Universidade Federal
de São João del-Rei

Departamento de Ciência da Computação

Leonardo de Oliveira Tedeschi
Geovanna Vittória de Jesus Assis

Computação Paralela: Trabalho prático 3

São João del-Rei, janeiro de 2025

Sumário

1 Introdução **3**
1.1 Objetivos 3

2 Listagem das Rotinas e Execução **3**

3 Análises e Resultados **4**
3.1 Testes Realizados 4
3.1.1 Teste 1: Tempo de Execução vs Número de Processos MPI 4
3.1.2 Teste 2: Tempo de Execução vs Número de Threads OpenMP 5
3.2 Resultados Obtidos 6
3.2.1 Tempo de Execução vs Número de Processos MPI 6

4 Conclusões **6**

1 Introdução

A computação paralela tem sido amplamente utilizada para acelerar a execução de algoritmos complexos. Este trabalho tem como objetivo implementar versões paralelas do cálculo de vizinhos comuns em grafos, explorando tanto a **memória compartilhada** quanto a **memória distribuída**, utilizando as APIs OpenMP e MPI. Além da implementação, busca-se avaliar o desempenho dessas abordagens em diferentes configurações de processamento.

1.1 Objetivos

O principal objetivo deste trabalho é otimizar o cálculo de vizinhos comuns em grafos por meio da paralelização, considerando arquiteturas de memória compartilhada e distribuída. Para isso, foram definidos os seguintes objetivos específicos:

- **Implementação do Algoritmo de Vizinhos Comuns**: Desenvolver um algoritmo eficiente para identificar os vizinhos comuns entre pares de vértices em um grafo representado por uma lista de arestas, inicialmente em uma versão sequencial.
- **Paralelização com MPI e OpenMP**: Implementar versões paralelizadas do algoritmo, explorando a memória distribuída com MPI e a memória compartilhada com OpenMP, analisando os desafios e benefícios de cada abordagem.
- **Avaliação do Desempenho por Número de Processos e Threads**: Medir o impacto da variação do número de processos MPI e threads OpenMP no desempenho do algoritmo, considerando diferentes configurações de execução.
- **Análise Comparativa das Abordagens**: Comparar o tempo de execução das versões sequencial, MPI e OpenMP para diferentes tamanhos de grafos, identificando os cenários em que cada técnica apresenta melhor desempenho.
- **Estudo da Escalabilidade**: Avaliar a escalabilidade das soluções paralelas, verificando como o tempo de execução varia conforme o aumento do número de processos, threads e do tamanho do grafo.
- **Documentação e Visualização dos Resultados**: Gerar gráficos e relatórios detalhados para uma análise clara da eficiência de cada implementação, destacando ganhos de desempenho e eventuais limitações.

2 Listagem das Rotinas e Execução

A implementação do algoritmo foi estruturada em diversas funções, cada uma responsável por uma etapa específica do processamento do grafo. As principais funções desenvolvidas incluem:

- `ler_e_distribuir_dados()`: Responsável pela leitura do arquivo de entrada no formato edgelist e pela distribuição dos dados entre os processos MPI. O processo de rank 0 carrega os dados do arquivo, determina o número de vértices e arestas, e transmite essas informações aos demais processos por meio de `MPI_Bcast()`.
- `construir_matriz_adjacencia()`: Constrói a matriz de adjacência do grafo a partir das arestas lidas. A estruturação dessa matriz é paralelizada com OpenMP para otimizar o tempo de construção.
- `calcular_vizinhos_comuns()`: Para cada par de vértices, calcula o número de vizinhos em comum. A contagem é realizada de forma vetorizada com `#pragma omp simd` e distribuída dinamicamente entre as threads usando `#pragma omp parallel for schedule(dynamic, 10)`.
- `gerenciar_buffers_paralelos()`: Cada thread mantém um buffer de saída para armazenar os resultados de forma eficiente, evitando concorrência na escrita. Quando o buffer atinge sua capacidade, os dados são gravados temporariamente em um arquivo intermediário.

- `mpi_main()`: Função principal da versão paralela utilizando MPI. Inicializa o ambiente MPI, distribui a carga de trabalho entre os processos, realiza as computações paralelas e, ao final, realiza a coleta e fusão dos resultados.
- `omp_main()`: Implementação alternativa com OpenMP, otimizando o desempenho para arquiteturas de memória compartilhada.

A execução do programa exige a especificação do arquivo de entrada e do número de threads por processo. O comando para execução utilizando MPI e OpenMP é:

```
mpirun -np <num_processos> ./programa <arquivo_entrada> <threads_por_processo>
```

onde:

- `<num_processos>` define o número de processos MPI a serem utilizados;
- `<arquivo_entrada>` é o arquivo contendo a lista de arestas no formato edgelist;
- `<threads_por_processo>` especifica a quantidade de threads a serem utilizadas por cada processo.

O programa gera um arquivo de saída no formato `.cng`, contendo os pares de vértices e a quantidade de vizinhos em comum para cada par. Ao término da execução, estatísticas de desempenho, como tempo total e tempo de computação paralela, são exibidas.

A execução dos programas segue a seguinte estrutura:

```
# Compilar a versão MPI\mpicc -o mpi_vizinhos mpi_grafo.c -lm
# Executar com 4 processos
mpirun -np 4 ./mpi_vizinhos edgelist.txt

# Compilar a versão OpenMP
gcc -o omp_vizinhos -fopenmp omp_grafo.c -lm
# Executar com 8 threads
OMP_NUM_THREADS=8 ./omp_vizinhos edgelist.txt
```

3 Análises e Resultados

3.1 Testes Realizados

Os experimentos foram conduzidos para avaliar o impacto da paralelização no desempenho do algoritmo, variando o número de processos MPI e o número de threads OpenMP. Os testes foram realizados utilizando diferentes tamanhos de grafos e configurações de processamento para analisar a escalabilidade de cada abordagem.

3.1.1 Teste 1: Tempo de Execução vs Número de Processos MPI

Este teste avaliou como o tempo de execução do algoritmo varia em função do número de processos MPI. Observou-se que, à medida que o número de processos aumenta, o tempo de execução reduz, evidenciando a escalabilidade da abordagem distribuída. No entanto, o ganho de desempenho pode ser influenciado pelo tamanho do grafo e pela sobrecarga de comunicação entre os processos.

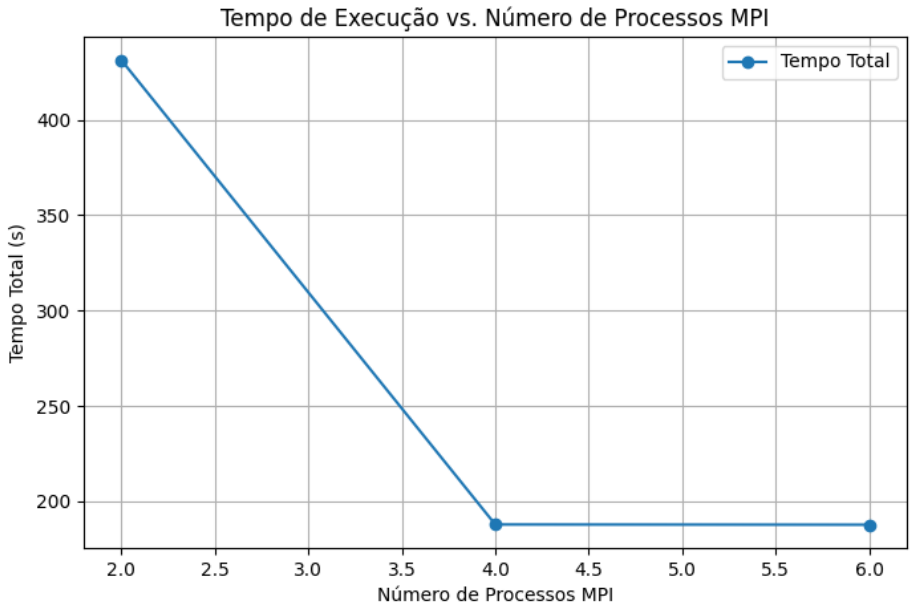

Figura 1: Tempo Total vs Número de Processos MPI

3.1.2 Teste 2: Tempo de Execução vs Número de Threads OpenMP

O objetivo deste teste foi analisar o impacto do número de threads no desempenho da versão OpenMP. Foi observado que o tempo de execução reduz com o aumento das threads até certo limite, após o qual a melhoria se estabiliza ou pode até ser prejudicada pelo overhead de gerenciamento das threads e pela contenção de recursos.

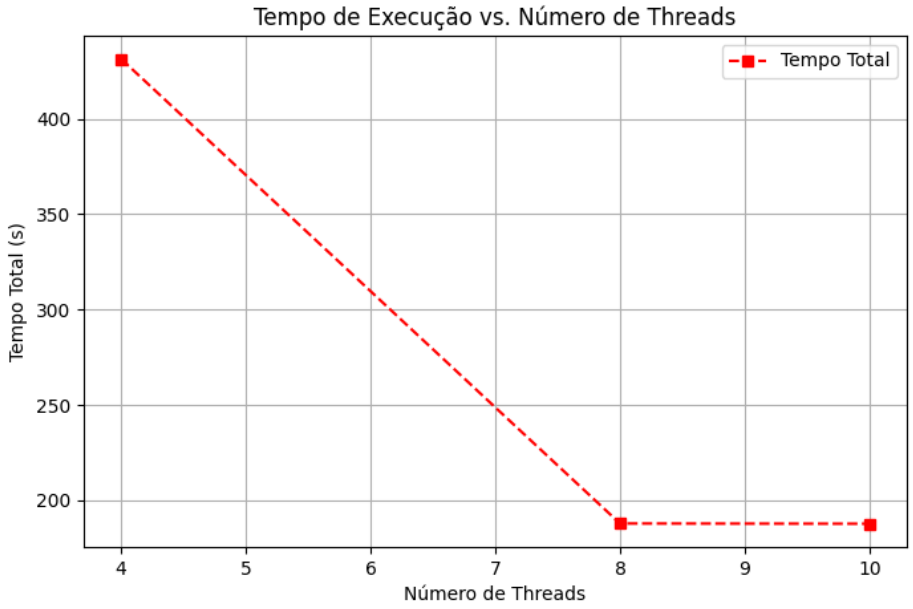

Figura 2: Tempo Total vs Número de Threads OpenMP

3.2 Resultados Obtidos

3.2.1 Tempo de Execução vs Número de Processos MPI

Processos MPI	Threads OpenMP	Tempo (segundos)
2	4	431.41
4	8	187.85
6	10	187.73

Tabela 1: Tempo de Execução em Função do Número de Processos MPI

4 Conclusões

Este trabalho implementou versões paralelas do algoritmo de vizinhos comuns em grafos utilizando as APIs MPI e OpenMP, explorando respectivamente a memória distribuída e a memória compartilhada. A análise dos resultados demonstrou que ambas as abordagens apresentam vantagens específicas, dependendo da configuração de hardware e do número de processos ou threads empregados.

A versão MPI mostrou-se mais escalável para um número elevado de processos, sendo particularmente eficiente em clusters, onde a comunicação entre nós pode ser distribuída de forma equilibrada. No entanto, a sobrecarga de comunicação pode impactar o desempenho em cenários com um número reduzido de processos ou grafos menores.

Por outro lado, a versão OpenMP apresentou uma redução significativa no tempo de execução quando executada em arquiteturas com múltiplos núcleos e memória compartilhada. Entretanto, foi observado que o ganho de desempenho tende a se estabilizar à medida que o número de threads aumenta, devido a fatores como contenção de recursos e overhead de sincronização.

Os testes realizados permitiram avaliar o impacto do número de processos MPI e threads OpenMP no desempenho do algoritmo, evidenciando a importância de escolher a abordagem mais adequada conforme as características do ambiente de execução. Como trabalhos futuros, sugere-se a otimização das implementações paralelas por meio de estratégias como balanceamento de carga e uso de técnicas híbridas que combinem MPI e OpenMP para explorar ao máximo os recursos computacionais disponíveis.