



**UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE TECNOLOGIA**



Sistemas Operacionais
Projeto “Ordenação de alto desempenho”

Nomes:

Irislene Silveira de Paiva	175007
Rodrigo de Lima Martins	157194

1. CÓDIGO FONTE

A seguir está o Código Fonte usado para o projeto proposto. As bibliotecas criadas *arquivo.h* e *mergesort.h* estão como anexo ao final deste relatório. Este código pode ser encontrado com o nome *projSO* no repositório do GitHub do grupo, disponível no link a seguir:

Link: <https://github.com/rodrigomlima/SnackOverflow>

Código fonte:

```
/* -----  
                                BIBLIOTECAS  
----- */  
  
#include<stdio.h>  
#include<stdlib.h>  
#include<time.h>  
#include<string.h>  
#include<pthread.h>  
// Criadas  
#include"arquivo.h" // Para manipular os arquivos  
#include"mergesort.h" // Método de ordenação Merge Sort  
/* -----  
                                ESTRUTURA PARA AS THREADS  
----- */  
  
typedef struct  
{  
    int tamanho, posicao;  
    double *parte;  
} thread_arg;  
  
/* -----  
                                PROTÓTIPO DAS FUNÇÕES  
----- */  
  
// Função para threads  
void *fthread (void *var);  
// Função auxiliar  
int def_tamanho(int tam, int *resto, int n_thread);  
  
/* -----  
                                FUNÇÃO MAIN  
----- */  
  
int main (void)  
{  
    // Delcaração de variáveis  
    int N, T, i;  
    char arq_e[100], arq_s[100];  
    int tam_parte, part_temp, resto;
```

```

thread_arg arg;

// Inserção do número de itens do vetor
printf("\nDigite a quantidade de numeros desejada: ");
scanf("%d", &N);

// Inserção do número de threads a serem usadas
printf("Digite a quantidade de threads desejada (2, 4, 8 ou 16): ");
scanf("%d", &T);

// Verifica se foi digitado o valor correto para o número de threads
while (T != 2 && T != 4 && T != 8 && T != 16)
{
    printf("Opcao digitada invalida! Digite novamente!");
    printf("\nDigite a quantidade de threads desejada (2, 4, 8 ou
16): ");
    scanf("%d", &T);
}

// Inserção dos nomes do arquivo de entrada
getchar();
printf("Digite o nome do arquivo de entrada: ");
fgets(arq_e, 100, stdin);
arruma_nome(arq_e);

printf("Digite o nome do arquivo de saida: ");
fgets(arq_s, 100, stdin); // Saída
arruma_nome(arq_s);

// Chama a função para criar o arquivo. Comente caso queira usar o
mesmo arquivo
cria_arquivo(N, arq_e);
printf("\nVetor aleatorio gravado como %s", arq_e);

// Carrega o vetor com os valores do arquivo
double vetor[N];
carrega_vetor(N, vetor, arq_e);

// Operações para dividir o vetor para as threads
tam_parte = def_tamanho(N, &resto, T);

// Cria as threads de acordo com o número especificado
pthread_t t_sort[T];

// Loop para iniciar as threads
for(i = 0; i < T; i++)
{
    // Coloca o tamanho das partes do vetor em uma variável
temporária
    part_temp = tam_parte;
    if (resto != 0) // Para distribuir o resto para cada thread

```

```

        {
            part_temp++;
            resto--;
        }
        // Passa os valores para struct
        arg.tamanho = part_temp;
        arg.posicao = i;
        arg.parte = &vetor[part_temp * i];

        // Criação das threads
        pthread_create(&t_sort[i], NULL, fthread, (void *)&arg);
    }

    // Espera as threads acabarem
    for(i = 0; i < T; i++)
        pthread_join(t_sort[i], NULL);

    // Une o resto usando mais uma vez Merge Sort
    mergeSort(vetor, 0, N - 1);

    // Salva o arquivo
    salva_arquivo(N, vetor, arq_s);
    printf("\nVetor ordenado gravado como %s\n", arq_s);

    // Encerra o programa
    pthread_exit((void *)NULL);
}

/* -----
    DECLARAÇÃO DAS FUNÇÕES
    ----- */
// Função para definir os tamanhos dos espaços de trabalho das threads
int def_tamanho(int tam, int *resto, int n_thread)
{
    int tam_parte;

    tam_parte = tam / n_thread; // Divisão
    *resto = tam % n_thread; // Resto

    return tam_parte;          // Retorna o resultado da divisão
}

// Função para thread
void *fthread (void *var)
{
    // Carrega a estrutura na thread
    thread_arg *arg = (thread_arg *) var;

    // Chama a função para o Merge Sort
    mergeSort(arg->parte, 0, arg->tamanho - 1);
}

```

```
    // Encerra a thread
    pthread_exit((void *) NULL);
}
```

2. VÍDEO

Para este relatório foi composto um vídeo mostrando os códigos utilizados e o mesmo em execução. O vídeo está disponível no link a seguir:

Link: <https://drive.google.com/file/d/18floTFKrQhrQcrkrLF5Gi1G9brPF-t1r/view?usp=sharing>

3. INSTRUÇÕES SOBRE O PROGRAMA

3.1. Descrição da solução do problema

O código seguiu o seguinte algoritmo para solucionar o problema proposto:

- a) **Declarar variáveis:** sendo elas: tamanho do vetor, número de threads, string para arquivos de entrada e saída, valores para armazenar a divisões das partes do vetor para as threads, estrutura para thread e variável auxiliar contadora;
- b) **Inserir dados:** sendo eles: número de posições do vetor, número de threads e nomes dos arquivos de entrada e saída;
- c) **Chama função para criar arquivo:** recebendo o nome do arquivo e o número de posições do vetor:
 - i) **Garantir aleatoriedade dos números criados;**
 - ii) **Declara variáveis:** sendo uma para o arquivo, uma para armazenar números reais e outra auxiliar para ser contador;
 - iii) **Abre o arquivo para escrita:** de acordo com o nome recebido pela função;
 - iv) **Criar valores aleatórios:** e armazena na variável para números reais;
 - v) **Salva sequencialmente no arquivo:** até o valor delimitado pelo usuário;
 - vi) **Fecha o arquivo criado.**
- d) **Declara variável vetor:** de acordo com o número de posições inserido pelo usuário;
- e) **Carrega o vetor com os valores do arquivo:** usa-se uma função para ler o arquivo e armazenar no vetor, que recebe o tamanho do vetor, o vetor e o nome do arquivo especificado:
 - i) **Declara variáveis:** sendo uma para o arquivo e outra auxiliar para ser contador;

- ii) **Abre o arquivo como somente leitura:** de acordo com o nome recebido pela função;
 - iii) **Lê cada item do arquivo e armazena no vetor:** até o valor delimitado pelo usuário;
 - iv) **Fecha o arquivo.**
- f) **Chama função para resolver os espaços do vetor:** recebendo o tamanho do vetor, número de threads e uma variável auxiliar para o resto;
- i) **Declara variável:** para receber o tamanho das partes divididas;
 - ii) **Tamanho das partes recebe a divisão do tamanho do vetor pelo número de threads;**
 - iii) **Resto recebe o resto pelo módulo do tamanho do vetor pelo número de threads;**
 - iv) **Retorna o tamanho das partes;**
- g) **Chama função para as threads:**
- i) **Carrega valores na thread:** usando uma estrutura para esta passagem e carregando a posição da thread, o espaço que ela lerá no vetor e a posição inicial do vetor;
 - ii) **Chama a função de criação das threads:** recebendo a variável da thread de acordo com o tamanho digitado e a estrutura para passagem de valores;
 - 1) **Carrega a estrutura na thread;**
 - 2) **Chama a função de mergesort:** a partir da posição da thread, o espaço que ela lerá no vetor e a posição inicial do vetor;
 - 3) **Finaliza a thread.**
- h) **Espera todas as threads acabarem;**
- i) **Chama mais uma vez a função de ordenação Merge Sort:** Para o vetor inteiro, ordenando as partes ordenadas;
- j) **Salva o vetor ordenado no arquivo:** com uma função que recebe o tamanho do vetor, o vetor e o nome do arquivo de saída especificado:
- i) **Declara variáveis:** sendo uma para o arquivo e outra auxiliar para ser contador;
 - ii) **Cria um arquivo para escrita:** de acordo com o nome recebido pela função;
 - iii) **Grava os valores do vetor no arquivo;**
 - iv) **Fecha o arquivo criado.**
- k) **Encerra o programa.**

3.2. Instruções para compilação

Para compilar o código, é recomendado utilizar um sistema Linux. As instruções aqui serão escritas tendo como base um sistema Linux e estão descritas a seguir:

- a) Primeiro, é preciso acessar o repositório do GitHub que contém os arquivos necessários. Ele está disponível no seguinte endereço:

Link: <https://github.com/rodrigomlima/SnackOverflow>

- b) Pode-se tanto clonar ou baixar os arquivos do repositório, via o site ou pela linha de comando. Com exceção do *README.md*, são necessários todos os arquivos. É importante que todos estejam na mesma pasta;
- c) Acesse a pasta via terminal. Uma vez que existe um arquivo *makefile* junto dos outros arquivos, use o comando *make* para compilá-lo. O conteúdo do *makefile* está em anexo no final do arquivo, como *Anexo X - makefile*, assim como também está disponível do repositório do grupo;
- d) Execute o comando *./projSO* para rodar o programa. Pode-se usar *time ./projSO* caso queira também ver os tempos de execução.

4. RESULTADOS OBTIDOS

O programa teve seu tempo de execução testado com o comando *time* usado antes do código para rodá-lo. Foram feitos 2 testes com tamanhos grandes de vetor (*N*) grandes, para que se pudesse ver mais claramente as diferenças de desempenho. O Teste 1 foi feito com *N* = 100.000 enquanto o Teste 2 foi feito com *N* = 500.000. Para garantir que as condições de cada teste fossem iguais, tirou-se do programa a parte que cria arquivos com vetores aleatórios, comentando as linhas 72 e 73 de *projSO.c*. Os vetores usados estão disponíveis nos links a seguir:

N = 100.000:

<https://drive.google.com/file/d/1jYO1P2v6Wp4gGocvmT7ursFt8jUt0xRQ/view?usp=sharing>

N = 500.000:

<https://drive.google.com/file/d/1rQDIzohTi2131FPAmqnX57a7Ulpqgx9n/view?usp=sharing>

Os testes foram realizados com todos os valores para o número de threads (*T*) pedidos (2, 4, 8 e 16) e também, para efeitos de comparação, sem o uso de threads. Para testar sem threads, basta comentar as linhas 83 a 102.

A Tabela 1 a seguir mostra os resultados obtidos, em milisegundos (ms):

Teste 1						Teste 2					
N	T	real	user	sys	#	N	T	real	user	sys	#
100000	2	7292	169	1	1	500000	2	6571	689	16	1
		5776	151	5	2			5244	711	4	2
		7917	152	12	3			6244	693	17	3
		5179	154	4	4			5954	706	4	4
		6541	157	6	Média			6003	700	10	Média
	4	6572	151	0	1		4	6167	708	28	1
		6063	149	0	2			6378	707	12	2
		7172	137	9	3			5118	689	20	3
		4861	147	0	4			6078	696	12	4
		6167	146	2	Média			5935	700	18	Média
	8	6187	142	4	1		8	6662	678	20	1
		5681	147	0	2			5600	680	8	2
		7157	139	4	3			4911	685	4	3
		5311	131	12	4			6715	698	8	4
		6084	140	5	Média			5972	685	10	Média
	16	7658	141	4	1		16	6377	673	8	1
		6696	138	8	2			5646	661	16	2
		8612	138	4	3			4666	662	16	3
		4396	136	8	4			6735	662	24	4
		6841	138	6	Média			5856	665	16	Média
	Sem Threads	7981	109	12	1		Sem Threads	8564	549	16	1
		4150	117	4	2			7078	543	24	2
		6409	125	0	3			4964	556	12	3
		5797	119	0	4			5852	565	8	4
		6084	118	4	Média			6615	553	15	Média

Tabela 1 - Resultados obtidos dos testes

Os dados originais podem ser vistos a partir do link a seguir:

Link: https://drive.google.com/file/d/1yTr3uaD-ndHW6zUbw76WHAv86_kHMsEz/view?usp=sharing

A Tabela 1 mostra os 3 diferentes tempos de execução que o comando *time* mostra ao final da execução do programa, *real*, *user* e *sys*. As definições de cada um são:

- **real:** Tempo de execução do início ao final da execução do programa, incluindo inclusive o tempo ocioso que o programa espera entradas do usuário;
- **user:** Tempo que a CPU levou para executar o processo. Neste tempo não se conta o período que o processo está bloqueado, nem tempos de outros processos;
- **sys:** Tempo que a CPU levou para executar o processo dentro do kernel.

No Gráfico 1 abaixo, há uma comparação dos tempos de execução médio usando o tempo *real* para cada caso testado.



Gráfico 1 - Tempos de execução *real* do código

Já nos Gráficos 2 e 3, foram testados os valores para $N = 100.00$ e $N = 500.00$, respectivamente:



Gráfico 2 - Tempos de execução *user* e *sys* do código para $N = 100.000$

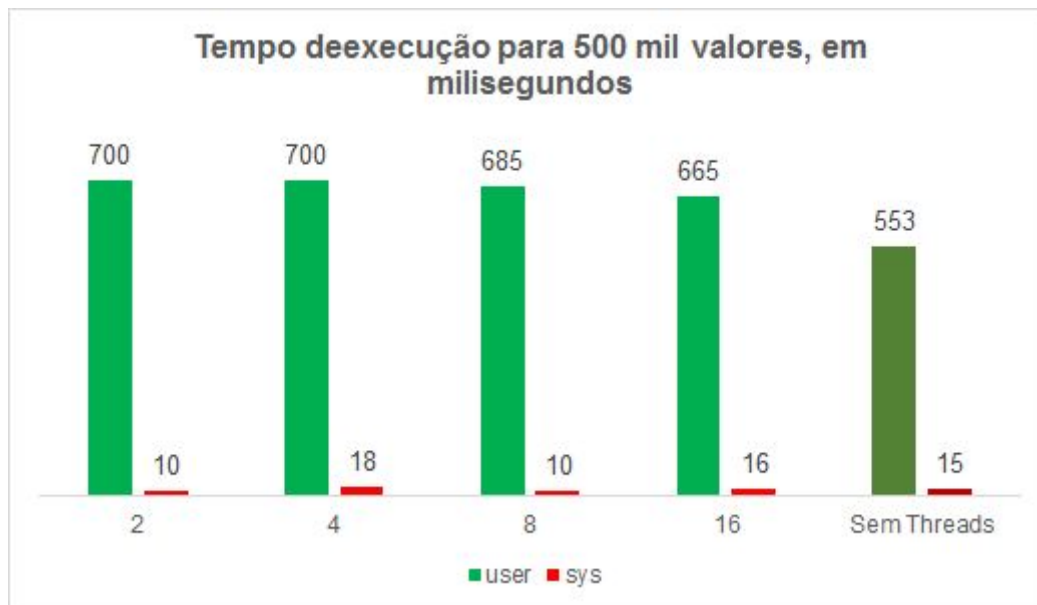


Gráfico 3 - Tempos de execução *user* e *sys* do código para N = 500.000

Os resultados obtidos serão discutidos na próxima seção. Os gráficos originais se encontram no mesmo link da tabela, também disponível a seguir:

Link: https://drive.google.com/file/d/1yTr3uaD-ndHW6zUbw76WHAv86_kHMsEz/view?usp=sharing

5. CONCLUSÃO

Caso olhássemos isoladamente para os valores do Gráfico 1, pensaríamos que há algumas situações mais rápidas do que outras para a execução do código. Mas, pelo atestado pelo grupo e vendo os outros tempos de execução, o maior influenciador nesta parte é o tempo que o próprio usuário leva para interagir com o código.

Os Gráficos 2 e 3 mostram que é mais rápido fazer o método de ordenação sem o uso de threads do que com elas quando leva-se em consideração o tempo *user*. Isto se deve, provavelmente, aos tempos de inicialização e de junção (*join*) de cada thread. No entanto, o uso de mais threads ainda parece ser um bom atenuante para o tempo. Quando olhamos o tempo *sys* vemos uma ligeira vantagem ao se usar threads, uma vez que as mesmas permitem processos paralelos que acabam tomando um tempo menor de execução. Nota-se, no entanto, um leve crescimento quando se vai de 8 para 16 threads. Provavelmente isso acontece devido ao número máximo de núcleos de processamento que o computador usado para testes tem (4 físicos, 8 lógicos).

Apesar do tempo de inicialização e de junção das threads ter interferido no tempo final de execução, concluímos que a programação multithread é vantajosa por permitir

processamentos simultâneos. No nosso código, por exemplo, o mesmo vetor é ordenado simultaneamente por pelo menos 2 threads e isso pode permitir otimizações muito maiores de tempo em projetos mais complexos.

6. REFERÊNCIAS

Uso da biblioteca *pthread.h*:

https://pt.wikibooks.org/wiki/Programação_Paralela_em_Arquiteturas_Multi-Core/Programação_em_Pthreads

Método de ordenação Merge Sort:

https://wikicoding.org/wiki/c/Merge_sort/

Modos de contagem de tempo do comando *time*:

<https://stackoverflow.com/questions/556405/what-do-real-user-and-sys-mean-in-the-output-of-time1>

7. CONTEÚDO ADICIONAL

Repositório GitHub do trabalho:

<https://github.com/rodrigomlima/SnackOverflow>

Todos os demais arquivos deste relatório estão no link abaixo:

<https://drive.google.com/drive/folders/1Cy38LPpRc2XyAp6bidRVrF3WR25y7XKX?usp=sharing>

8. ANEXOS

ANEXO 1 - arquivo.h

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<string.h>

// Função para o valor aleatório
double real_aleatorio();

// Funções para os arquivos
void cria_arquivo(int tam, char arq[100]);
int carrega_vetor(int tam, double vetor[tam], char arq[100]);
void salva_arquivo(int tam, double vetor[tam], char arq[100]);

// Funções auxiliares
void arruma_nome(char arq[]);
```

ANEXO 2 - arquivo.c

```
#include "arquivo.h"

// Declaração das funções

// Função para gerar números reais aleatórios
double real_aleatorio()
{
    // A multiplicação por 1000 faz com que se gere números reais de 0 a 1000
    return (((double)rand() / (double)RAND_MAX) * 1000);
}

void arruma_nome(char arq[])
{
    arq[strcspn(arq, "\n")] = 0; // Tira o '\n' da string
    strcat(arq, ".txt"); // Adiciona a extensão .txt do nome
}

// Função que cria o arquivo
void cria_arquivo(int tam, char arq[100])
{
    // Garante que o número será aleatório
    srand((unsigned)time(NULL));

    // Declaração de variáveis
    int i;
    double valor;
    FILE *vetor_real; // Arquivo

    // Abre o arquivo
    vetor_real = fopen(arq, "w");

    // Cria o vetor aleatório e salva no arquivo
    for (i = 0; i < tam; i++)
    {
        valor = real_aleatorio();
        fprintf(vetor_real, "%f ", valor);
    }

    // Fecha o arquivo
    fclose(vetor_real);
}

// Função que carrega os valores do arquivo num vetor
int carrega_vetor(int tam, double vetor[tam], char arq[100])
{
    // Declaração de variáveis
    int i;
```

```

FILE *vetor_real; // Arquivo

// Abertura do arquivo para leitura
vetor_real = fopen(arq, "r");

// Leitura de cada item do arquivo para carregar no vetor
for (i = 0; i < tam; i++)
{
    fscanf(vetor_real, "%lf ", &vetor[i]);
}

// Fecha o arquivo
fclose(vetor_real);

return 0;
}

// Função para salvar o vetor num arquivo
void salva_arquivo(int tam, double vetor[tam], char arq[100])
{
    // Declaração de variáveis
    int i;
    FILE *saida; // Arquivo

    // Abre o arquivo
    saida = fopen(arq, "w");

    // Cria o vetor aleatório e salva no arquivo
    for (i = 0; i < tam; i++)
    {
        fprintf(saida, "%f ", vetor[i]);
    }

    // Fecha o arquivo
    fclose(saida);
}

```

ANEXO 3 - mergesort.h

```
#include<stdio.h>
#include<stdlib.h>

void merge(double vetor[], int inicio, int meio, int fim);

void mergeSort(double vetor[], int inicio, int fim);
```

ANEXO 4 - mergesort.c

```
#include "mergesort.h"

// Para juntar vetores
void merge(double vetor[], int inicio, int meio, int fim)
{
    int i, j, k;
    int n1 = meio - inicio + 1;
    int n2 = fim - meio; // Para vetores temporários

    // Vetores temporários
    double temp_ini[n1], temp_fim[n2];

    // Copia os dados do vetor para os vetores temporários
    for (i = 0; i < n1; i++)
        temp_ini[i] = vetor[inicio + i];
    for (j = 0; j < n2; j++)
        temp_fim[j] = vetor[meio + 1 + j];

    // Junta os vetores temporários de volta ao vetor original
    i = 0; // Índice do primeiro vetor
    j = 0; // Índice do segundo vetor
    k = inicio; // Índice do vetor original
    while (i < n1 && j < n2)
    {
        if (temp_ini[i] <= temp_fim[j])
        {
            vetor[k] = temp_ini[i];
            i++;
        }
        else
        {
            vetor[k] = temp_fim[j];
            j++;
        }
        k++;
    }

    // Copia os elementos faltantes de temp_ini[], se houver
    while (i < n1)
    {
        vetor[k] = temp_ini[i];
        i++;
        k++;
    }

    // Copia os elementos faltantes de temp_fim[], se houver
    while (j < n2)
    {

```



```

        vetor[k] = temp_fim[j];
        j++;
        k++;
    }
}

// Função específica do Merge Sort
void mergeSort(double vetor[], int inicio, int fim)
{
    if (inicio < fim)
    {
        // Calcula o meio
        int meio = inicio + (fim - inicio) / 2;

        // Ordena as duas metades
        mergeSort(vetor, inicio, meio);
        mergeSort(vetor, meio + 1, fim);

        // Une as 2 metades
        merge(vetor, inicio, meio, fim);
    }
}

```

ANEXO 5 - makefile

```
# Arquivo Makefile para compilar o programa de ordenação de vetores com
multithreads
#
# Compilador padrao
CC=gcc
#
# Arquivos fonte
FONTES=projSO.c arquivo.c mergesort.c
OBJETOS=$(FONTES:.c=.o)
EXECUTAVEL=projSO

#Dependencias de Compilacao
all: $(EXECUTAVEL)

projSO: projSO.c arquivo.o mergesort.o
    $(CC) -o projSO projSO.c arquivo.o mergesort.o -lpthread

arquivo.o: arquivo.c arquivo.h
    $(CC) -c arquivo.c

mergesort.o: mergesort.c mergesort.h
    $(CC) -c mergesort.c

#Limpeza
clean:
    rm -f $(OBJETOS) $(EXECUTAVEL)
```