         FILE *entrada = fopen(ne, "r");
145
146         if(entrada == NULL)
147                 printf("Erro ao abrir o arquivo\n");
148
149         for(p = 0; p < N; p++){
150                 fscanf(entrada, "%d", &i);
151                 varquivo[p] = i;
152         }
153
154         fclose(entrada);
```

O programa então abre o arquivo em modo de leitura (*r*) com o nome especificado pelo usuário. Caso o arquivo não exista ou tenha algum problema em sua abertura, uma mensagem de erro é mostrada.

Na leitura do arquivo, os valores lidos são armazenados no vetor *varquivo* (ponteiro declarado globalmente), que é alocado dinamicamente de acordo com o N inserido pelo usuário.

```
156         pthread_t meusthreads[T];
```

Declaração de *T* threads.

```
158         int vpi[T], vpf[T], aux, tamv, cont;
```

Variáveis utilizadas para divisão em trechos e passagem de argumentos das threads.

```
160         cont = 0;
161         for(p = 0; p < T; p++){
162                 cont++;
163                 fim = cont * (N/T) - 1;
164                 ini = fim - (N/T) + 1;
165                 vpi[p] = ini;
166                 vpf[p] = fim;
167         }
168
169         if( ((N - 1) - T * (N/T) - 1) != 0 ){
170                 cont = 0;
171                 for(p = 0; p < N%T; p++){
172                         vpf[cont] += 1;
173                         aux = cont + 1;
174                         while(aux <= T){
175                                 vpi[aux] += 1;
176                                 vpf[aux] += 1;
177                                 aux++;
178                         }
179                 cont++;
180                 }
181         }
```

Para que cada thread ordene uma determinada parte do vetor, é necessário separar o vetor *varquivo* em diferentes trechos. Para isso, utilizamos uma variável *cont*, que indica a qual thread o trecho será dedicado.

A variável *ini* guarda o valor da posição inicial para determinado trecho, assim como *fim* guarda o valor da posição final, que são passados para os vetores *vpi[ ]* e *vpf[ ]*, ambos com *T* posições.

O *if* apresenta a condição caso não seja possível dividir o vetor *varquivo* em trechos iguais. Se houver resto na divisão, as posições serão distribuídas entre os trechos do vetor principal.

```
183         for(p = 0; p < T; p++){
184                 argthread = (thread *)malloc(sizeof(thread));
185
186                 tamv = vpf[p] + 1 - vpi[p];
187
188                 argthread->tam = tamv;
189                 argthread->posi = vpi[p];
190
191                 pthread_create(&meusthreads[p], NULL, tfunc, (void *)argthread);
192         }
193
194         for(p = 0; p < T; p++){
195                 pthread_join(meusthreads[p], NULL);
196         }
```

O primeiro *for* dessa parte mostra a alocação dinâmica de uma struct do tipo *struct argumentos*, que irão guardar os valores de *tamv* e *vpi[ ]*.

O tamanho do vetor que será alocado por determinada thread, *tamv*, considera a posição inicial e final do trecho correspondente do vetor principal *varquivo*.

O segundo *for* possui o papel de semáforo para as threads. O *pthread_create* espera o final da execução das threads para que se possa continuar a execução do programa principal.

```
198         merge(varquivo, vpi[0], vpf[0], vpf[1]);
```

A função *merge* deve ser chamada para se ordenar dois trechos já ordenados adjacentes do vetor *varquivo*.

A primeira chamada da função *merge* será feita independentemente de quantas threads foram criadas (quantidade de trechos ordenados do vetor principal *varquivo*).

Para 2 trechos ordenados, realiza-se apenas a primeira chamada de função.

O *merge* possui como parâmetros, o vetor a ser ordenado, o início do trecho, final do trecho, e o meio.

Para o parâmetro de início, passamos o *vpi[ ]* do primeiro trecho ordenado, e para o parâmetro de final, passamos o *vpf[ ]* do segundo trecho ordenado. No parâmetro meio, o mergesort do qual nos baseamos utilizava o cálculo:

meio = floor(início + final)/2

floor faz o arredondamento para baixo, e por isso passamos como meio, o valor da posição final *vpf* do primeiro trecho ordenado.

```
200         if(T == 4){
201                 merge(varquivo, vpi[2], vpf[2], vpf[3]);
202                 merge(varquivo, vpi[0], vpf[1], vpf[3]);
203         }
204
205         if(T == 8){
206                 merge(varquivo, vpi[2], vpf[2], vpf[3]);
207                 merge(varquivo, vpi[4], vpf[4], vpf[5]);
208                 merge(varquivo, vpi[6], vpf[6], vpf[7]);
209                 merge(varquivo, vpi[0], vpf[1], vpf[3]);
210                 merge(varquivo, vpi[4], vpf[5], vpf[7]);
211                 merge(varquivo, vpi[0], vpf[3], vpf[7]);
212         }
213
214         if(T == 16){
215                 merge(varquivo, vpi[2], vpf[2], vpf[3]);
216                 merge(varquivo, vpi[4], vpf[4], vpf[5]);
217                 merge(varquivo, vpi[6], vpf[6], vpf[7]);
218                 merge(varquivo, vpi[8], vpf[8], vpf[9]);
219                 merge(varquivo, vpi[10], vpf[10], vpf[11]);
220                 merge(varquivo, vpi[12], vpf[12], vpf[13]);
221                 merge(varquivo, vpi[14], vpf[14], vpf[15]);
222                 merge(varquivo, vpi[0], vpf[1], vpf[3]);
223                 merge(varquivo, vpi[4], vpf[5], vpf[7]);
224                 merge(varquivo, vpi[8], vpf[9], vpf[11]);
225                 merge(varquivo, vpi[12], vpf[13], vpf[15]);
226                 merge(varquivo, vpi[0], vpf[3], vpf[7]);
227                 merge(varquivo, vpi[8], vpf[11], vpf[15]);
228                 merge(varquivo, vpi[0], vpf[7], vpf[15]);
229         }
```

A chamada de *merge* foi então feita em duplas de trechos ordenados, de acordo com a quantidade de threads que foram utilizadas para a ordenação do vetor *varquivo*.

Todos as quantidades de threads especificadas na proposta do projeto para a ordenação do vetor, são valores que podem ser representados em potência de base 2, e por isso é possível ordenar em pares, e respectivamente ordenar os trechos já unidos por *merge* até que todo o vetor seja ordenado.

```
231         FILE *saida = fopen(ns, "w");
232
233         for(p = 0; p < N; p++){
234                 fprintf(saida, "%d ", varquivo[p]);
235         }
236
237         fclose(saida);
238
239 return 0;
240 }
```

Ao final do código, para o arquivo de saída, abre-se em modo escrita (*w*). Em seguida, gravam-se os valores do vetor principal *varquivo* nesse arquivo com nome especificado pelo usuário na entrada do programa.

Compilação:

```
~/Documentos$ gcc codfinal.c -o cf -lpthread
```

Para compilar, é preciso utilizar -lpthread devido a biblioteca POSIX thread.

Gráfico:

Tempos de execução obtidos na ordenação de 500.000 valores:

```
Quantidade de valores a serem ordenados : 5000000
Quantidade de threads : 2
Nome do arquivo de entrada : entrada
Nome do arquivo de saída : saida

real    0m20.646s
user    0m8.935s
sys     0m0.173s
```

```
Quantidade de valores a serem ordenados : 5000000
Quantidade de threads : 4
Nome do arquivo de entrada : entrada
Nome do arquivo de saída : saida

real    0m16.273s
user    0m8.337s
sys     0m0.159s
```

```
Quantidade de valores a serem ordenados : 5000000
Quantidade de threads : 8
Nome do arquivo de entrada : entrada
Nome do arquivo de saída : saida

real    0m16.872s
user    0m7.915s
sys     0m0.179s
```

```
Quantidade de valores a serem ordenados : 5000000
Quantidade de threads : 16
Nome do arquivo de entrada : entrada
Nome do arquivo de saída : saida

real    0m16.542s
user    0m7.625s
sys     0m0.198s
```

Tempos de execução obtidos na ordenação de 50.000.000 valores:

```
Quantidade de valores a serem ordenados : 50000000
Quantidade de threads : 2
Nome do arquivo de entrada : entrada
Nome do arquivo de saída : saida

real    2m11.689s
user    2m0.013s
sys     0m2.598s
```

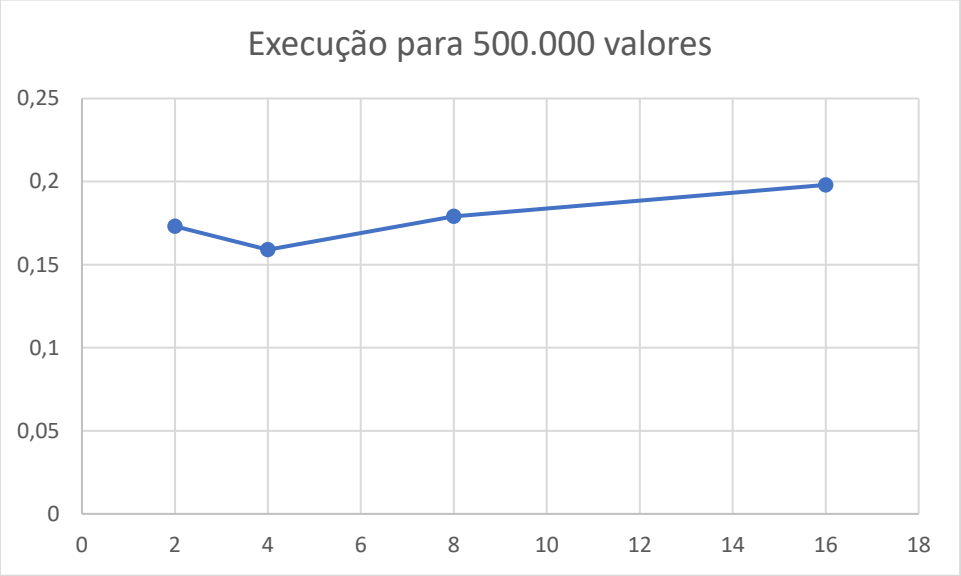
```
Quantidade de valores a serem ordenados : 50000000
Quantidade de threads : 4
Nome do arquivo de entrada : entrada
Nome do arquivo de saída : saida

real    2m3.428s
user    1m51.079s
sys     0m2.298s
```

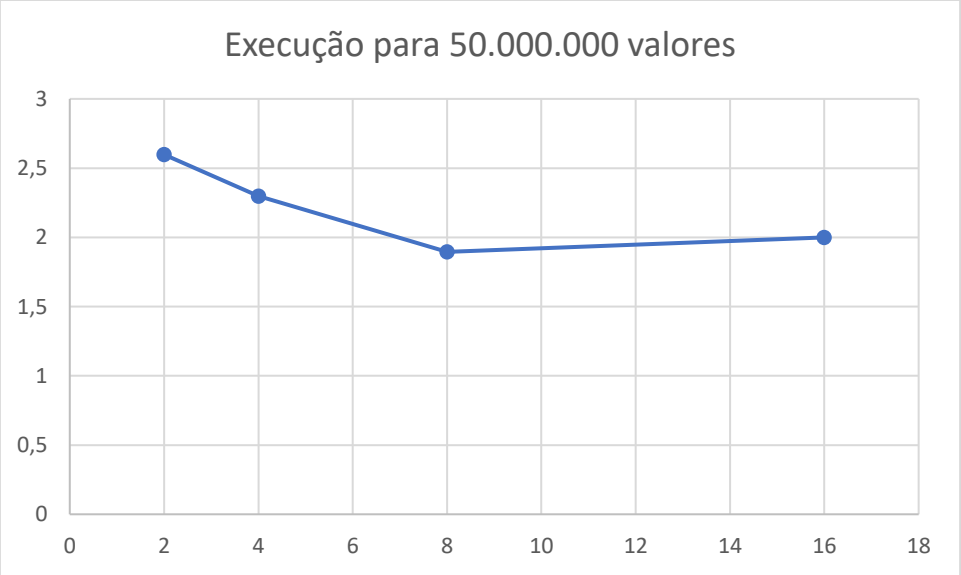
```
Quantidade de valores a serem ordenados : 50000000
Quantidade de threads : 8
Nome do arquivo de entrada : entrada
Nome do arquivo de saída : saida

real    1m57.067s
user    1m43.529s
sys     0m1.895s
```

```
Quantidade de valores a serem ordenados : 50000000
Quantidade de threads : 16
Nome do arquivo de entrada : entrada
Nome do arquivo de saída : saida

real    1m49.371s
user    1m37.388s
sys     0m1.999s
```

Os tempos utilizados nos gráficos são os tempos do sistema (*sys*).

Tempo de execução (em segundos) x Quantidade de threads:





Conclusão:

Ao obter os tempos de execução para 500.000 e 50.000.000 de valores, com 2, 4, 8 e 16 threads, é notável que, com poucos dados a serem ordenados, uma menor quantidade de threads é mais eficiente em tempo se comparado com uma maior quantidade. Quanto mais valores a serem ordenados, melhor é o desempenho para mais threads sendo executadas.

Multithreads se tornam mais vantajosas com um programa em maior escala.

Em relação ao nosso código, por termos utilizado, ao final, chamadas normais de função (sem a utilização de multithreads), consideramos que também seja um fator que influencie nos tempos de execução, além da alocação de vetores menores para cada uma das threads. A própria criação de threads é algo que exige tempo de execução e, por isso, utilizar mais threads as vezes requer mais tempo.

Link para o repositório do Github (código comentado e relatório):

https://github.com/vmb18/SistemasOperacionais

Link para o Google Drive (vídeo):

https://drive.google.com/file/d/1OKiTZrOGjyXM0p-x_BWlFfYj9PsLT8Pu/view