UNIVERSIDADE FEDERAL DE LAVRAS

Departamento de Ciência da Computação

GAC 105 - Programação Paralela e Concorrente

Projeto Prático

Alunos:

- Lucas Neves Sáber Gabriel - 202020459
- Otávio Augusto Trindade Fonseca - 202020551
- Thiago Odilon de Almeida - 202021025

Paralelização do MergeSort

O problema que escolhemos é a ordenação de um grande volume de dados contidos em um arquivo. Esse arquivo contém informações detalhadas sobre filmes, sendo cada linha representando um filme e contendo diversos atributos, tais como título, avaliação, gênero, ano de lançamento, pontuação, número de votos, diretor, escritor, elenco, país de origem, orçamento, arrecadação bruta, produtora, entre outros. A quantidade de dados pode ser substancial, tornando a tarefa de ordenação uma operação que demanda um alto grau de eficiência e performance. O objetivo é implementar uma solução paralela que seja capaz de ordenar eficientemente esses dados, aproveitando as vantagens da paralelização para melhorar o desempenho do algoritmo.

Com isso, o algoritmo selecionado para abordar esse problema é o Merge Sort. Trata-se de um algoritmo de ordenação baseado no princípio "dividir para conquistar", que se destaca por sua eficiência e estabilidade. O MergeSort tem uma complexidade média de tempo O(n log n), o que o torna uma escolha apropriada para a ordenação de grandes conjuntos de dados. O objetivo é implementar uma solução paralela que seja capaz de ordenar eficientemente esses dados, aproveitando as vantagens da paralelização para melhorar o desempenho do algoritmo.

Descrição do Algoritmo Merge Sort:

- Divisão: O algoritmo inicia dividindo a lista não ordenada em duas sublistas de tamanhos aproximadamente iguais.
- Conquista: Cada sub-lista é ordenada de forma recursiva usando o Merge Sort, realizando a divisão novamente até que as sublistas contenham apenas um elemento.
- Combinação: As sub listas ordenadas são mescladas para obter a lista final ordenada. Nesse processo, o algoritmo compara os elementos de ambas as sub listas, realizando a fusão de maneira ordenada.

A estabilidade do Merge Sort é outro atributo relevante, pois garante que elementos com chaves iguais permaneçam em suas posições relativas originais. Além disso, o algoritmo não exige que todos os elementos estejam disponíveis na memória principal o tempo todo, tornando-o uma opção viável para a ordenação de conjuntos de dados que não cabem integralmente na RAM. Porém apesar dessas vantagens, o Merge Sort apresenta uma desvantagem em relação ao espaço adicional necessário para criar as sublistas temporárias durante o processo de mesclagem. No entanto, seu desempenho geral

compensa essa limitação, e quando aplicada a técnicas de paralelização, como será discutido neste trabalho, o Merge Sort pode se tornar ainda mais eficiente e eficaz para a ordenação de dados em larga escala.

A decisão tomada pelo grupo foi ao implementar o programa paralelo para o Merge Sort, optamos por utilizar a biblioteca de paralelização "pthread", que permite a criação e gerenciamento de threads. Essa escolha foi feita devido à simplicidade de uso e à familiaridade e pelo fato de ter sido ensinado na disciplina.

Outra decisão importante foi a definição do número de threads a serem utilizadas na execução paralela do algoritmo. Optamos por variar o número de threads de 1 a 100, a fim de analisar o impacto da paralelização em diferentes cenários.

O programa paralelo foi implementado em linguagem de programação C++ e utiliza a biblioteca "pthread" para criar e gerenciar as threads. A função principal do programa chama a função "funcao_pthread()", que é responsável por executar o Merge Sort em paralelo.

A função "funcao_pthread()" divide a lista de filmes em partes iguais e atribui cada parte a uma thread para realizar a ordenação paralela. Em seguida, as sub listas ordenadas são mescladas usando uma função "merge_parte_vetor()" para obter a lista final ordenada. Durante o processo de mesclagem, o programa utiliza a função "merge()" para combinar duas metades ordenadas do vetor.

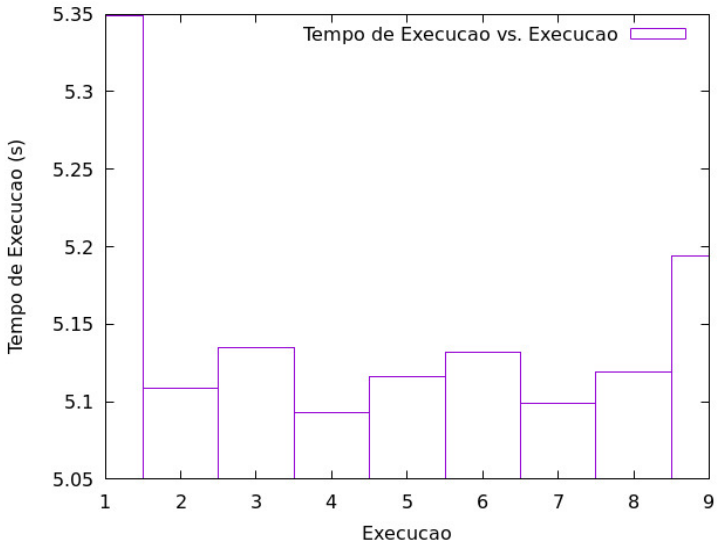
O programa foi testado com diferentes tamanhos de entrada, variando de conjuntos pequenos a grandes volumes de dados, para avaliar o desempenho em cenários diversos.

Ao testar a implementação, obtemos alguns resultados que indicam que a paralelização do Merge Sort trouxe melhorias significativas no tempo de execução à medida que o número de threads aumentou, alcançando um speedup notável. No entanto, foram observados casos em que o overhead paralelo tornou-se mais significativo para conjuntos de dados menores e com menos threads.

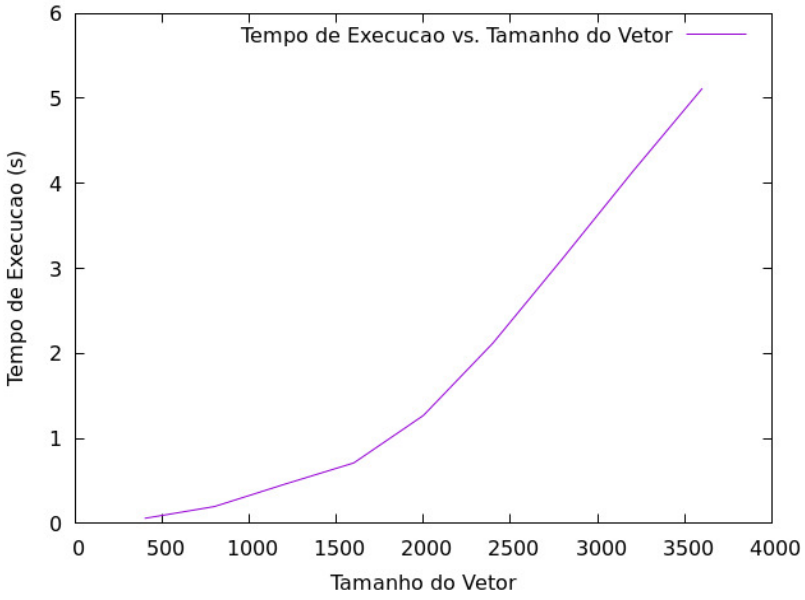
E com essa análise dos resultados revelou-se que o desempenho da ordenação paralela é fortemente influenciado pelo tamanho da entrada e pela quantidade de threads utilizados. E em casos de cenários com um grande volume de dados, a paralelização mostrou-se altamente vantajosa, proporcionando uma redução significativa no tempo de execução. Porém, em entradas menores, o custo da criação e gerenciamento das threads pode superar os benefícios da paralelização.

O hardware utilizado foi variado, sendo que cada membro do grupo utilizou do seu próprio computador para fazer os testes, desde um i5-7500 e um Apple M2, em que o desempenho se mostrou incrivelmente melhor nas duas opções.

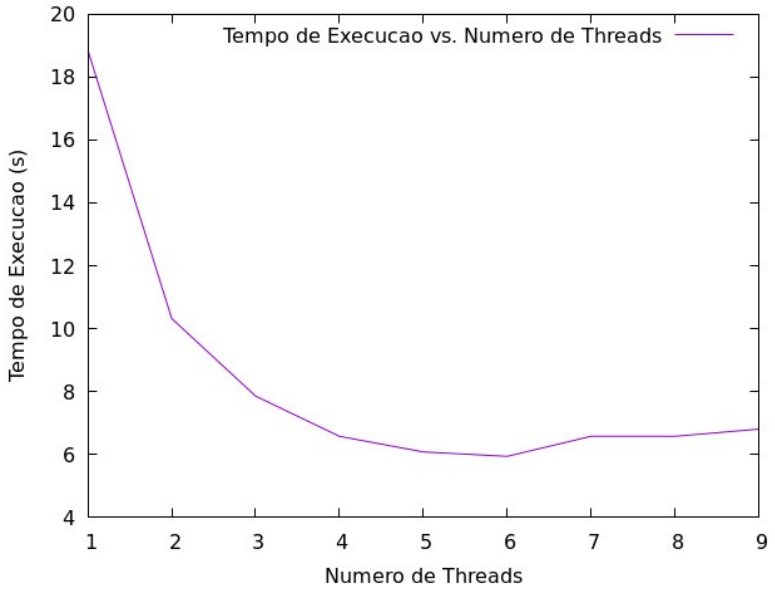
Agora, na geração das tabelas, primeiro testamos 10 execuções no nosso vetor de filmes com 4000 elementos no vetor e obtivemos os seguintes resultados:



Em que como esperávamos, obtivemos em média dados similares pois se trata do mesmo vetor e a mesma quantidade de threads. Agora, fizemos uma função para que a ordenação era chamada cada hora com um tamanho diferente do vetor, aumentando de 500 em 500 utilizando ainda 100 threads para a ordenação:



Como observado, o tempo foi aumentando com a quantidade de dados também sendo aumentada, o que também nos retorna um resultado previsto. Após isso, fizemos com que o número de threads se tornasse gradativo, começando com 1 thread e aumentando gradativamente até 10, com o tamanho do vetor fixo em 4000. Após a execução o seguinte gráfico foi gerado:



Número de Threads	Tempo de Execução
1	18.8477
2	10.3226
3	7.85723
4	6.57869
5	6.08313
6	5.94298
7	6.58215
8	6.5801
9	6.80963

Nesse gráfico podemos observar que obtemos o melhor desempenho em relação ao tempo de execução foi utilizando 6 threads no caso de um vetor com 4000 dados. Existem várias razões pelas quais o nosso resultado pode ter obtido o melhor resultado com 6 threads, dentre elas:

- Arquitetura do hardware: o número ideal de threads pode estar relacionado à arquitetura do hardware em que o nosso programa está sendo executado. Diferentes CPUs e sistemas têm números ótimos de threads para diferentes tipos de tarefas. O número de núcleos da CPU e a disponibilidade de recursos podem influenciar o desempenho.
- Overhead de criação de threads: A criação e o gerenciamento de threads têm um certo overhead, especialmente se o número de threads for muito grande. Se o overhead de criação e gerenciamento de threads for

significativo em comparação com o tempo de execução real do algoritmo, o desempenho pode piorar com um número maior de threads.

- Overhead de criação de threads: A criação e o gerenciamento de threads têm um certo overhead, especialmente se o número de threads for muito grande. Se o overhead de criação e gerenciamento de threads for significativo em comparação com o tempo de execução real do algoritmo, o desempenho pode piorar com um número maior de threads.
- Fatores não controlados: Há também outros fatores que podem afetar o desempenho, como a aleatoriedade inerente ao sistema operacional, escalonamento do sistema, entre outros.

Com isso concluímos que é importante realizar experimentos com diferentes configurações de threads e tamanhos de vetor para determinar o número ótimo de threads para o seu cenário específico. Fatores como o tamanho do vetor, a arquitetura do hardware e a implementação da paralelização podem afetar o desempenho do algoritmo.

Utilizando dos dados para o cálculo do speedup obtemos:

2 threads = 18.8477 / 10.3226 ≈ 1.825
3 threads = 18.8477 / 7.85723 ≈ 2.397
4 threads = 18.8477 / 6.57869 ≈ 2.862
5 threads = 18.8477 / 6.08313 ≈ 3.096
6 threads = 18.8477 / 5.94298 ≈ 3.171
7 threads = 18.8477 / 6.58215 ≈ 2.860
8 threads = 18.8477 / 6.5801 ≈ 2.860
9 threads = 18.8477 / 6.80963 ≈ 2.763

E para o cálculo da eficiência temos:

2 threads = 1.825 / 2 ≈ 0.9125
3 threads = 2.397 / 3 ≈ 0.799
4 threads = 2.862 / 4 ≈ 0.7155
5 threads = 3.096 / 5 ≈ 0.6192
6 threads = 3.171 / 6 ≈ 0.5285
7 threads = 2.831 / 7 ≈ 0.4044
8 threads = 2.860 / 8 ≈ 0.3575
9 threads = 2.763 / 9 ≈ 0.307

Nesse caso a eficiência para 2 threads é de aproximadamente 0.9125, o que significa que a utilização eficiente das duas threads não é completa, pois a eficiência ideal é 1, o que significa que estamos aproveitando completamente os recursos e que cada thread está contribuindo para o desempenho geral. Como o caso de 2 threads isso pode acontecer devido a diversos fatores, como possíveis gargalos de comunicação ou problemas de balanceamento.

Para o cálculo de fração temos:
2 threads = (1.825 - 1) / (2 - 1) ≈ 0.825
3 threads = (2.397 - 1) / (3 - 1) ≈ 0.6985
4 threads = (2.862 - 1) / (4 - 1) ≈ 0.954

5 threads = (3.096 - 1) / (5 - 1) ≈ 0.774
6 threads = (3.171 - 1) / (6 - 1) ≈ 0.6342
7 threads = (2.831 - 1) / (7 - 1) ≈ 0.4785
8 threads = (2.860 - 1) / (8 - 1) ≈ 0.51
9 threads = (2.763 - 1) / (9 - 1) ≈ 0.345

E para a métrica de Karp-Flat de 6 threads:

T1 = 18.8477 (tempo sequencial)

Tp = 18.8477 / 6 = 3.14128333333 (tempo paralelo com 6 threads)

N = 6 (número de threads) Karp-Flatt = 18.8477 / (3.14128333333 * 6) ≈ 0.9999

Indicando que temos um bom grau de paralelismo sendo explorado.

Em suma, o projeto nos proporcionou uma oportunidade valiosa para aprofundar nosso conhecimento em programação paralela e explorar os desafios e benefícios da paralelização de algoritmos em larga escala. O trabalho em equipe foi fundamental para a elaboração e execução do projeto, permitindo que compartilhássemos conhecimentos, ideias e soluções para alcançarmos os resultados obtidos. Adquirimos uma compreensão mais sólida sobre a importância da otimização de algoritmos e da escolha adequada de técnicas de paralelização para melhorar o desempenho de aplicações que lidam com grandes conjuntos de dados.

Por fim, a experiência adquirida neste projeto prático contribui para o nosso desenvolvimento acadêmico e profissional, preparando-nos para enfrentar desafios em áreas de programação e computação paralela e concorrente, bem como para aprimorar nossas habilidades como desenvolvedores e pesquisadores na área de Ciência da Computação.