Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Cornélio Procópio

Desenvolvimento Paralelo e Distribuído de um Problema Computacional

Amanda Moura Cavalcante - 2261049
Melina Alves Gonçalves - 2312727
Henrique Galiano de Moraes - 2418266
Gustavo Morais Alves - 2418240
Lucas André Munhoz da Cruz- 2418312
Danilson Matsushita Junior - 2278235

Cornélio Procópio
2025

Introdução – O Problema do Caixeiro Viajante

O **Problema do Caixeiro Viajante** (do inglês *Travelling Salesman Problem – TSP*) é um dos desafios mais clássicos e estudados da ciência da computação e da otimização combinatória. O problema consiste em encontrar o menor caminho possível que um vendedor deve percorrer para visitar um conjunto de cidades exatamente uma vez e retornar à cidade de origem. Esse problema é classificado como **NP-difícil**, ou seja, não se conhece um algoritmo eficiente que o resolva em tempo polinomial para todos os casos.

Devido à sua complexidade, o TSP é um ótimo exemplo para aplicação de técnicas de **paralelismo** e **computação distribuída**, pois sua solução envolve grande número de combinações e processamento intensivo, especialmente à medida que o número de cidades aumenta.

1. Implementação Sequencial

A versão **sequencial** do Problema do Caixeiro Viajante foi implementada pelas alunas **Melina Alves Gonçalves** e **Amanda Moura Cavalcante**, utilizando a linguagem **Java**, com foco em clareza, organização e fidelidade à definição original do problema.

Optamos por resolver o TSP utilizando o método de **força bruta**, ou seja, gerando todas as permutações possíveis de caminhos entre as cidades. Embora esse método tenha complexidade exponencial (O(n!)), ele garante a obtenção da **solução ótima**, sendo ideal para fins de comparação com as versões paralela e distribuída.

I. Estrutura da Solução

- **Matriz de distâncias:** Utilizamos uma matriz `int[][]` para representar o custo de deslocamento entre cada par de cidades. Essa matriz pode ser facilmente adaptada para outros conjuntos de dados.

- **Permutação recursiva:** A geração dos caminhos possíveis foi feita com permutação in-place usando `Collections.swap()`, evitando criação excessiva de listas em memória.

- **Cálculo da distância total:** Para cada permutação gerada, o algoritmo soma as distâncias entre as cidades consecutivas, incluindo o retorno à cidade de origem, para fechar o ciclo.

- **Comparação de resultados:** A menor distância é armazenada em uma variável global, junto com o caminho correspondente, atualizados sempre que uma nova permutação com distância menor é encontrada.

- **Temporização:** Medimos o tempo de execução com `System.currentTimeMillis()` antes e depois da execução do algoritmo, permitindo análise comparativa de desempenho com outras abordagens.

I. Justificativas e Objetivos

Essa implementação serve como **base de referência** para as versões paralela e distribuída do trabalho. Com ela, conseguimos:

- Validar a lógica correta do cálculo de caminhos;

- Observar o impacto do aumento do número de cidades no tempo de execução;

- Identificar limitações da abordagem sequencial frente a problemas maiores, reforçando a necessidade de paralelismo.

Referência do Algoritmo

O algoritmo de permutação utilizado foi inspirado em exemplos disponíveis em fontes educacionais de domínio público e adaptado para a linguagem Java. Uma estrutura similar pode ser encontrada no repositório:
https://www.geeksforgeeks.org/travelling-salesman-problem-using-brute-force-approach/

Implementação Paralela

A versão paralela do Problema do Caixeiro Viajante foi desenvolvida com foco na utilização de múltiplos núcleos de processamento para acelerar a resolução por força bruta. O objetivo principal desta abordagem é reduzir o tempo de execução do algoritmo por meio da execução concorrente de diferentes ramos do problema.

Estrutura da Solução

- **Matriz de distâncias**: Assim como na versão sequencial, foi utilizada uma matriz `int[][]` para representar as distâncias entre as cidades. Essa estrutura permite fácil leitura e manutenção dos dados, além de ser eficiente para acesso direto em tempo constante.

- **Divisão de tarefas com Threads**: A execução paralela foi implementada utilizando a API `ExecutorService`, com um pool fixo de threads baseado no número de núcleos disponíveis no sistema. Cada thread é responsável por explorar todas as permutações que começam com uma determinada segunda cidade, enquanto a primeira cidade (cidade 0) é fixa para todas as rotas.

- **Permutação recursiva in-place**: Para cada tarefa, foi utilizado o mesmo esquema de permutação recursiva aplicado à sub-lista de cidades restantes, garantindo que todas as possíveis rotas sejam analisadas sem sobrecarregar a memória.

- **Cálculo da distância total**: Após cada permutação, é calculada a distância total percorrida pelas cidades no caminho, incluindo o retorno à cidade de origem para fechar o ciclo. Se essa distância for menor que a menor registrada até então, o caminho e a distância são atualizados de forma sincronizada.

- **Sincronização de resultados**: Para garantir consistência e evitar condições de corrida, o método que atualiza o melhor caminho e sua distância é sincronizada, permitindo que apenas uma thread modifique os dados globais por vez.

Implementação Distribuída

A implementação distribuída do Problema do Caixeiro Viajante (TSP) foi desenvolvida pelos alunos Gustavo Morais Alves e Henrique Galiano de Moraes com o objetivo de distribuir a carga computacional de encontrar a rota ótima entre múltiplos processos, que podem estar em máquinas diferentes. Esta abordagem é particularmente eficaz para problemas com alta complexidade, como o TSP, onde a força bruta gera um número exponencial de rotas a serem avaliadas.

Estrutura da Solução

A solução distribuída consiste em um modelo mestre-trabalhador, onde um processo central (o "Mestre") coordena a distribuição das tarefas e a coleta dos resultados, e múltiplos processos (os "Trabalhadores") executam a computação intensiva.

- **Mestre (`MestreTSP.java`):**
    - **Configuração Inicial:** O Mestre é configurado para escutar conexões em uma porta específica (e.g., `65432`). Ele aguarda um número predefinido de trabalhadores se conectarem.
    - **Geração de Cidades:** Define um conjunto fixo de cidades com suas coordenadas (X, Y). Para o exemplo, são utilizadas 8 cidades (A a H).
    - **Geração de Permutações:** O Mestre é responsável por gerar *todas* as permutações possíveis das cidades, excluindo a cidade inicial para evitar redundância e adicionando-a ao início e fim de cada rota após a geração. Esta etapa é crucial, pois é onde a explosão combinatória ocorre, resultando em um grande número de rotas (ex: para 8 cidades, 7!=5040 permutações de cidades internas, resultando em 5040 rotas completas).
    - **Distribuição de Lotes:** Após gerar todas as rotas, o Mestre divide essa lista em lotes menores, distribuindo um lote para cada trabalhador conectado. A divisão é feita de forma que cada trabalhador receba uma parte aproximadamente igual do trabalho total.
    - **Comunicação com Trabalhadores:** Utiliza `ServerSocket` para aceitar conexões e, para cada trabalhador, cria um `ObjectOutputStream` para enviar o lote de rotas e um `ObjectInputStream` para receber o melhor resultado local. A comunicação é assíncrona, com cada interação com o trabalhador sendo executada em uma `Thread` separada.
    - **Coleta e Consolidação de Resultados:** O Mestre aguarda o retorno de todos os trabalhadores. Cada trabalhador envia seu melhor resultado local (a rota mais curta encontrada em seu lote e sua respectiva distância). O Mestre

então compara todos os resultados locais recebidos para determinar a melhor rota global e a menor distância total.
    - **Temporização:** Mede o tempo total desde o início da conexão com os trabalhadores até a consolidação final dos resultados, utilizando `System.currentTimeMillis()` para uma análise de desempenho precisa.
- **Trabalhador (`TrabalhadorTSP.java`):**
    - **Conexão ao Mestre:** Cada Trabalhador tenta se conectar ao Mestre em um endereço e porta pré-definidos (`localhost:65432` no exemplo).
    - **Recepção de Lote:** Após a conexão, o Trabalhador aguarda e recebe um lote de rotas do Mestre através de um `ObjectInputStream`.
    - **Processamento Local:** O Trabalhador executa a lógica principal do TSP para o seu lote de rotas:
        - Para cada rota recebida, calcula a distância total percorrida somando as distâncias euclidianas entre cidades consecutivas. A classe `Cidade` contém um método `distanciaPara()` para este cálculo.
        - Mantém o controle da menor distância encontrada em seu lote e da rota correspondente.
    - **Envio de Resultado Local:** Uma vez que todas as rotas em seu lote foram processadas, o Trabalhador encapsula o melhor resultado local (distância mínima e a melhor rota) em um objeto `Resultado` e o envia de volta ao Mestre através de um `ObjectOutputStream`.
    - **Encapsulamento de Resultados (`Resultado.java`):** Uma classe `Resultado` serializável é utilizada para empacotar a distância mínima e a melhor rota encontrada por cada trabalhador, facilitando a transmissão entre o Mestre e os Trabalhadores.