Instituto Federal do Norte de Minas Gerais - Campus Montes Claros
Ciência da Computação
Arquitetura de Computadores II

Travelling Salesman Problem Parallel

João Victor Pereira de Araújo
Roberto Ramos Ferreira
Thiago Evangelista dos Santos
Professor: Caribe Zampirolli de Souza

Montes Claros
Setembro de 2024

Sumário

1 Resumo

O Problema do Caixeiro Viajante (TSP) é um dos desafios clássicos de otimização combinatória, que envolve encontrar o caminho mais curto que percorre todas as cidades de um conjunto dado. Este artigo propõe uma solução paralelizada para o TSP utilizando a API OpenMP, explorando múltiplos pontos de partida para identificar a melhor rota. A estratégia de paralelização envolve o cálculo independente das distâncias entre as cidades e a execução simultânea do algoritmo TSP para diferentes pontos de partida, distribuindo a carga computacional entre diversas threads.

Palavras- Chave: Paralelismo, multithreading, TSP, Computação

2 Introdução

O Problema do Caixeiro Viajante (TSP) é um dos mais estudados no campo da otimização combinatória, com aplicações práticas em logística, roteamento de veículos, design de circuitos, entre outros. O problema consiste em encontrar o caminho mais curto que permita a um caixeiro visitar um conjunto de cidades, passando por cada uma exatamente uma vez, e retornando à cidade de origem. A dificuldade do TSP está em sua complexidade exponencial, pois o número de possíveis soluções cresce drasticamente com o aumento do número de cidades. Como consequência, resolver o TSP de forma eficiente torna-se desafiador, especialmente para grandes conjuntos de dados.

Diversas abordagens foram propostas para enfrentar o TSP, incluindo heurísticas e algoritmos exatos. Contudo, o tempo de execução desses algoritmos para grandes instâncias do problema ainda é um desafio. Uma solução promissora é o uso de paralelismo, que permite a execução simultânea de múltiplas tarefas em diferentes núcleos de processamento, reduzindo significativamente o tempo total de execução. O uso de ferramentas como OpenMP facilita essa paralelização, proporcionando uma abordagem eficiente para problemas com grande demanda computacional.

Este artigo propõe uma solução paralela para o TSP utilizando a API OpenMP, visando explorar o paralelismo tanto no cálculo das distâncias entre as cidades quanto na execução do algoritmo TSP para múltiplos pontos de partida. A estratégia consiste em distribuir a carga de trabalho entre múltiplas threads, permitindo a execução simultânea de diferentes partes do algoritmo. Os resultados obtidos demonstram a eficiência do método proposto, especialmente ao lidar com grandes conjuntos de dados.

3 Conceitos Básicos de Paralelismo

Paralelismo é uma técnica de computação que permite a execução simultânea de múltiplas tarefas, aproveitando o poder de processamento de múltiplos núcleos de CPU ou processadores. O paralelismo de dados ocorre quando um problema é dividido em partes menores que podem ser processadas independentemente. Já o paralelismo de tarefas distribui diferentes tarefas para serem executadas em paralelo.

No contexto de computação, uma ferramenta comum para paralelização é o OpenMP (Open Multi-Processing), uma API que facilita a criação de programas paralelos para arquiteturas de memória compartilhada. Ele fornece diretivas para o compilador que indicam que determinadas regiões do código devem ser paralelizadas, otimizando a utilização de múltiplos núcleos de CPU.

Vantagens do paralelismo incluem maior eficiência, menor tempo de execução e melhor utilização de recursos. No entanto, ele também traz desafios como condições de corrida e sincronização entre threads, além de aumento na complexidade de implementação.

4 Descrição do Problema

O Problema do Caixeiro Viajante (TSP) é um problema clássico de otimização combinatória. Dado um conjunto de cidades e as distâncias entre elas, o objetivo é encontrar o caminho mais curto que passa por todas as cidades exatamente uma vez, retornando à cidade de origem.

Contagem de pontos	Contagem de maneiras possíveis
4	24
8	40 320
12	479 001 600
16	20 922 789 888 000
20	2 432 902 008 176 640 000
25	15 511 210 043 330 985 984 000 000

Figura 1: Contagem de permutações para resolver TSP

No contexto deste trabalho, o problema é modificado para permitir múltiplos pontos de partida, com o objetivo de encontrar a melhor rota começando a partir

de cada ponto inicial. O desafio computacional é a alta complexidade do problema, que cresce exponencialmente com o número de cidades, tornando a solução inviável para grandes conjuntos de dados sem o uso de técnicas de otimização, como a paralelização.

Tomaremos como base um exemplo de uma empresa de vans escolares, essa empresa possui um numero X de carros que estão dispostos em diferentes garagens pela cidade, um grupo de pais procurou essa empresa para levar seus filhos para uma escola Y, porem cada criança mora em diferentes bairros, e com esse algoritmo de TSP o dono dessa empresa vai conseguir saber qual é a melhor local de partida do carro para que possa passar em todos os bairros da maneira mais otimizada.

5 Metodologia

5.1 Descrição da Solução

O algoritmo base utilizado para resolver o TSP é uma versão recursiva do método de força bruta, que tenta todas as permutações possíveis de rotas para identificar a de menor custo. Para otimizar o tempo de execução, adotamos uma abordagem paralela, onde o problema é dividido em duas partes principais:

- Cálculo das distâncias entre as cidades: Como as distâncias entre os pares de cidades são independentes, esta etapa é ideal para paralelização.
- Busca do menor caminho para diferentes pontos de partida: Cada ponto inicial do TSP é tratado como uma tarefa independente, permitindo que o algoritmo explore vários caminhos simultaneamente.

A paralelização foi implementada utilizando OpenMP, aproveitando a capacidade de múltiplos threads para realizar as operações de forma simultânea. Isso acelera o cálculo e a busca pelo menor caminho, especialmente em grandes instâncias do problema.

5.2 Implementação

A implementação foi realizada em C++, utilizando OpenMP para explorar o paralelismo. O código começa com a leitura dos dados de entrada, que incluem as coordenadas das cidades. Em seguida, as distâncias entre todas as cidades são calculadas de forma paralela, utilizando a função omp parallel for. O cálculo das distâncias é seguido pela execução do algoritmo TSP para cada ponto inicial.

A estratégia de paralelização envolve a utilização de múltiplos threads para:

- Cálculo das distâncias: Cada thread é responsável por calcular um subconjunto das distâncias entre as cidades, dividindo a matriz de distâncias entre os núcleos disponíveis.
- Execução do TSP: Cada thread executa o TSP para um ponto de partida diferente, explorando o espaço de busca de maneira independente. O uso de seções críticas (diretiva omp critical) garante que as escritas na saída sejam sincronizadas e sem conflitos.

O código a seguir foi a base do TSP seguido para criação da versão paralelizada. O principio são as funções de calcular distancia euclidiana entre dois pontos e o calculo de menor caminho.

```
// Funcao para calcular a distancia euclidiana entre dois pontos
double calcularDistancia(pair<int, int> ponto1, pair<int, int> ponto2)
{
    int dx = ponto1.first - ponto2.first;
    int dy = ponto1.second - ponto2.second;
    return sqrt(dx * dx + dy * dy);
}

// Funcao para calcular a menor distancia
double tsp(const vector<vector<double>> &distancias,
vector<bool> &visitado, int atual, int count, double custoAtual,
double &menorCusto, vector<int> &caminhoAtual,
vector<int> &melhorCaminho, int nos, int pontoFinal)
{
    if (count == nos - 1)
    {
        custoAtual += distancias[atual][pontoFinal];
        caminhoAtual.push_back(pontoFinal);
        if (custoAtual < menorCusto)
        {
            menorCusto = custoAtual;
            melhorCaminho = caminhoAtual;
        }
        caminhoAtual.pop_back();
        return menorCusto;
    }

    for (int i = 0; i < nos; ++i)
    {
```

```
        if (!visitado[i] && i != pontoFinal)
        {
            visitado[i] = true;
            caminhoAtual.push_back(i);
            tsp(distancias, visitado, i, count + 1, custoAtual
            + distancias[atual][i], menorCusto, caminhoAtual,
            melhorCaminho, nos, pontoFinal);
            caminhoAtual.pop_back();
            visitado[i] = false;
        }
    }

    return menorCusto;
}
```

Tendo como base essas funções alteramos a chamada delas utilizando a opemMP para colocar cada chamada dessas funções em threads diferentes. O código a seguir mostra como foi chamada essas funções.

```
    // Calcula as distancias entre os pontos
    int tid;
#pragma omp parallel for schedule(dynamic) num_threads(NUM_THREADS)
shared(distancias, coordenadas, quantidadeNos)
private(tid) default(none)

    for (int i = 0; i < quantidadeNos; ++i)
    {
        for (int j = 0; j < quantidadeNos; ++j)
        {
            if (i != j)
            {
                distancias[i][j] = calcularDistancia(coordenadas[i],
                coordenadas[j]);
#pragma omp critical
                {
                    tid = omp_get_thread_num();
                    printf("Thread: %d calculou a distancia
                    entre os pontos %d e %d\n", tid, i + 1, j + 1);
                }
            }
            else
            {
```

```
                distancias[i][j] = INFINITY;
            }
        }
    }

    // Paralelizacao das execucoes da funcao tsp
#pragma omp parallel for schedule(dynamic) num_threads(NUM_THREADS)
shared(distancias, quantidadeNos, pontoFinal, quantidadeIniciais,
arqSaida) private(tid) default(none)

    for (int inicio = 0; inicio < quantidadeIniciais; ++inicio)
    {
        tid = omp_get_thread_num();
        printf("Thread: %d iniciando TSP para ponto
        inicial %d\n", tid, inicio + 1);

        int *visitado = (int *)malloc(quantidadeNos * sizeof(int));
        int *caminhoAtual = (int *)malloc(quantidadeNos * sizeof(int));
        int *melhorCaminho = (int *)malloc(quantidadeNos * sizeof(int));
        double menorCusto = INFINITY;

        auto start = high_resolution_clock::now();

        for (int i = 0; i < quantidadeNos; i++)
            visitado[i] = 0;

        visitado[inicio] = 1;
        caminhoAtual[0] = inicio;
        tsp(distancias, visitado, inicio, 1, 0,
        &menorCusto, caminhoAtual, melhorCaminho,
        quantidadeNos, pontoFinal);

        auto end = high_resolution_clock::now();
        auto duration = duration_cast<microseconds>(end - start);

#pragma omp critical
        {
            printf("Thread %d finalizou TSP para ponto inicial %d
            com menor custo %.2f\n", tid, inicio + 1, menorCusto);
```

```
            fprintf(arqSaida, "Menor distancia comecando do
            ponto %d: %.2f\n", inicio + 1, menorCusto);

            fprintf(arqSaida, "Caminho percorrido: ");
            for (int i = 0; i < quantidadeNos; ++i)
                fprintf(arqSaida, "%d ", melhorCaminho[i] + 1);
            fprintf(arqSaida, "\nTempo de execucao: %lld
            microssegundos\n\n", duration.count());
        }

        // Libera a memoria alocada para cada thread
        free(visitado);
        free(caminhoAtual);
        free(melhorCaminho);
    }
```

5.3 Validação

A validação do algoritmo foi realizada comparando os resultados da implementação paralela com uma versão sequencial. O foco da validação foi garantir que o custo total da rota e o caminho obtido na versão paralela fossem equivalentes aos da versão sequencial, garantindo a correção do algoritmo.

Além disso, foram feitos testes com diferentes quantidades de cidades e pontos de partida, de modo a avaliar o comportamento do algoritmo em termos de escalabilidade e desempenho. Cada experimento foi repetido várias vezes para garantir a consistência dos resultados.

5.4 Conjunto de Dados

O conjunto de dados utilizado para testar o algoritmo consiste em coordenadas bidimensionais de cidades simuladas. Para cada instância do problema, as coordenadas de cada cidade são geradas aleatoriamente dentro de um plano cartesiano. Foram testadas diferentes instâncias com números variados de cidades (por exemplo, 10, 20, 50, e 100 cidades), a fim de analisar o comportamento do algoritmo em diferentes escalas.

Os arquivos de entrada contêm:

- O número total de cidades.
- O número de pontos de partida considerados na execução do TSP.
- As coordenadas x e y de cada cidade.

5.5 Métricas

As métricas utilizadas para avaliar o desempenho da solução foram:

- Tempo de execução: Medido em microssegundos, tanto para a versão paralela quanto para a sequencial, comparando o ganho de desempenho obtido com o uso de paralelismo.
- Custo total da rota: A menor distância total percorrida ao visitar todas as cidades, utilizada para validar a corretude da solução.
- Uso de threads: A eficiência da distribuição de tarefas entre os diferentes threads, monitorada através de prints que indicam qual thread está realizando cada tarefa.

Essas métricas foram escolhidas para demonstrar a eficiência da implementação paralela em termos de tempo e corretude, bem como para analisar como o algoritmo escala com o aumento do número de cidades e threads.

6 Resultado e Discussão

Os resultados obtidos indicaram uma redução significativa no tempo de execução da versão paralelizada em relação à versão sequencial, especialmente para conjuntos de dados maiores. A paralelização permitiu que o cálculo do caminho mais curto a partir de diferentes pontos iniciais fosse realizado simultaneamente, distribuindo a carga computacional de forma eficiente.

Primeiramente vamos discutir sobre os resultados encontrados na variação do numero de threads.

Para esse teste foi usado uma entrada de 14 pontos, 5 pontos iniciais e o ponto final no 13. E foi gerado a tabela a seguir com os resultados:

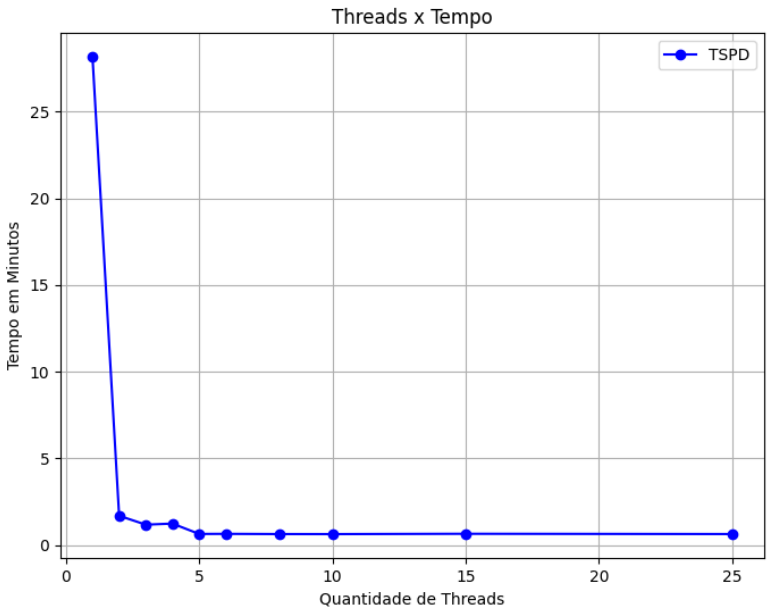

Figura 2: Evolução do tempo gasto conforme aumenta o número de threads

NumThreds	Tempo Ms	Tempo Min
1	1690169622	28.169494
2	100988106	1.683135
3	70104311	1.168405
4	73931601	1.232193
5	38012616	0.633544
6	38328047	0.638801
8	37602236	0.626704
10	37443089	0.624051
15	38572233	0.642871
25	37635453	0.627258

Tabela 1: Tabela Da Correlação do Numero de Threds em Relação ao Tempo

Utilizando esses resultados como base podemos perceber que em um certo ponto o numero de threds deixa de influenciar significativamente a velocidade de execução, isso se dá muito por conta da quantidade de pontos iniciais escolhida para o exemplo já que a parte mais demorada é a parte do calculo de menor caminho e quando eles estão sendo executados todos simultaneamente melhora muito

o desempenho.

Outro resultado verificado foi a relação do numero de cidades totais com o tempo de execução. Para esse teste foi usado um valor fixo de 8 threads para o algorítimo paralelo.

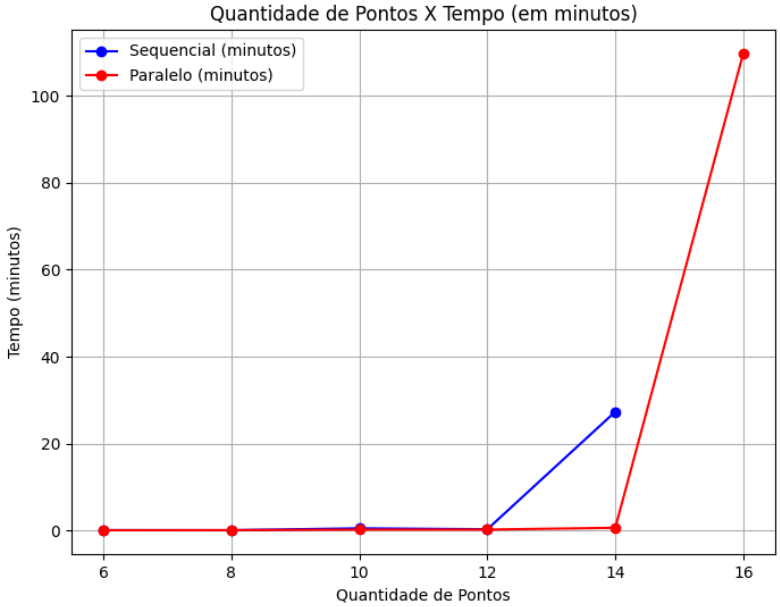

Figura 3: Evolução do tempo gasto conforme aumenta o número de pontos

NumPontos	TempSeq MS	TempSeq Min	TempPar MS	TempPar Min
6	5841150	0.097352	5128693	0.085478
8	4750785	0.079180	6349206	0.105820
10	31308823	0.0521814	12979795	0.216330
12	18449512	0.307492	13915669	0.231928
14	1639734738	27.328912	38731350	0.645522
16	>3000000000	>500	6581379845	109.689664

Tabela 2: Caption

Observando esses dados outra observação é que para entrada de dados pequena a variação é muito pequena, porém quando a entrada vai aumentando o tempo vai expandindo exponencialmente, até no ponto observado o algoritmo sequencial não conseguiu concluir em tempo hábil o teste com 16 pontos, chegando a demorar mais de 8 horas e ainda assim não terminando a execução.

7 Conclusão

Os resultados apresentados demonstram a eficiência da paralelização no contexto do problema do caixeiro viajante (TSP). A implementação paralela do algoritmo usando OpenMP conseguiu reduzir significativamente o tempo de execução, especialmente à medida que o número de pontos e threads aumentou. Observou-se uma melhoria considerável quando o número de threads aumentou de 1 para 5, com ganhos marginais adicionais até 8 threads. Acima desse número, os benefícios se estabilizaram, indicando que a sobrecarga de gerenciamento de threads começa a compensar os ganhos de paralelismo, especialmente quando a quantidade de pontos iniciais é limitada.

Ao comparar os resultados da execução sequencial e paralela para diferentes tamanhos de entrada, constatou-se que a paralelização torna-se especialmente vantajosa à medida que o número de pontos aumenta. Nos experimentos com 16 pontos, o algoritmo sequencial não foi capaz de produzir uma solução em tempo hábil, enquanto a versão paralela conseguiu executar em pouco mais de 109 segundos. Esse comportamento confirma que, em cenários de grande escala, a paralelização é uma estratégia essencial para lidar com a complexidade computacional do TSP.

Entretanto, também foi evidenciado que para entradas menores, a diferença entre a execução sequencial e paralela não é tão significativa, sugerindo que a paralelização pode não ser necessária em instâncias de menor porte. Portanto, a escolha entre uma abordagem sequencial ou paralela depende do tamanho do conjunto de dados e do tempo de resposta exigido.

Em conclusão, a abordagem paralela oferece uma solução eficiente para problemas de TSP em larga escala, onde a execução sequencial se torna impraticável. Contudo, a seleção adequada do número de threads e a análise do tamanho da entrada são cruciais para otimizar o desempenho geral do algoritmo.

8 Referências

- **O que é TSP**.Disponível em: <https://simpliroute.com/pt/blog/problema-do-caixeiro-viajante-tsp>. Acesso em: Set. 2024.
- **Aplicaçao do Problema do Caixeiro Viajante**.Disponível em:<https://ifrs.edu.br/veranopolis/wp-content/uploads/sites/10/2022/04/Artigo-TCC-Ana-Paula-Picolo-2021-Pub.pdf>. Acesso em: Set. 2024.
- **Algoritmo Base de Estudo**.Disponível em:<https://github.com/RodolfoHerman/caixeiro-viajante-com-programacao-dinamica>. Acesso em: Set. 2024.
- **OpenMP e Como Usar**.Disponível em:<Slides Vistos em Aula>. Acesso em: Set. 2024.