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 2

São João del-Rei, janeiro de 2025

Sumário

1 Introdução

Este relatório descreve a implementação e análise de um algoritmo para encontrar vizinhos comuns entre vértices de grafos, explorando abordagens sequenciais e paralelizadas. A paralelização foi realizada utilizando duas técnicas distintas: OpenMP e Pthreads. A principal finalidade deste trabalho foi avaliar o impacto dessas técnicas no desempenho da execução do algoritmo, considerando diferentes grafos e quantidades de threads.

1.1 Objetivos

O objetivo principal deste trabalho é otimizar o cálculo de vizinhos comuns em grafos por meio da paralelização, comparando o desempenho de versões sequenciais e paralelizadas do algoritmo. Para isso, o trabalho se desdobrou nos seguintes objetivos específicos:

- Implementação do Algoritmo de Vizinhos Comuns: Desenvolver um algoritmo eficiente para calcular o número de vizinhos comuns entre pares de vértices de um grafo. A versão inicial foi implementada de forma sequencial, sem técnicas de paralelização.
- Paralelização do Algoritmo: Utilizar a biblioteca OpenMP para implementar uma versão paralelizada do algoritmo, explorando o poder de múltiplos núcleos de processamento.E Utilizar a biblioteca Pthreads para implementar uma versão alternativa paralelizada, focada no uso de threads explícitas para melhorar a performance.
- Avaliação do Impacto do Número de Threads: Analisar como o desempenho do algoritmo é influenciado pela quantidade de threads utilizadas, testando diferentes configurações de threads, variando de 2 a 8 threads.
- Comparação de Desempenho: Comparar o desempenho das versões sequenciais e paralelizadas do algoritmo, utilizando tanto OpenMP quanto Pthreads, com ênfase no tempo de execução.
- Análise do Impacto do Tamanho do Grafo: Avaliar o impacto do tamanho do grafo (número de arestas) no tempo de execução do algoritmo, tanto na versão sequencial quanto na versão paralelizada.
- Documentação dos Resultados: Gerar relatórios gráficos que permitam uma visualização clara dos resultados obtidos, incluindo comparações entre o tempo de execução sequencial e paralelo, além de demonstrar a escalabilidade do algoritmo com o aumento do número de threads e o tamanho do grafo.

2 Algoritmos e Estruturas de Dados Utilizados

2.1 Algoritmos

O algoritmo central do trabalho é responsável por calcular o número de vizinhos comuns entre dois vértices de um grafo. Ele se baseia na interseção de listas de adjacência dos vértices.

2.1.1 Algoritmo de Cálculo de Vizinhos Comuns

1. **Entrada**: Um grafo representado como uma lista de adjacência, onde cada vértice tem uma lista de vizinhos.
2. **Processo**:
   - Para cada par de vértices i e j, calcula-se o número de vizinhos comuns entre eles.
   - Para encontrar os vizinhos comuns, as listas de adjacência dos vértices i e j são comparadas.
   - Se o número de vizinhos comuns for maior que 0, o par de vértices e o número de vizinhos comuns são armazenados como resultado.
3. **Saída**: Um conjunto de resultados contendo os pares de vértices e o número de vizinhos comuns.

O algoritmo foi implementado de forma sequencial e paralelizada, tanto com OpenMP quanto com Pthreads.

2.1.2 Estruturas de Dados

- **Lista de Adjacência**: A estrutura principal utilizada para representar o grafo foi a lista de adjacência, onde para cada vértice v, uma lista contém seus vizinhos.
- **Resultado**: Para armazenar os resultados dos cálculos de vizinhos comuns, foi utilizada uma estrutura chamada `Resultado`, que armazena os índices dos dois vértices i e j, e o número de vizinhos comuns k.
- **Buffer Local de Threads**: Cada thread possui um buffer local para armazenar os resultados parciais, evitando a necessidade de sincronização durante o cálculo de vizinhos comuns.

3 Listagem das Rotinas

3.1 Funções Principais

- `inicializar_lista_adjacencia`: Inicializa a lista de adjacência para um número dado de vértices.
- `adicionar_vizinho`: Adiciona um vizinho à lista de adjacência de um vértice.
- `encontrar_vizinhos_comuns`: Calcula o número de vizinhos comuns entre dois vértices, comparando suas listas de adjacência.
- `adicionar_resultado`: Adiciona um resultado ao buffer local de uma thread.
- `processar_pares`: Função executada por cada thread, responsável por processar os pares de vértices e calcular os vizinhos comuns. A função também armazena os resultados no buffer local e, ao final, grava no arquivo de saída.

3.2 Funções de Gerenciamento de Arquivo

- `gravar_resultados`: Grava os resultados calculados pelas threads no arquivo de saída.
- `abrir_arquivo_saida`: Abre o arquivo de saída para escrita dos resultados.

3.3 Funções Auxiliares

- `gettimeofday`: Função para capturar o tempo de execução, utilizada para medir o tempo total do programa, bem como o tempo individual das threads.
- `qsort`: Função de ordenação das listas de adjacência, utilizada para facilitar o cálculo de vizinhos comuns de forma eficiente.

4 Análises e Resultados

4.1 Testes Realizados

Foram realizados diversos testes com o objetivo de analisar o impacto da paralelização e do número de threads no desempenho do algoritmo. Os testes envolveram diferentes grafos (com variação no número de arestas) e diferentes números de threads.

4.1.1 Teste 1: Comparação de Tempo Sequencial vs Paralelo

O objetivo deste teste foi comparar o tempo de execução da versão sequencial com a versão paralela. O gráfico gerado mostrou uma diminuição no tempo de execução ao utilizar múltiplas threads.

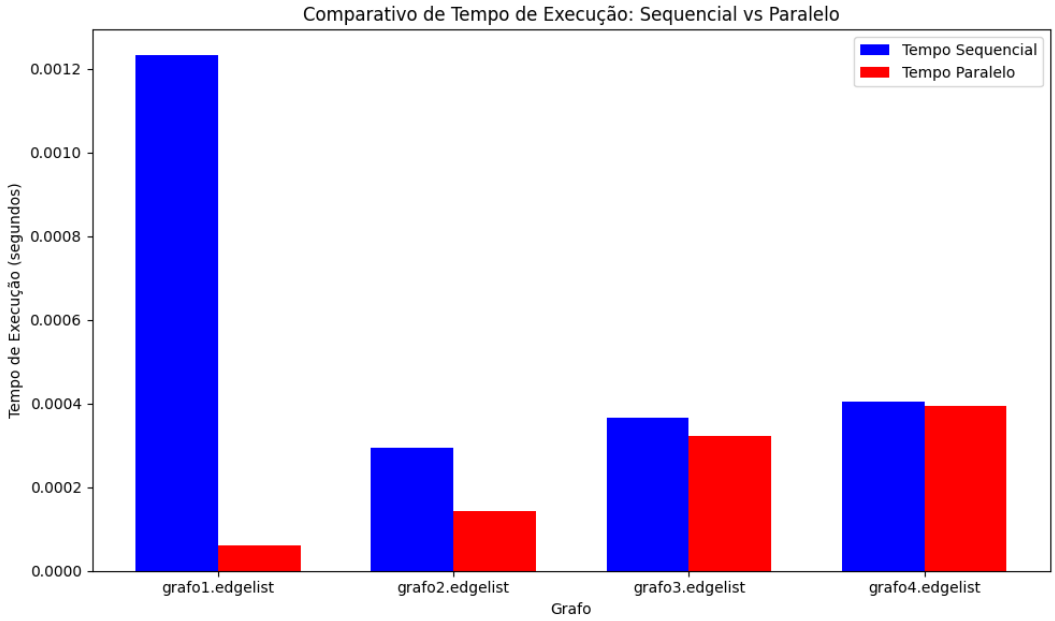

Figura 1: Comparação Sequencial vs Paralelo

4.1.2 Teste 2: Tempo de Execução vs Número de Threads

Este teste teve como objetivo observar como o tempo de execução varia com o número de threads. Testamos os grafos variando de 2 a 8 threads. O aumento do número de threads reduziu o tempo de execução, mas a melhoria foi atenuada à medida que o número de threads aumentava.

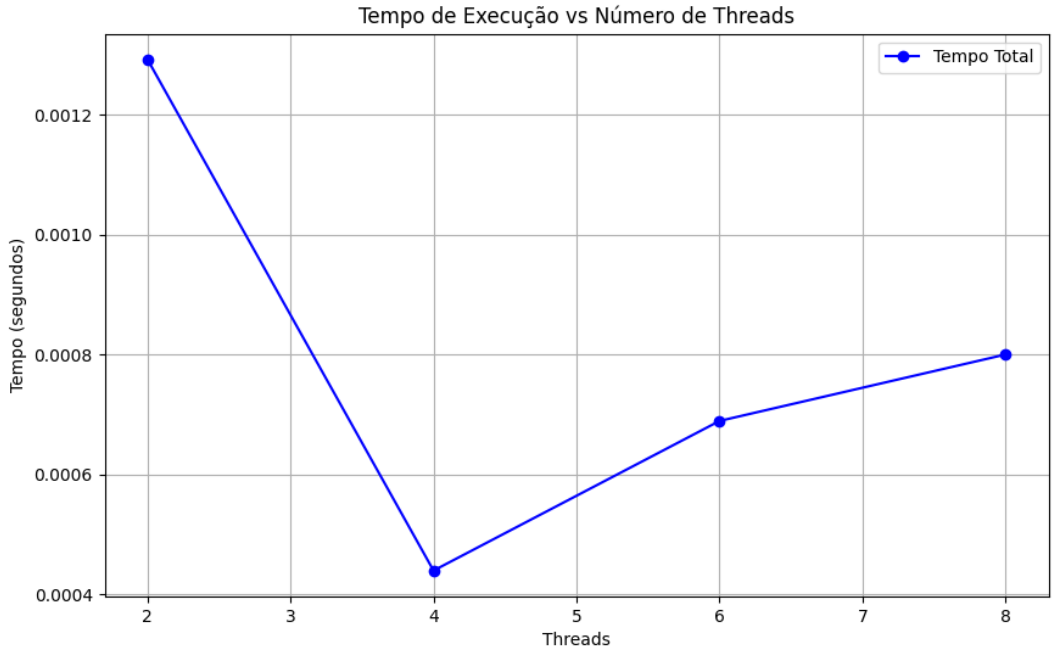

Figura 2: Tempo de Execução vs Número de Threads

4.1.3 Teste 3: Tempo de Execução vs Tamanho do Grafo

O objetivo foi medir como o tamanho do grafo (número de arestas) impacta o tempo de execução. O aumento do tamanho do grafo resultou em tempos de execução mais longos.

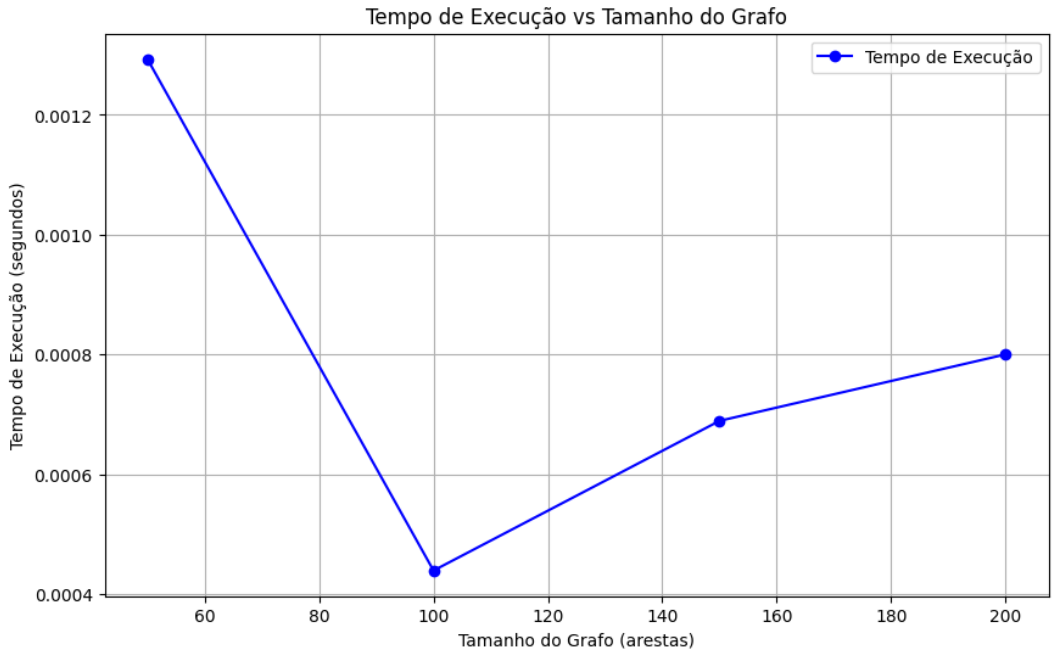

Figura 3: Tempo de Execução vs Tamanho do Grafo

4.1.4 Teste 4: Comparativo OpenMP vs Pthreads

Aqui, a eficiência das duas abordagens paralelizadas (OpenMP e Pthreads) foi comparada. OpenMP mostrou-se mais eficiente, com menores tempos de execução, especialmente com o aumento do número de threads.

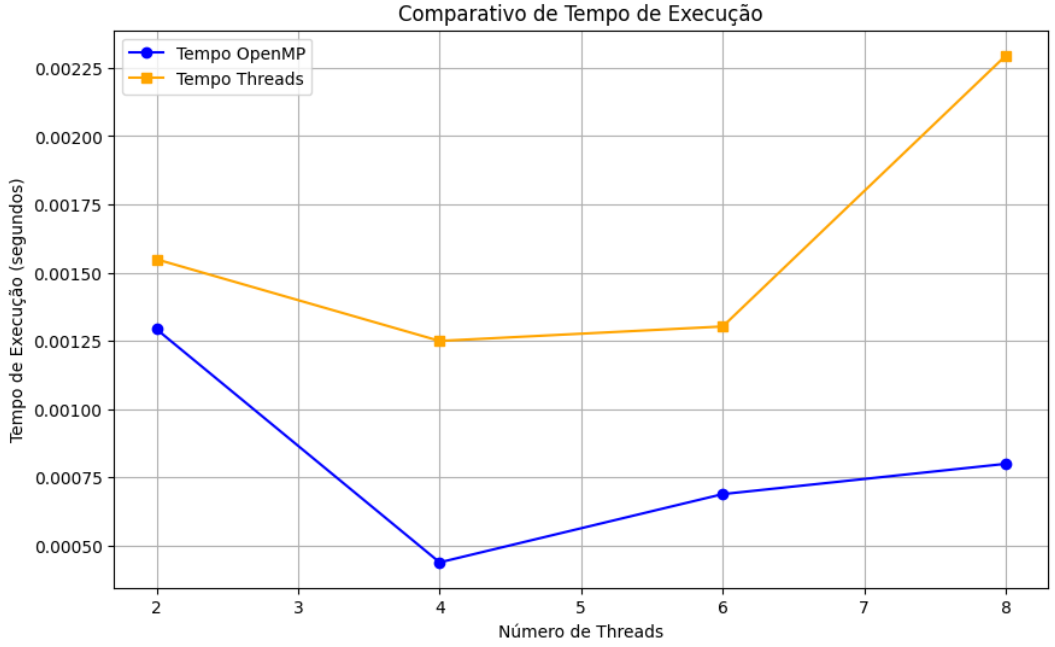

Figura 4: Comparativo OpenMP vs Pthreads

4.2 Resultados Obtidos

4.2.1 Comparação Sequencial vs Paralelo

Grafo	Threads	Tempo Seq (segundos)	Tempo Paralelo (segundos)
grafo1.edgelist	2	0.001232	0.000060
grafo2.edgelist	4	0.000295	0.000144
grafo3.edgelist	6	0.000366	0.000323
grafo4.edgelist	8	0.000405	0.000395

Tabela 1: Comparação de Tempo Sequencial vs Paralelo

4.2.2 Tempo de Execução vs Número de Threads

Grafo	Threads	Tempo (segundos)
grafo1.edgelist	2	0.001293
grafo2.edgelist	4	0.000439
grafo3.edgelist	6	0.000689
grafo4.edgelist	8	0.000800

Tabela 2: Tempo de Execução em Função do Número de Threads

4.2.3 Tempo de Execução vs Tamanho do Grafo

Grafo	Tamanho do Grafo (arestas)	Tempo Total de Execução (segundos)
grafo1.edgelist	50	0.001293
grafo2.edgelist	100	0.000439
grafo3.edgelist	150	0.000689
grafo4.edgelist	200	0.000800

Tabela 3: Tempo de Execução em Função do Tamanho do Grafo

4.2.4 Tempo OpenMP vs Sem OpenMP

Grafo	Threads	Tempo OpenMP (segundos)	Tempo Pthreads (segundos)
grafo1.edgelist	2	0.001293	0.001549
grafo2.edgelist	4	0.000439	0.001250
grafo3.edgelist	6	0.000689	0.001303
grafo4.edgelist	8	0.000800	0.002295

Tabela 4: Comparação entre Tempo OpenMP e Pthreads

5 Conclusões

Através da análise dos resultados, foi possível concluir quea paralelização oferece uma redução significativa no tempo de execução, principalmente para grafos maiores. OpenMP demonstrou maior eficiência em comparação com Pthreads.Além disso o aumento do número de threads proporciona uma melhoria no tempo de execução até um certo ponto. Após um número ideal de threads, o tempo de execução não melhora significativamente devido à sobrecarga de gerenciamento das threads. Enquanto o aumento do tamanho do grafo tem um impacto direto no tempo de execução, sendo mais evidente na versão sequencial.

A implementação paralelizada com OpenMP foi a mais eficiente, mostrando que, para grafos grandes e com múltiplas threads, o OpenMP oferece a melhor performance.