Relatório Projeto Prático
Programação com Múltiplos Threads

Sistemas Operacionais

2º semestre 2024

Docente:

André Leon S. Gradvohl

Discentes:

Davi Paiva Souza RA:222481

Murillo Martins Proveza RA:215700

Sumário

1. Introdução

O projeto prático da disciplina tem como objetivo ampliar e consolidar os conhecimentos sobre programação com múltiplos threads.

A proposta consiste no desenvolvimento de um programa em linguagem C que, compatível com sistemas operacionais Linux, utilize múltiplos threads através da biblioteca POSIX Threads (<pthread>). Este programa deverá manipular arquivos de entrada e saída, realizando a leitura de valores inteiros de vários arquivos, e ordená-los de forma crescente em um único arquivo de saída.

Esse desafio envolve não só o uso eficiente de threads para otimizar o processamento, mas também a implementação de algoritmos de ordenação, contribuindo para o domínio prático das técnicas de paralelismo e sincronização de threads.

2. Descrição do problema

O problema proposto envolve a leitura de vários arquivos texto, onde cada arquivo contém números inteiros dispostos um por linha.

A tarefa é ordenar todos esses números em ordem crescente, utilizando múltiplos threads de execução (2, 4 ou 8). Após a ordenação, o programa deve gerar um único arquivo de saída que reúna todos os números dos arquivos de entrada, ordenados de forma crescente. Utilizando algoritmos de ordenação otimizados para trabalhar com múltiplos threads, visando aumentar a eficiência.

3.Implementação

Para o desenvolvimento deste projeto, foi criado um programa em C que atende aos requisitos definidos na descrição do problema, como o uso de múltiplas threads para ordenação de números contidos em vários arquivos de entrada.

Em sua criação foram utilizadas as ferramentas: Replit, GitHub, Vscode e VirtualBox.

O código foi organizado de maneira modular para facilitar a compreensão, o desenvolvimento e possíveis ajustes ao longo do projeto. A seguir, são descritos os principais módulos e componentes:

3.1 Bibliotecas

Nessa seção, são listadas e explicadas as bibliotecas utilizadas:

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

<pthread.h>: Esta biblioteca pertence a POSIX Threads, responsável pela manipulação dos threads que serão utilizados no programa.

Ela é capaz de realizar a criação, edição e sincronização de threads. Funções como **pthread_create** e **pthread_join**, pertencentes no desenvolvimento do projeto, são derivadas desta biblioteca.

<stdio.h>: Esta biblioteca é a padrão de entrada e saída em C, permitindo a inserção de dados com a função **scanf** e exibição na tela com a função **printf** por exemplo. Também é utilizada fornecendo funções de manipulação de arquivos como **fopen** que é responsável pela abertura de um arquivo, **fscanf** responsável pela leitura dos dados de um arquivo e o **fprintf** que é responsável por gravar dados no arquivo.

<stdlib.h>: Esta biblioteca foi utilizada neste projeto para alocação dinâmica de memória contendo funções como **malloc** e **free**

<string.h>: Esta biblioteca contém diversas funções responsáveis pela manipulação de strings Ela inclui funções como **strlen** para obter o comprimento de uma string e **strcpy** para copiar strings.

<time.h>: Por fim a biblioteca que é responsável por captar o tempo de execução dos threads. Inclui funções como **time**, **clock** e **clock_gettime**.

3.2 Instruções

Para compilar use:
gcc -Wall -o mergesort mergesort.c funcoes.c -lpthread

Para testar com os arquivos use:
Teste 1
./mergesort 2 nmrs1.dat nmrs2.dat nmrs3.dat nmrs4.dat nmrs5.dat -o saida.txt

Teste 2
./mergesort 4 nmrs1.dat nmrs2.dat nmrs3.dat nmrs4.dat nmrs5.dat -o saida.txt

Teste 3
./mergesort 8 nmrs1.dat nmrs2.dat nmrs3.dat nmrs4.dat nmrs5.dat -o saida.txt

3.3 Funções

Foram desenvolvidas diversas funções para atender as necessidades do programa, sendo elas:

```c
int comparar(const void *a, const void *b) {
  return (*(int *)a - *(int *)b);
}
```

Comparar: A função comparar é usada como uma função de comparação para a ordenação de números inteiros em ordem crescente. Ela é utilizada pela função **qsort** para determinar a ordem dos elementos no array de inteiros, facilitando a organização dos números na memória antes que eles sejam escritos no arquivo final.

```c
void *ordenar_thread(void *arg) {
  DadosThread *dados = (DadosThread *)arg;
  struct timespec inicio, fim;
  clock_gettime(CLOCK_MONOTONIC, &inicio);

  qsort(dados->vetor, dados->tamanho, sizeof(int), comparar);

  clock_gettime(CLOCK_MONOTONIC, &fim);
  dados->tempo_execucao = (fim.tv_sec - inicio.tv_sec) + (fim.tv_nsec - inicio.tv_nsec) / 1e9;

  pthread_exit(NULL);
}
```

Ordenar_thread: A função ordenar_thread é responsável por permitir que cada thread ordene um bloco do vetor de inteiros, dividindo o trabalho de ordenação entre as threads para otimizar o tempo de execução do programa.

Ela utiliza a função **qsort** para ordenar o bloco de inteiros associado a cada thread. Além disso, essa função mede o tempo de execução da ordenação de cada thread individualmente, armazenando-o para que possa ser exibido e somado posteriormente para calcular o tempo total de execução do programa.

```c
// Função para ler números de um arquivo e armazená-los em um vetor
int *ler_arquivo(const char *nome_arquivo, int *tamanho) {
  FILE *arquivo = fopen(nome_arquivo, "r");
  if (arquivo == NULL) {
    perror("Erro ao abrir o arquivo");
    exit(EXIT_FAILURE);
  }

  int *vetor = NULL;  // Ponteiro inicial nulo
  int contador = 0;

  int valor;
  while (fscanf(arquivo, "%d", &valor) == 1) {
    // Realoca o vetor para um espaço adicional para o novo número
    int *temp = realloc(vetor, (contador + 1) * sizeof(int));
    if (temp == NULL) {
      perror("Erro ao redimensionar memória");
      free(vetor);
      fclose(arquivo);
      exit(EXIT_FAILURE);
    }
    vetor = temp;

    vetor[contador] = valor;
    contador++;
  }

  *tamanho = contador;
  fclose(arquivo);
  return vetor;
}
```

ler_arquivo: A função ler_arquivo abre um arquivo especificado no modo de leitura e lê todos os números inteiros contidos nele, armazenando-os em um vetor alocado dinamicamente. Caso o arquivo não possa ser aberto, a função exibe uma mensagem de erro e encerra o programa.

Durante a leitura, para cada número lido, o vetor é redimensionado dinamicamente utilizando **realloc** para adicionar uma nova posição, sem uma capacidade inicial predefinida. Se **realloc** falhar, a função exibe uma mensagem de erro, libera a memória previamente alocada e encerra o programa.

Ao final do processo, a função armazena o número total de elementos lidos na variável apontada por tamanho e retorna um ponteiro para o vetor contendo todos os números lidos.

```c
void mesclar_blocos_ordenados(int *todos_numeros, int total_numeros, int num_threads, int tamanho_bloco, int *vetor_ordenado) {
  int *indices = malloc(num_threads * sizeof(int));
  memset(indices, 0, num_threads * sizeof(int));

  for (int i = 0; i < total_numeros; i++) {
    int menor_valor = __INT_MAX__;
    int indice_menor = -1;

    for (int j = 0; j < num_threads; j++) {
      int inicio = j * tamanho_bloco;
      int fim = (j == num_threads - 1) ? total_numeros : inicio + tamanho_bloco;

      if (indices[j] < (fim - inicio)) {
        int valor_atual = todos_numeros[inicio + indices[j]];
        if (valor_atual < menor_valor) {
          menor_valor = valor_atual;
          indice_menor = j;
        }
      }
    }

    vetor_ordenado[i] = menor_valor;
    indices[indice_menor]++;
  }
```

mesclar_blocos_ordenados: A função mesclar_blocos_ordenados é responsável por combinar blocos previamente ordenados em um único vetor ordenado.

Cada bloco representa uma seção do vetor todos_numeros, que foi ordenado por diferentes threads. A função utiliza um algoritmo de **merge** para selecionar o menor valor entre os blocos em cada iteração e colocá-lo no próximo espaço disponível em vetor_ordenado. O objetivo é gerar um único vetor ordenado contendo todos os números dos blocos originais. Essa função também utiliza um vetor de índices para rastrear a posição atual de cada bloco durante o processo de mesclagem. Ao final, o vetor ordenado conterá todos os números ordenados, prontos para serem escritos no arquivo de saída.

3.4 Estrutura

```c
#ifndef FUNCOES_H
#define FUNCOES_H

// Estrutura para armazenar informações de cada thread
typedef struct {
  int *vetor;           // Ponteiro para o vetor de números que a thread irá ordenar
  int tamanho;          // Tamanho do vetor que a thread irá processar
  int id_thread;        // ID da thread, para identificação
  double tempo_execucao; // Tempo de execução da thread, armazenado para análise de desempenho
} DadosThread;

// Função para comparar dois elementos, usada para ordenação
int comparar(const void *a, const void *b);

// Função para ordenar um vetor de números em uma thread
void *ordenar_thread(void *arg);

// Função para ler números de um arquivo e armazená-los em um vetor
int *ler_arquivo(const char *nome_arquivo, int *tamanho);

// Função para mesclar blocos ordenados de números em um único vetor ordenado
void mesclar_blocos_ordenados(int *todos_numeros, int total_numeros, int num_threads, int tamanho_bloco, int *vetor_ordenado);

#endif
```

Arquivo funcoes.h: Este arquivo define uma estrutura chamada DadosThread e declara as funções necessárias para a execução de threads e manipulação de dados para ordenação e mesclagem. A estrutura DadosThread armazena informações essenciais para o processamento paralelo, incluindo um ponteiro para o vetor de números a ser ordenado pela thread, o tamanho desse vetor, um identificador para a thread e o tempo de execução da thread para análise de desempenho.

Além disso, o arquivo declara funções para:

- Comparar elementos para ordenação (comparar).
- Ordenar um vetor em uma thread específica (ordenar_thread).
- Ler números de um arquivo e armazená-los em um vetor (ler_arquivo).
- Mesclar blocos ordenados de números em um único vetor ordenado (mesclar_blocos_ordenados).

3.5 Principal (main)

```c
int main(int argc, char *argv[]) {

  // Verifica os argumentos da linha de comando
  if (argc < 4)
  {
    fprintf(stderr, "Uso: %s <num_threads> <arquivo1> <arquivo2> ... -o <arquivo_saida>\n", argv[0]);
    return 1;
  }

  // Converte o primeiro argumento para o número de threads desejado
  int num_threads = atoi(argv[1]);

  // Verifica se o número de threads é válido (2, 4 ou 8)
  if (num_threads != 2 && num_threads != 4 && num_threads != 8)
  {
    fprintf(stderr, "Erro: O número de threads deve ser 2, 4 ou 8.\n");
    return 1;
  }

  // Define o número de arquivos de entrada e o nome do arquivo de saída
  int num_arquivos = argc - 4;
  char *arquivo_saida = argv[argc - 1];
  char **arquivos_entrada = &argv[2];
```

Esta seção do código **main** inicia a sua execução verificando se todos os argumentos da linha de comando foram inseridos corretamente conforme o programado para ser escrito. Após é feita a leitura e verificação da quantidade de threads desejada pelo usuário, se não for 2, 4 ou 8, exibe uma mensagem de erro e encerra o programa. E também é compreendido o número de arquivos de entrada e definido o nome do arquivo de saída

```c
  // Inicializa variáveis para armazenar os dados dos arquivos
  int total_numeros = 0;
  int **vetores = malloc(num_arquivos * sizeof(int *));
  int *tamanhos = malloc(num_arquivos * sizeof(int));

  // Lê cada arquivo e armazena seus dados em vetores
  for (int i = 0; i < num_arquivos; i++) {
    vetores[i] = ler_arquivo(arquivos_entrada[i], &tamanhos[i]);
    total_numeros += tamanhos[i];
  }

  // Cria um vetor único para armazenar todos os números
  int *todos_numeros = malloc(total_numeros * sizeof(int));
  int posicao = 0;

  // Copia os dados de cada vetor individual para o vetor único
  for (int i = 0; i < num_arquivos; i++) {
    memcpy(&todos_numeros[posicao], vetores[i], tamanhos[i] * sizeof(int));
    posicao += tamanhos[i];
  }

  // Define o tamanho de cada bloco para cada thread
  int tamanho_bloco = total_numeros / num_threads;
```

A próxima parte do código aloca dinamicamente variáveis para armazenar os dados (números) lidos dos arquivos. Em seguida, utiliza um loop **for** para ler os arquivos de entrada e armazena os dados de cada arquivo em vetores separados.

Após a leitura dos arquivos, um vetor único é alocado dinamicamente para armazenar todos os números juntos. Um segundo loop **for** copia os dados dos vetores individuais para o vetor único, consolidando todos os números em uma única

estrutura. Por fim, é definido o tamanho de cada bloco de dados que cada thread irá processar, com base no número total de números e na quantidade de threads.

```c
// Aloca memória dinamicamente para as threads e dados de cada thread
pthread_t *threads = malloc(num_threads * sizeof(pthread_t));
DadosThread *dados_thread = malloc(num_threads * sizeof(DadosThread));

// Marca o tempo de início do processamento
struct timespec inicio_total, fim_total;
clock_gettime(CLOCK_MONOTONIC, &inicio_total);

// Cria threads para ordenar cada bloco
for (int i = 0; i < num_threads; i++) {
  dados_thread[i].vetor = &todos_numeros[i * tamanho_bloco];
  dados_thread[i].id_thread = i;
  dados_thread[i].tamanho = (i == num_threads - 1) ? total_numeros - (i * tamanho_bloco) : tamanho_bloco;
  pthread_create(&threads[i], NULL, ordenar_thread, (void *)&dados_thread[i]);
}

// Espera todas as threads terminarem e exibe o tempo de execução de cada uma
for (int i = 0; i < num_threads; i++) {
  pthread_join(threads[i], NULL);
  printf("Tempo de execução do Thread %d: %.5f segundos.\n", dados_thread[i].id_thread, dados_thread[i].tempo_execucao);
}

// Marca o tempo de fim do processamento
clock_gettime(CLOCK_MONOTONIC, &fim_total);
double tempo_execucao_total = (fim_total.tv_sec - inicio_total.tv_sec) + (fim_total.tv_nsec - inicio_total.tv_nsec) / 1e9;
printf("Tempo total de execução: %.5f segundos.\n", tempo_execucao_total);
```

A próxima parte do código realiza a alocação dinâmica para as threads que serão utilizadas e inicia a contagem do tempo de processamento com a função **clock_gettime**. Em seguida, as threads são criadas em um loop **for**, recebendo os blocos de dados com os quais cada uma irá trabalhar.

Após a execução das threads, o tempo de execução individual de cada uma é exibido, e a contagem do tempo total é encerrada e somada, mostrando o tempo total de execução.

```c
  // Mescla os blocos ordenados em um vetor final ordenado
  int *vetor_ordenado = malloc(total_numeros * sizeof(int));
  mesclar_blocos_ordenados(todos_numeros, total_numeros, num_threads, tamanho_bloco, vetor_ordenado);

  // Grava o vetor ordenado no arquivo de saída
  FILE *saida = fopen(arquivo_saida, "w");
  if (saida == NULL) {
    perror("Erro ao abrir o arquivo de saída");
    exit(EXIT_FAILURE);
  }
  for (int i = 0; i < total_numeros; i++) {
    fprintf(saida, "%d\n", vetor_ordenado[i]);
  }
  fclose(saida);

  // Libera a memória alocada
  for (int i = 0; i < num_arquivos; i++) {
    free(vetores[i]);
  }
  free(vetores);
  free(tamanhos);
  free(todos_numeros);
  free(vetor_ordenado);
  free(threads);
  free(dados_thread);

  return 0;
}
```

Por fim, o código mescla os blocos ordenados separadamente em um único vetor ordenado utilizando a função **mesclar_blocos_ordenados**. Em seguida, ele cria o arquivo de saída com o nome especificado na linha de comando e grava o vetor totalmente ordenado neste arquivo. Caso ocorra algum erro na criação ou abertura do arquivo de saída, o programa exibe uma mensagem de erro e encerra sua execução.

Após concluir a gravação, o programa fecha o arquivo de saída e libera toda a memória alocada, garantindo a finalização correta e a liberação dos recursos do sistema.

4. Entradas

Referem-se aos arquivos e a linha de comando que foram utilizadas para desenvolvimento dos testes do programa. Sendo:

4.1. Arquivos de entrada

Foram utilizados 5 arquivos com um total de 1000 números inteiros ordenados de maneira totalmente aleatória, os nomes dos arquivos utilizados foram: **nmrs1.dat, nmrs2.dat, nmrs3.dat, nmrs4.dat, nmrs5.dat** (todos disponíveis no repositório do Github).

4.2. Compilação

A compilação foi feita utilizando o GCC juntamente com alguns comandos adicionais:

gcc -Wall -o mergesort mergesort.c funcoes.c -lpthread

Sendo:
gcc: Compilador utilizado;
-Wall: Ativa os warnings da compilação;
-o: Utilizado para especificar o arquivo de "saída";
mergesort: Arquivo de programa compilado;
mergesort.c funcoes.c: Arquivos .c compilados;
-lpthread: Usado para vincular a biblioteca pthread.

5.Testes

1. No primeiro teste foram usados os 5 arquivos de teste, utilizando 2 threads para realizar a ordenação com a seguinte linha de comando:

./mergesort 2 nmrs1.dat nmrs2.dat nmrs3.dat nmrs4.dat nmrs5.dat -o saida.txt

Explicação:

./mergesort: Nome do arquivo para executar;

2: Número de threads;

nmrs1.dat nmrs2.dat nmrs3.dat nmrs4.dat nmrs5.dat: Nome dos arquivos de teste;

-o saida.txt: Identificar o arquivo de saída + nome do arquivo.

2. Foi-se utilizada a mesma linha de código e mesmos arquivos, somente com a mudança no número de threads, ao invés de 2, usou-se 4. Sendo:

./mergesort 4 nmrs1.dat nmrs2.dat nmrs3.dat nmrs4.dat nmrs5.dat -o saida.txt

3. Foi-se utilizada a mesma linha de código e mesmos arquivos, somente com a mudança no número de threads, ao invés de 4, usou-se 8. Sendo:

./mergesort 8 nmrs1.dat nmrs2.dat nmrs3.dat nmrs4.dat nmrs5.dat -o saida.txt

6. Resultados

Teste 1:

```
Tempo de execução do Thread 0: 0.01436 segundos.
Tempo de execução do Thread 1: 0.01445 segundos.
Tempo total de execução: 0.03045 segundos.
```

Teste 2:

```
Tempo de execução do Thread 0: 0.00019 segundos.
Tempo de execução do Thread 1: 0.00015 segundos.
Tempo de execução do Thread 2: 0.00014 segundos.
Tempo de execução do Thread 3: 0.00013 segundos.
Tempo total de execução: 0.00423 segundos.
```

Teste 3:

```
Tempo de execução do Thread 0: 0.00009 segundos.
Tempo de execução do Thread 1: 0.00006 segundos.
Tempo de execução do Thread 2: 0.00006 segundos.
Tempo de execução do Thread 3: 0.00006 segundos.
Tempo de execução do Thread 4: 0.00006 segundos.
Tempo de execução do Thread 5: 0.00005 segundos.
Tempo de execução do Thread 6: 0.00005 segundos.
Tempo de execução do Thread 7: 0.00005 segundos.
Tempo total de execução: 0.00354 segundos.
```

Em todos os testes, foi-se criado o seguinte arquivo de saída:

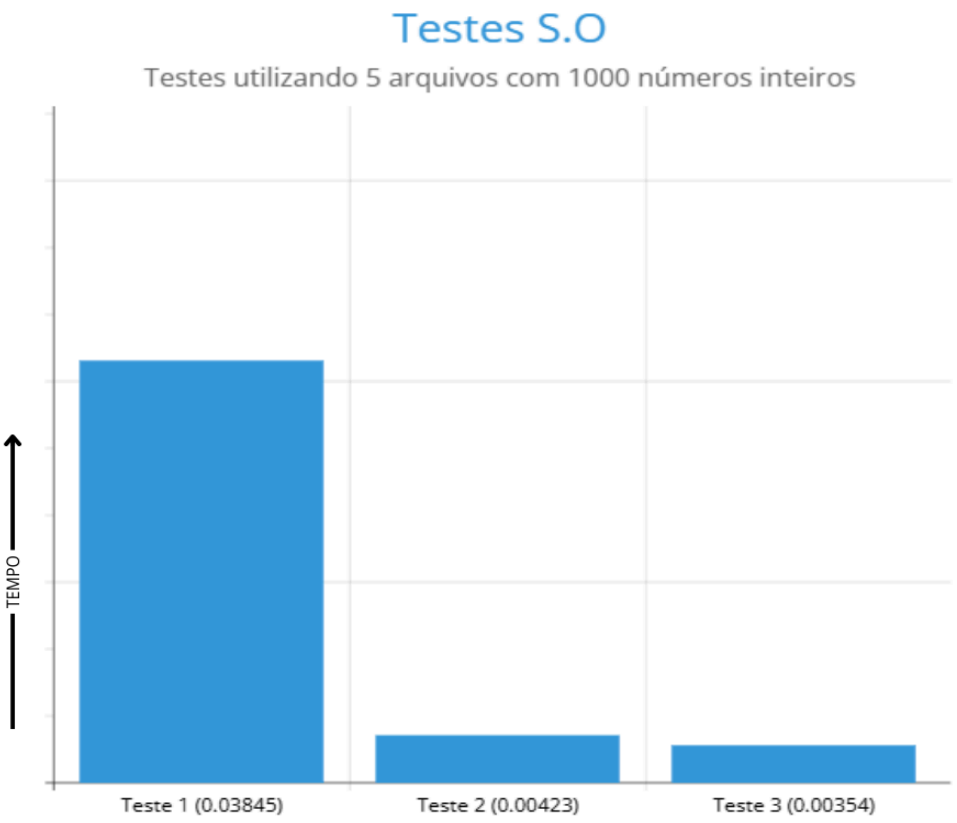
saida.txt

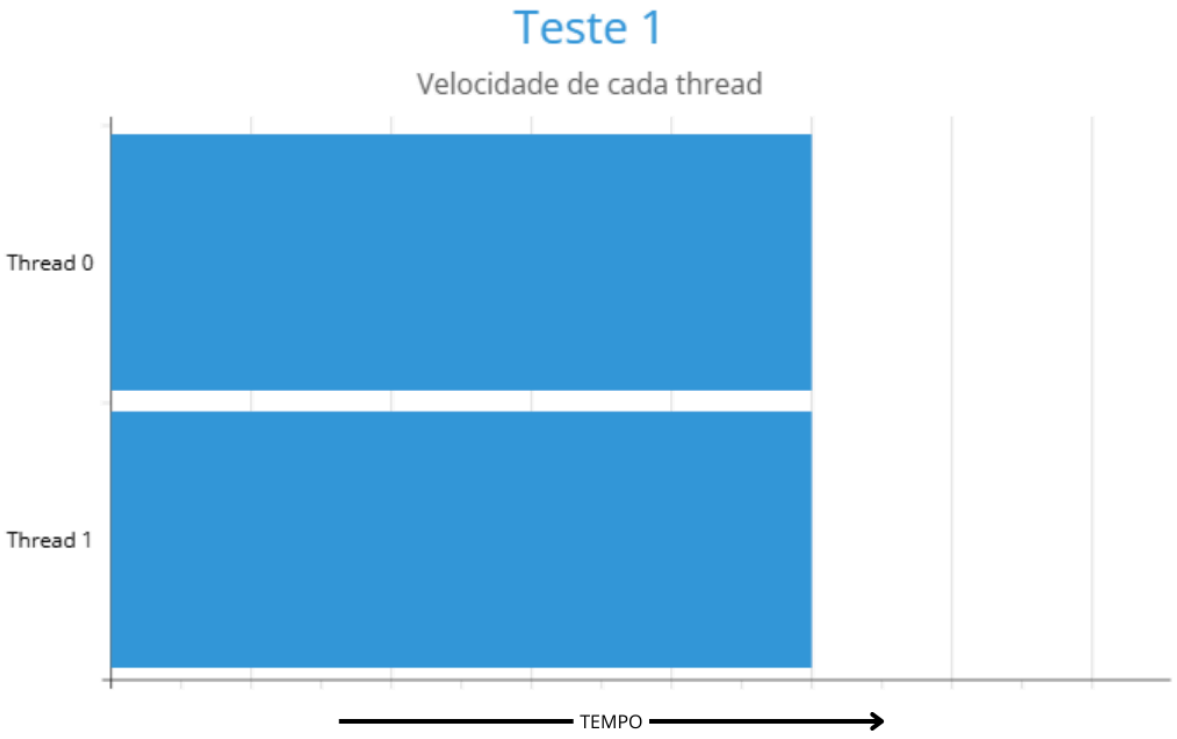
1	1
2	1
3	1
4	1
5	1
6	2
7	2
8	2
9	2
10	2
11	3
12	3
13	3
14	3
15	3
16	4
17	4
18	4
19	4
20	4

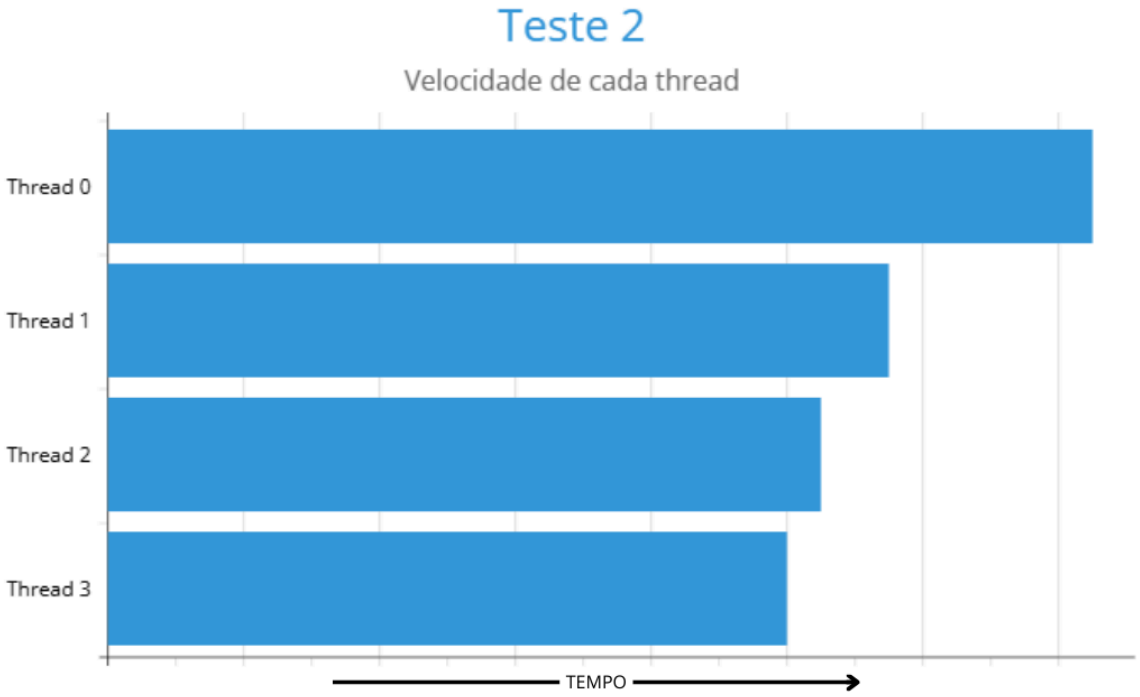
[...]

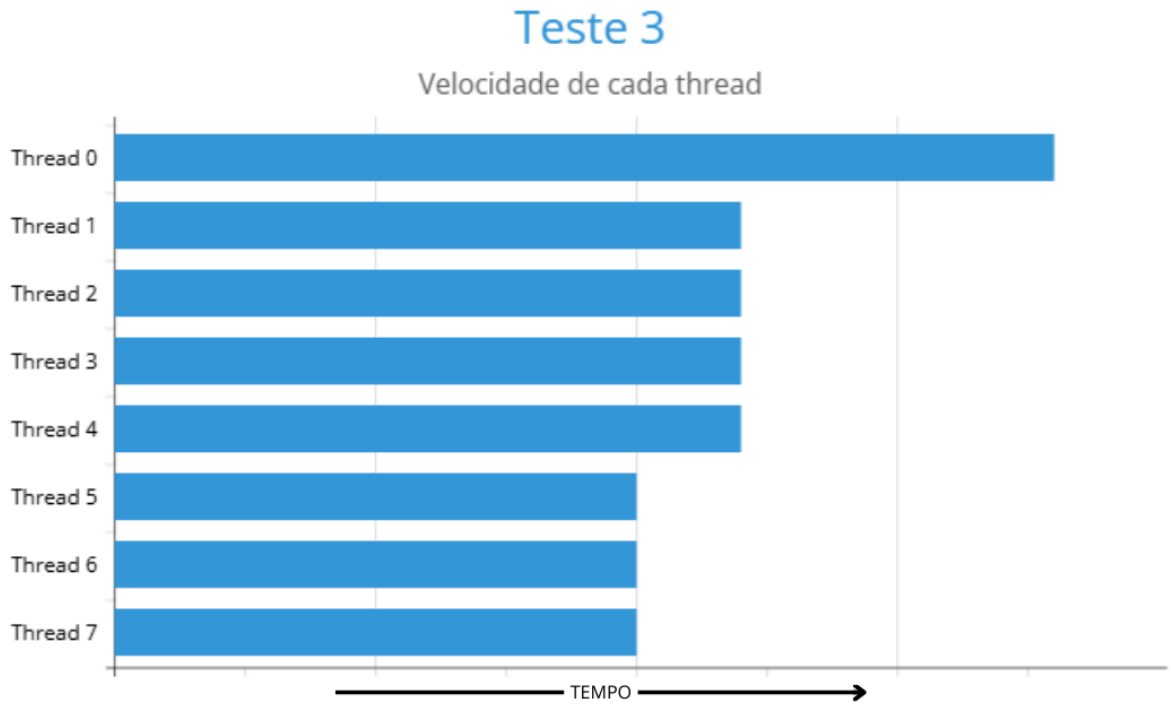
4981	997
4982	997
4983	997
4984	997
4985	997
4986	998
4987	998
4988	998
4989	998
4990	998
4991	999
4992	999
4993	999
4994	999
4995	999
4996	1000
4997	1000
4998	1000
4999	1000
5000	1000

7. Gráficos


Testes S.O
Testes utilizando 5 arquivos com 1000 números inteiros
TEMPO
Teste 1 (0.03845)
Teste 2 (0.00423)
Teste 3 (0.00354)


Teste 1
Velocidade de cada thread
Thread 0
Thread 1
TEMPO


Teste 2
Velocidade de cada thread
Thread 0
Thread 1
Thread 2
Thread 3
TEMPO


Teste 3
Velocidade de cada thread
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
TEMPO

8. Conclusão

Concluindo, o projeto implementa uma solução eficiente para leitura, processamento paralelo de threads e ordenação de dados numéricos a partir de arquivos de entrada, utilizando multithreading para otimizar o tempo de execução.

A divisão dos dados em blocos, distribuídos entre várias threads para um processamento simultâneo, e a subsequente mesclagem dos blocos ordenados em um único vetor, permite que o programa processe grandes volumes de dados de forma mais rápida. Além disso, o código inclui verificações de integridade dos parâmetros de entrada e de operação de arquivos, além de uma gestão eficiente de memória, garantindo confiabilidade e desempenho.

Essa abordagem exemplifica o uso de técnicas de programação paralela e alocação dinâmica de memória para resolver problemas de ordenação de dados, com aplicação potencial em diversas áreas que exigem processamento rápido e eficiente de grandes conjuntos de dados.

9. Endereços dos Documentos

Repositório GitHub:
https://github.com/MurilloMartinsProveza/Trabalho-de-SO

Vídeo Youtube:
https://youtu.be/OWbU3tQnnwc

10. Referências

- Playlist de vídeos sobre a biblioteca POSIX threads. Canal CodeVault: https://www.youtube.com/watch?v=d9s_d28yJq0&list=PLfqABt5AS4FmuQf70psXrsMLEDQXNkLq2&pp=iAQB

- Video sobre a função clock_gettime. Canal CodeVault: https://www.youtube.com/watch?v=mSUChCEE-rs

- Documento de passagem de argumentos pela lista de comandos Linux - GitHub - gradvohl/ArgsLinhaComando: Breve tutorial para a passagem de argumentos pela linha de comando para programas na linguagem C.

- Documentação realizada pela IBM sobre a função clock_gettime: https://www.ibm.com/docs/en/zos/3.1.0?topic=functions-clock-gettime-retrieve-time-specified-clock