UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE TECNOLOGIA

UNIVERSIDADE ESTADUAL DE CAMPINAS FACULDADE DE TECNOLOGIA - FT

TT304A - Sistemas Operacionais

Projeto Prático da disciplina

Grupo X-men Evolution

Adriano Baumgarte Bassani Filho **288824**

João Vitor Da Fraga Anacleto **269175**

Prof. Dr. André Leon S. Gradvohl

Limeira - SP

Programa de Ordenação Paralela com Threads

Descrição:

Este programa realiza a ordenação de números inteiros lidos a partir de arquivos de entrada. A ordenação é paralelizada, utilizando múltiplas threads para dividir a tarefa, o que melhora o desempenho em sistemas com múltiplos núcleos de CPU. Cada thread ordena uma parte dos dados, e, no final, a última thread realiza uma ordenação global dos dados combinados.

Este projeto é dividido em múltiplos arquivos (`mergesort.c`, `ordenacao.c`, `ordenacao.h`, `arq1.dat`, `arq2.dat`, `arq3.dat`, `arq4.dat`, `arq5.dat`, `makefile` e `saida.dat`) para uma organização clara e modular do código.

Estrutura dos Arquivos

- **mergesort.c:** Contém a função `main`, onde ocorre a configuração inicial, leitura dos arquivos, criação das threads, sincronização e exibição dos tempos de execução.
- **ordenacao.c:** Implementa as funções de leitura, ordenação e criação de threads, além de lidar com a ordenação paralela e gravação dos resultados no arquivo de saída.
- **ordenacao.h:** Cabeçalho que declara as estruturas de dados e as funções públicas usadas em `mergesort.c` e `ordenacao.c`.
- **arq1.dat, arq2.dat, arq3.dat, arq4.dat e arq5.dat:** Arquivos que armazenam 1000 inteiros fora de ordem cada.
- **makefile:** Arquivo para facilitar a compilação apenas usando `make` no terminal.
- **saida.dat:** Arquivo que agrega todos os inteiros de todos os arquivos da execução de forma ordenada.

1. COMPILAÇÃO

Requisitos:

- GCC (GNU Compiler Collection)
- Biblioteca pthread (nativa no Linux)

Comandos de Compilação:

Você pode compilar o programa manualmente com:

```
Unset
gcc mergesort.c ordenacao.c -o mergesort -lpthread
```

ou usando o Makefile com o seguinte comando no terminal:

```
Unset
make
```

2. EXECUÇÃO

Após a compilação, execute o programa com o comando:

```
Unset
./mergesort <num_threads> <arquivo1> <arquivo2> ... -o <arquivo_saida
```

- `<num_threads>`: Número de threads para dividir a tarefa de ordenação.
- `<arquivo1> <arquivo2> ...`: Arquivos de entrada contendo números inteiros (um por linha).
- `-o <arquivo_saida>`: Arquivo de saída para salvar o resultado ordenado.

Exemplo de uso:

```
Unset
./mergesort 4 dados1.dat dados2.dat -o saida.dat
```

3. ESTRUTURA E EXPLICAÇÃO DO CÓDIGO

ordenacao.h

Este cabeçalho declara as estruturas e funções utilizadas para ordenação paralela.

thread_args: Define os argumentos que cada thread recebe, como o intervalo de dados que irá ordenar, seu ID, e o número total de threads.

Variáveis:

- `int* numeros`: Vetor dinâmico que armazena todos os números lidos dos arquivos.
- `int total_numeros`: Total de números armazenados em `numeros`.

Funções Declaradas:

- `int ler_arquivos(int argc, char* argv[])`: Lê os arquivos de entrada e armazena os dados no vetor `numeros`.
- `void* ordenar_dados(void* args)`: Função executada por cada thread para ordenar uma parte do vetor `numeros`.
- `void criar_threads_para_ordenacao(int num_threads, pthread_t* threads, thread_args* args)`: Cria e inicializa as threads para dividir o trabalho de ordenação.
- `int gravar_saida(const char* arquivo_saida)`: Grava o vetor ordenado no arquivo de saída.

ordenacao.c

Contém as implementações das funções de leitura, ordenação e criação de threads.

- `int ler_arquivos(int argc, char* argv[])`: Lê números de inteiros a partir dos arquivos fornecidos e armazena no vetor `numeros`.
- `void* ordenar_dados(void* args)`: Ordena uma porção dos dados e mede o tempo de execução individual de cada thread.

  A lógica de comparação usada no código foi retirada daqui.

- `void criar_threads_para_ordenacao(int num_threads, pthread_t* threads, thread_args* args)`: Divide o trabalho entre as threads.
- `int gravar_saida(const char* arquivo_saida)`: Grava os dados ordenados no arquivo de saída.

mergesort.c

Contém a função `main`, onde ocorre a configuração inicial e a execução do programa**.**

Fluxo da Função main

- Verificação de Argumentos: Verifica se os argumentos fornecidos são suficientes.
- Leitura dos Arquivos: Carrega os números dos arquivos de entrada.
- Criação das Threads: Inicia as threads para a ordenação.
- Sincronização das Threads: Garante que todas as threads terminem.
- Exibição do Tempo de Execução de Cada Thread: Exibe o tempo gasto por thread.
- Exibição do Tempo Total de Execução: Calcula e exibe o tempo total.
- Gravação dos Dados Ordenados: Salva o vetor ordenado no arquivo de saída.
- Liberação de Memória: Libera o vetor `numeros`.

Mensagens de Erro

Falha ao alocar memória: Caso `malloc` ou `realloc` falhe.

Erro ao abrir arquivo: Quando um arquivo de entrada ou de saída não pode ser acessado.

4. TESTES

Compilação no terminal:

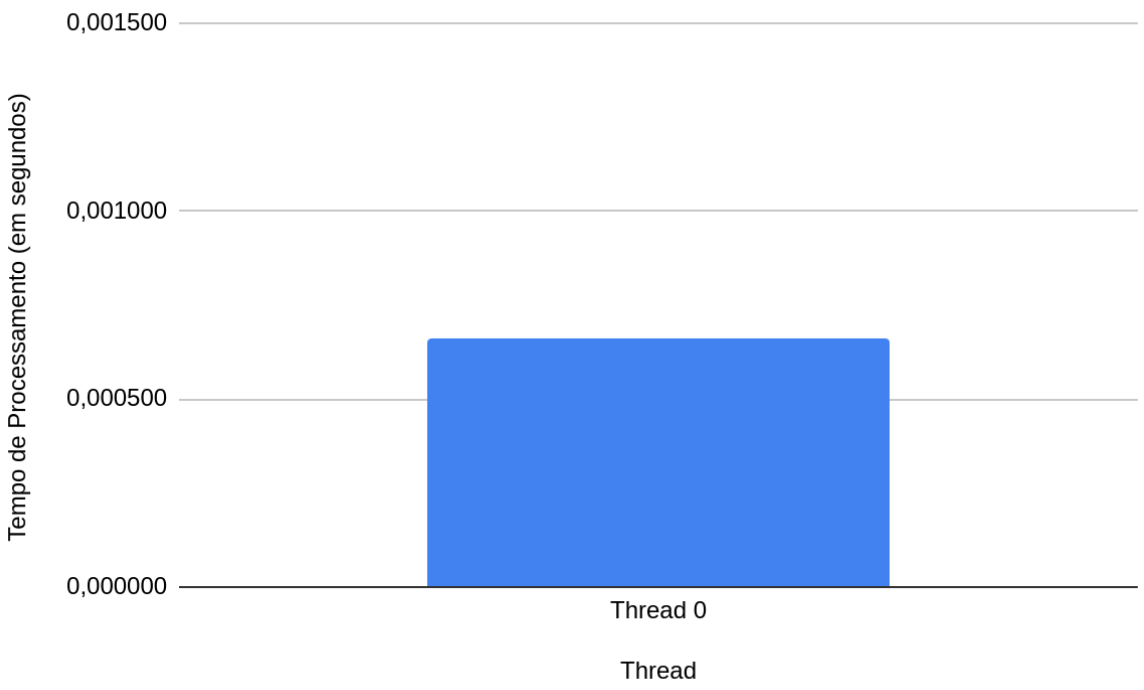
```
Unset
make
```

ou

```
Unset
gcc mergesort.c ordenacao.c -o mergesort -lpthread
```

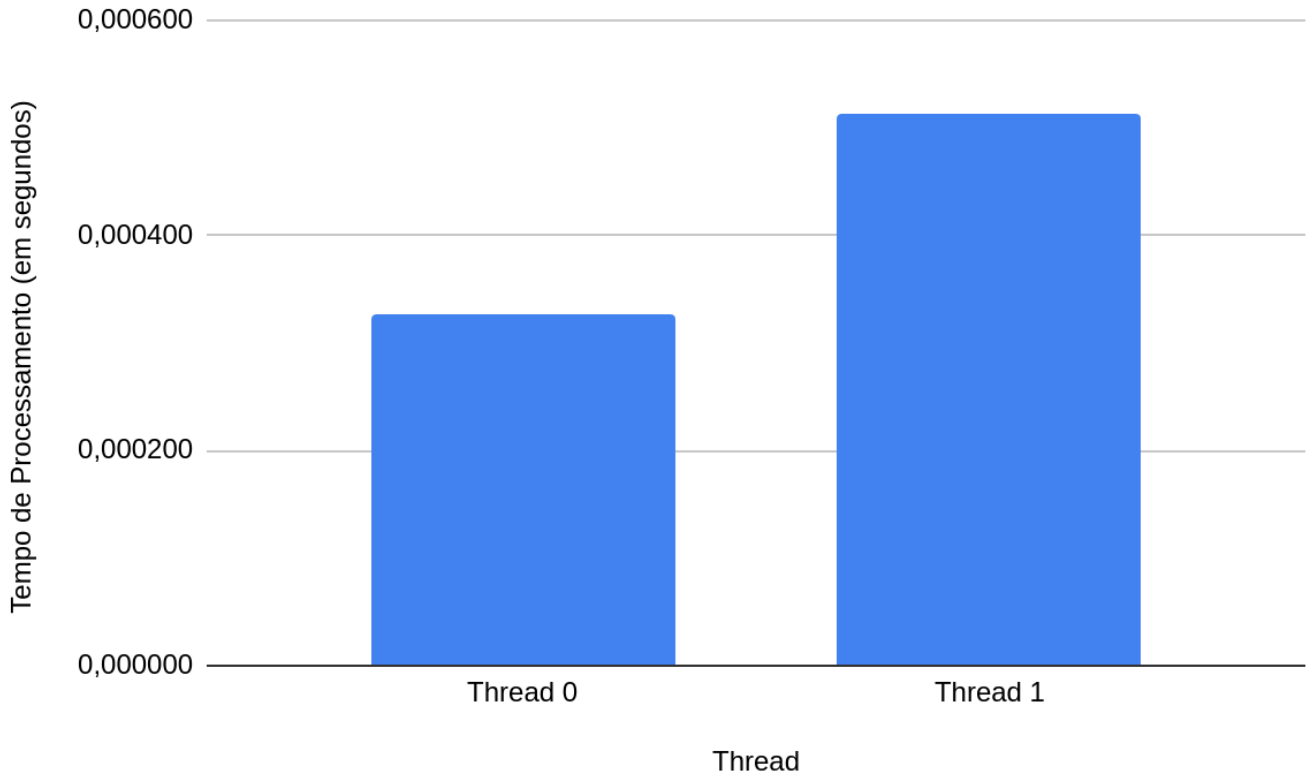
Teste com 1 thread:

```
Unset
Entrada: ./mergesort 1 arq1.dat arq2.dat arq3.dat arq4.dat arq5.dat -o saida.dat

Saída:
      Tempo de execução do thread 0: 0.000663 segundos.
      Tempo total de execução: 0.000929 segundos.
```



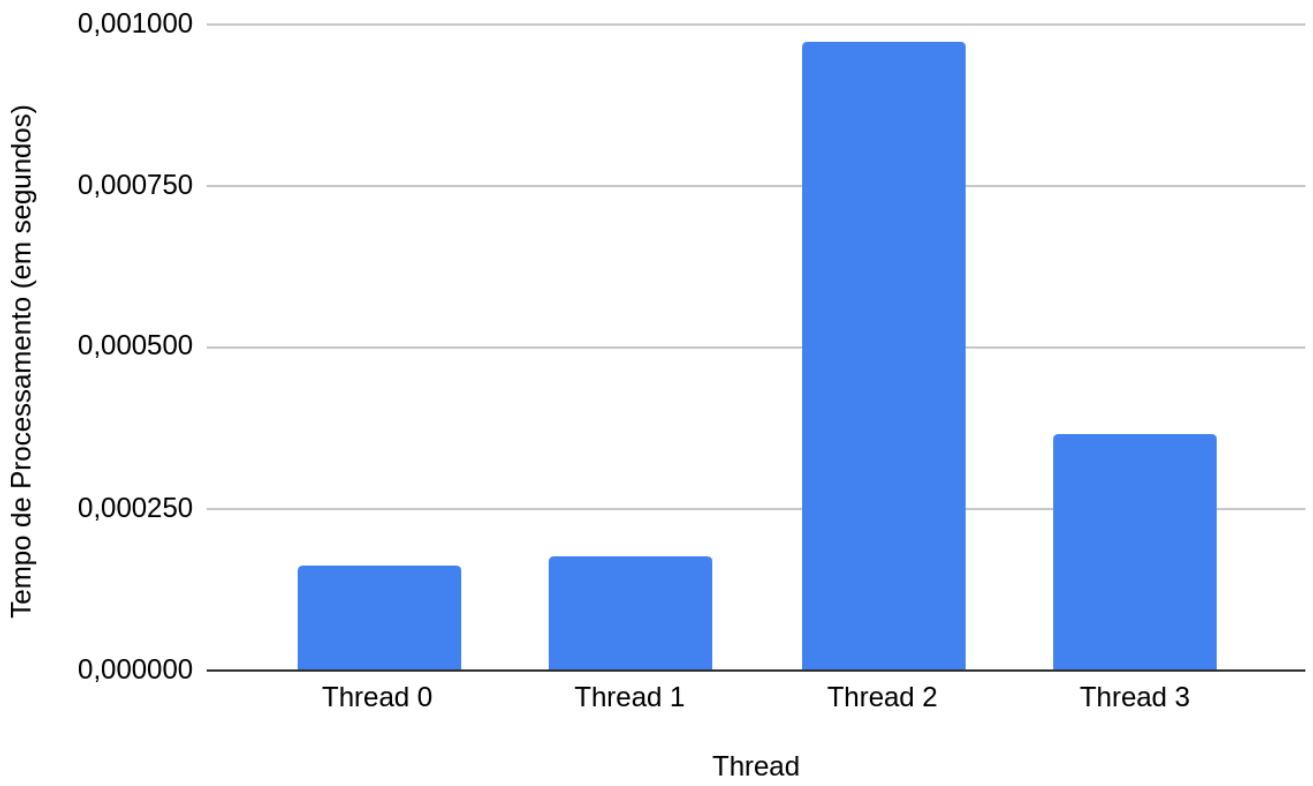
Teste com 2 threads:

```
Unset
Entrada: ./mergesort 2 arq1.dat arq2.dat arq3.dat arq4.dat arq5.dat -o saida.dat

Saída:
	Tempo de execução do thread 0: 0.000327 segundos.
	Tempo de execução do thread 1: 0.000513 segundos.
	Tempo total de execução: 0.000647 segundos.
```



Teste com 4 threads:

```
Unset
Entrada: ./mergesort 4 arq1.dat arq2.dat arq3.dat arq4.dat arq5.dat -o saida.dat

Saída:
      Tempo de execução do thread 0: 0.000162 segundos.
      Tempo de execução do thread 1: 0.000178 segundos.
      Tempo de execução do thread 2: 0.000974 segundos.
      Tempo de execução do thread 3: 0.000367 segundos.
      Tempo total de execução: 0.001157 segundos.
```



Teste com 8 *threads*

```
Unset
Entrada: ./mergesort 8 arq1.dat arq2.dat arq3.dat arq4.dat arq5.dat -o saida.dat

Saída:
      Tempo de execução do thread 0: 0.000150 segundos.
      Tempo de execução do thread 1: 0.000215 segundos.
      Tempo de execução do thread 2: 0.000161 segundos.
      Tempo de execução do thread 3: 0.000064 segundos.
      Tempo de execução do thread 4: 0.000058 segundos.
      Tempo de execução do thread 5: 0.000047 segundos.
      Tempo de execução do thread 6: 0.000046 segundos.
      Tempo de execução do thread 7: 0.000261 segundos.
      Tempo total de execução: 0.000928 segundos.
```

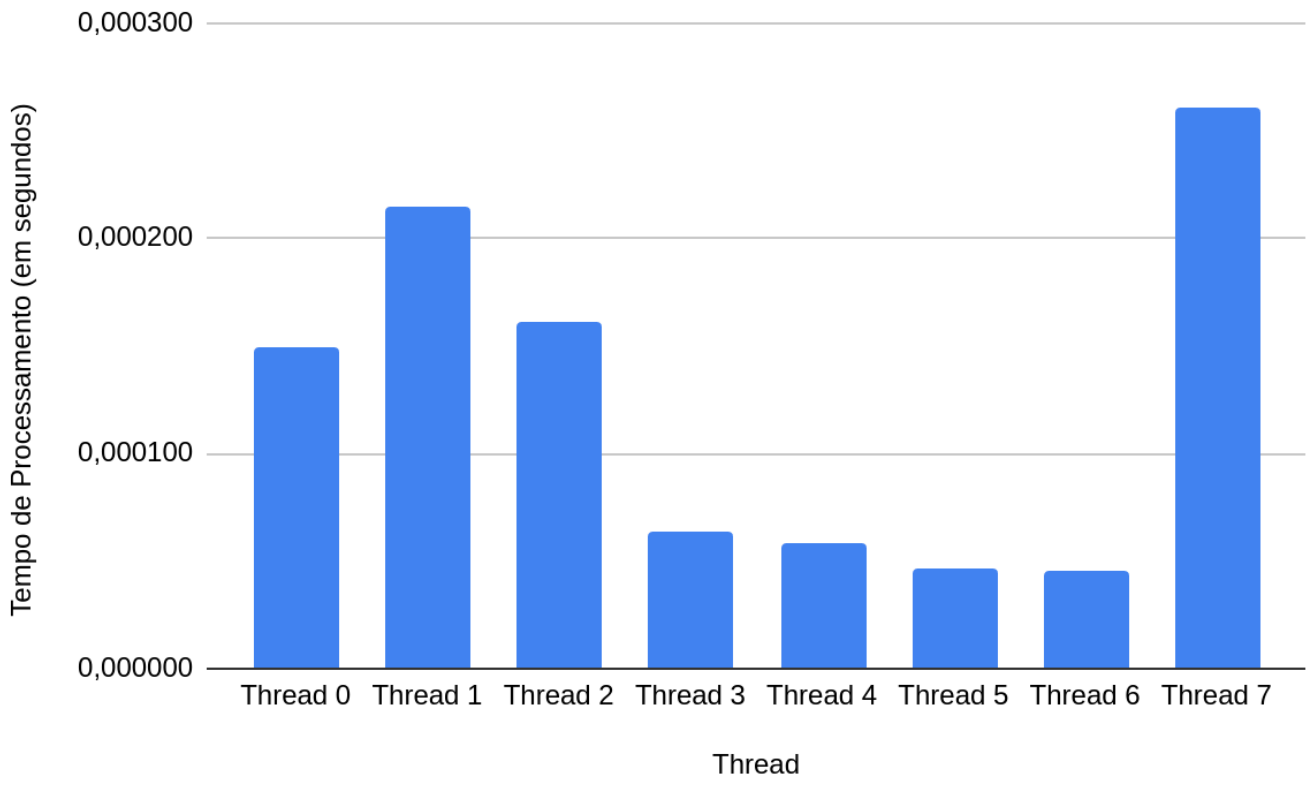
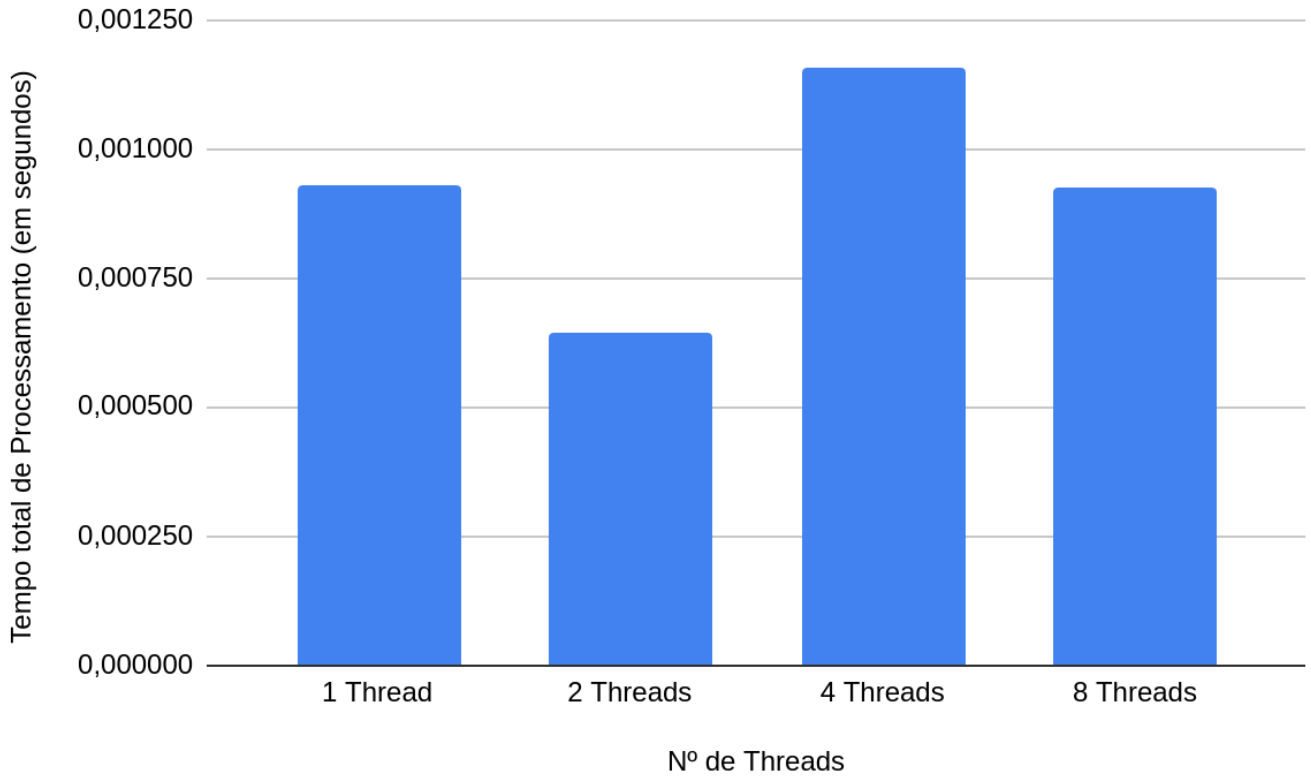


Gráfico de comparação entre os tempos totais de ordenação



5. CONCLUSÃO

Ao analisar o tempo de execução de cada *thread*, observa-se uma distribuição mais eficiente à medida em que o número de *threads* é aumentado nos testes. Com o acréscimo de threads, o sistema passa a gerenciar as tarefas de forma mais paralela, o que geralmente melhora a performance até um certo limite. Além disso, ao examinar os gráficos de distribuição do tempo por *thread*, nota-se que a última *thread* frequentemente leva mais tempo para completar suas tarefas. Isso ocorre devido à sua responsabilidade de fazer a ordenação final, além da sua parte.

Em outra análise, ao observar o tempo total de execução em função do número de *threads*, percebe-se uma redução significativa de tempo ao se passar de 1 para 2 *threads*, evidenciando uma melhora no desempenho ao se explorar o processamento paralelo. No entanto, a partir de 4 *threads*, o tempo de execução total começa a aumentar em relação ao teste de 2 *threads*. Esse efeito se deve à sobrecarga gerada pelo gerenciamento de múltiplas *threads*, incluindo a sincronização entre elas e a limitação de recursos do sistema (como núcleos de processamento e memória disponíveis), o que leva a um efeito inverso ao esperado, aumentando o tempo global de processamento em vez de reduzi-lo. Em suma, usar 2 *threads* foi mais eficiente.