Análise de Paralelismo de Força Bruta no Caixeiro Viajante usando OpenMP

Richard Fernando Heise Ferreira
GRR20191053

9 de outubro de 2023

1 Recorte do Kernel do algoritmo

```
void tsp (int depth, int current_length, int *path) {
    int i;
    if (current_length >= min_distance) return;
    if (depth == nb_towns) {
        current_length += dist_to_origin[path[nb_towns - 1]];
        if (current_length < min_distance)
            min_distance = current_length;
    } else {
        int town, me, dist;
        me = path[depth - 1];
        for (i = 0; i < nb_towns; i++) {
            town = d_matrix[me][i].to_town;
            if (!paths[town]) {
                path[depth] = town;
                paths[town] = 1;
                dist = d_matrix[me][i].dist;
                tsp (depth + 1, current_length + dist, path);
                paths[town] = 0;
            }
        }
    }
}
```

A função `tsp` implementa uma abordagem de força bruta para resolver o Problema do Caixeiro Viajante (TSP - *Traveling Salesman Problem*). Ela recebe três parâmetros:

- `int depth`: Representa a profundidade atual na busca recursiva.
- `int current_length`: Armazena o comprimento atual do caminho percorrido.
- `int *path`: É um vetor que mantém o caminho atualmente percorrido.
- **Corte por Viabilidade**:
  - A primeira verificação condicional, `if (current_length >= min_distance) return;`, é um corte por viabilidade. Se o comprimento atual (`current_length`) exceder ou for igual à distância mínima conhecida (`min_distance`), não há necessidade de explorar mais esse caminho, uma vez que já sabemos que ele não levará a uma solução melhor.

- **Caso Base da Recursão**:
  - A segunda verificação condicional, `if (depth == nb_towns)`, representa o caso base da recursão. Se todas as cidades foram visitadas:
    * Adicionamos a distância da última cidade visitada de volta à cidade de origem ao `current_length`.
    * Verificamos se o comprimento total do caminho atual (`current_length`) é menor que a distância mínima conhecida (`min_distance`). Se for o caso, atualizamos `min_distance` com o novo valor.
- **Recursão para Explorar Todas as Possibilidades**:
  - Se a função não entrou nos casos anteriores, isso significa que precisamos continuar a recursão para explorar todas as possibilidades de caminhos.
  - Iniciamos um loop que itera sobre todas as cidades possíveis a partir da cidade atual (`me`).
  - Para cada cidade não visitada (`!paths[town]`), fazemos o seguinte:
    * Adicionamos a cidade atual ao vetor `path` na posição `depth`, representando o próximo ponto a ser visitado.
    * Marcamos a cidade como visitada, para que não seja explorada novamente na mesma iteração.
    * Calculamos a distância da cidade atual (`me`) para a cidade que estamos explorando (`town`) usando uma matriz de distâncias (`d_matrix`).
    * Chamamos a função `tsp` recursivamente com uma profundidade aumentada em 1 e atualizamos o comprimento atual (`current_length`) somando a distância da matriz.
    * Após a chamada recursiva, desmarcamos a cidade como não visitada, para que outros caminhos possam explorá-la em iterações subsequentes.

O algoritmo continua explorando todas as permutações possíveis das cidades até atingir a condição de parada ou até que todas as possibilidades tenham sido verificadas. O resultado final será a distância mínima possível que visita todas as cidades uma vez e retorna à cidade de origem. É importante destacar que essa abordagem é de força bruta e pode ser extremamente lenta para um grande número de cidades devido à sua complexidade fatorial.

2 Estratégia vitoriosa

```
void tsp_aux(int depth, int current_length, int *path, int *paths) {

if (current_length >= min_distance) return;

if (depth == nb_towns) {
    current_length += dist_to_origin[path[nb_towns - 1]];
    if (current_length < min_distance) {
        min_distance = current_length;
    }
    return;
}
int me = path[depth - 1];
for (int i = 0; i < nb_towns; i++) {
    int town = d_matrix[me][i].to_town;
    if (!paths[town]) {
        int dist = d_matrix[me][i].dist;
        path[depth] = town;
        paths[town] = 1;
```

```
            tsp_aux(depth + 1, current_length + dist, path, paths);
            paths[town] = 0;
        }
    }
}
void tsp(int depth, int current_length, int *path, int *paths) {

    if (current_length >= min_distance) return;

    if (depth == nb_towns) {
        current_length += dist_to_origin[path[nb_towns - 1]];
        if (current_length < min_distance) {
            min_distance = current_length;
        }
        return;
    }
    int me = path[depth - 1];

    #pragma omp parallel for schedule(guided) reduction(min:min_distance)
    for (int i = 0; i < nb_towns; i++) {
        int *path_local = (int *)malloc(nb_towns * sizeof(int));
        int *paths_local = (int *)malloc(nb_towns * sizeof(int));
        memcpy(path_local, path, nb_towns * sizeof(int));
        memcpy(paths_local, paths, nb_towns * sizeof(int));

        int town = d_matrix[me][i].to_town;
        if (!paths_local[town]) {
            int dist = d_matrix[me][i].dist;
            path_local[depth] = town;
            paths_local[town] = 1;
            tsp_aux(depth + 1, current_length + dist, path_local, paths_local);
            paths_local[town] = 0;
        }

        free(path_local);
        free(paths_local);
    }
}
```

A estratégia de paralelização utiliza #pragma omp parallel for com a configuração schedule(dynamic) para dividir o trabalho de maneira dinâmica entre as threads disponíveis. Cada thread processa diferentes cidades simultaneamente, explorando eficazmente o espaço de busca do TSP. A cláusula reduction(min:min_distance) garante que o menor valor encontrado entre todas as threads seja preservado como o resultado final, otimizando o TSP ao explorar várias cidades em paralelo, sem comprometer a integridade dos resultados.

Além disso, a escolha do agendador no modo dinâmico (dynamic) mostrou-se mais eficaz na distribuição de carga em comparação com os modos estático (static) ou guiado (guided). O modo estático é ideal para cargas de trabalho uniformemente distribuídas, enquanto o modo guiado é útil quando os tamanhos das tarefas variam à medida que o algoritmo avança. No entanto, no contexto deste problema, o modo dinâmico provou ser mais eficiente, proporcionando um melhor balanceamento de carga e, consequentemente, melhorando o desempenho da paralelização.

3 Metodologia

- Versão do S.O.: Linux Mint 21 (Vanessa)
- Versão do Kernel: 5.15.0-73-generic
- Compilador: gcc
- Flags de compilação sequencial: -O3
- Flags de compilação paralela: -O3 -fopenmp
- Modelo do Processador: Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz. 8 Núcleos lógicos.

Os testes foram conduzidos em um ambiente padrão, com um número mínimo de processos em execução. Foram utilizados quatro cenários diferentes, com um total de 16, 17, 18 e 19 cidades envolvidas. Para cada cenário, foram realizadas análises de média e desvio padrão, obtidas por meio de algumas execuções sequenciais e algumas execuções paralelas.

Especificamente, para o cenário com 16 cidades, realizamos 10 execuções em paralelo e 10 sequenciais. Com 17 foram 10 paralelas e 5 sequenciais. Para o cenário com 18 cidades, foram feitas 5 execuções paralelas e 2 sequenciais. Finalmente, para o cenário mais complexo com 19 cidades, optamos por 3 execuções em paralelo e 1 sequencial.

Essa variação na quantidade de execuções sequenciais e paralelas se deve ao tempo necessário para a conclusão dos testes. Por exemplo, a execução sequencial para 19 cidades levou mais de 3 horas, o que justificou a preferência por mais execuções em paralelo para economizar tempo.

Além disso, para as tabelas de speedup e eficiência foram utilizados casos triviais feitos manualmente que simulam uma linha "reta"entre as cidades, não necessariamente são casos ótimos, mas são casos bons.

Todos os testes foram gerados usando um script em python para gerar testes ou feitos manualmente para ser um caso trivial fixo.

Todos os arquivos utilizados nos testes, bem como os resultados obtidos, estão disponíveis no GitHub e podem ser acessados no final deste relatório para referência e análise detalhada.

4 Medições de Tempo

Esse é o tempo do algoritmo nos casos de destes descritos na metodologia. A porcentagem de tempo que o algoritmo passa na seção paralela foi de 99.99996%. Na seção sequencial só há leitura de entradas, cálculo da heurística de distâncias euclidianas e manipulação de memória, portanto, é um tempo muito pequeno em todos os casos, cerca de 0,00004%.

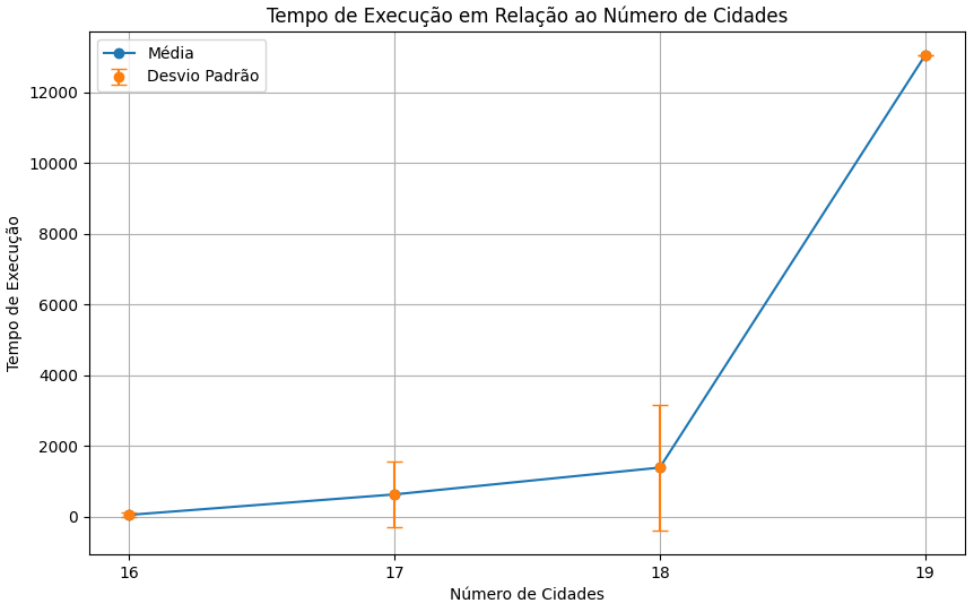

Figura 1: Tempo de execução sequencial

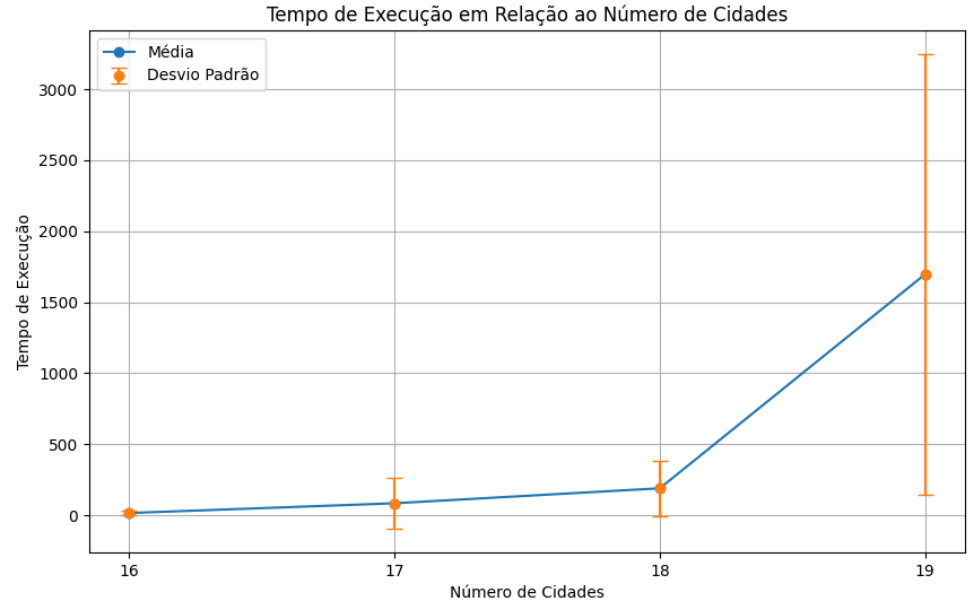

Figura 2: Tempo de execução paralelo

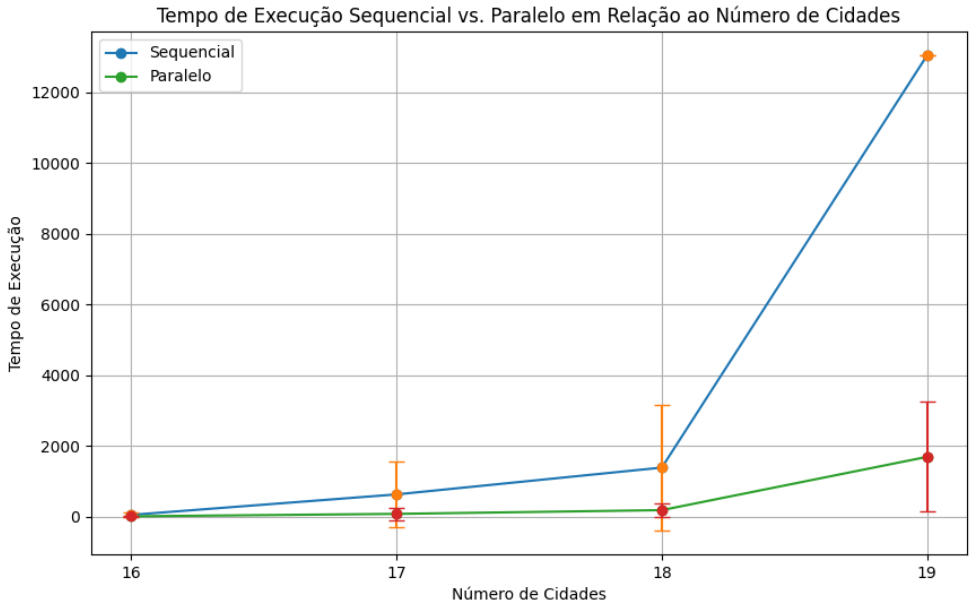

Figura 3: Comparativo

5 Lei de Amdahl

1. Speedup teórico para 2 núcleos (Beta = 0,000036): 1,999928.
2. Speedup teórico para 4 núcleos (Beta = 0,00004): 3,999845.
3. Speedup teórico para 8 núcleos (Beta = 0,000055): 7,996921.
4. Speedup Teórico Infinito (beta médio = 0,0000436): 22.935,779816.

O tempo paralelizável e puramente sequencial foi obtido através de profiling do código, há uma função que calcula o tempo de execução paralelo e total do algoritmo; diminuindo, então, o tempo paralelo do total obtêm-se o tempo sequencial.

6 Tabelas de Speedup e Eficiência

Figura 4: Tabela de Speedup

N	Número de CPUs			
	1 CPU	2 CPU	4 CPU	8 CPU
16	1	1,90	2,90	2,80
17	1	1,90	2,90	2,73
18	1	1,92	2,77	2,66
19	1	1,80	2,58	2,46

Figura 5: Tabela de Eficiência

N	Número de CPUs			
	1 CPU	2 CPU	4 CPU	8 CPU
16	1	0,95	0,72	0,35
17	1	0,95	0,73	0,34
18	1	0,96	0,69	0,33
19	1	0,90	0,65	0,30

Lembrando que essas tabelas foram feitas usando os casos triviais. Para o caso médio também há a tabela abaixo que mostra um speedup médio maior, porém não houve tempo hábil para coletar os dados de 2 e 4 cores.

Figura 6: Tabela de Speedup médio

N	Número de CPUs	
	1 CPU	8 CPU
16	1	3,67
17	1	7,57
18	1	7,36
19	1	7,69

7 Análise dos Resultados

Os resultados e experimentos revelam aspectos interessantes: o algoritmo paralelo conseguiu uma redução significativa no tempo médio de execução dos programas. No entanto, é importante notar que os resultados podem ser interpretados com certa cautela devido à escassez de dados fornecidos pelo autor. Observa-se um desvio-padrão considerável nos casos de teste, o que sugere que variações nas entradas podem ter um impacto substancial, inclusive na execução paralela. Portanto, não basta analisar o N, seria interessantes analisar os outliers de cada caso.

Além disso, a escolha de um caso trivial como referência pode não ser a métrica mais adequada, uma vez que os valores de eficiência e speedup apresentaram uma diferença significativa em relação aos casos médios. Especificamente, o speedup obtido para 8 núcleos no caso trivial difere substancialmente do speedup alcançado no caso médio. Portanto, é importante considerar a representatividade dos casos de teste ao avaliar o desempenho do algoritmo paralelo.

8 Escalabilidade dos Algoritmos

Os resultados obtidos a partir das tabelas e análises indicam que o algoritmo não demonstra um comportamento escalável.

Primeiramente, ao observar o speedup do algoritmo, notamos que, embora haja um aumento inicial no desempenho ao migrar de 1 para 2 CPUs, esse aumento não é sustentado à medida que o número de CPUs continua a crescer. Para N=16 a N=19, vemos uma diminuição do speedup quando passamos de 2 para 4 CPUs e de 4 para 8 CPUs. Esse comportamento indica que, à medida que mais recursos de processamento são adicionados, o algoritmo não consegue obter ganhos proporcionais no tempo de execução, o que é uma característica de escalabilidade limitada.

Além disso, analisando a eficiência, o algoritmo não mantém a mesma eficiência conforme o número de processadores aumenta. Quando aumentam-se o número de processadores e o N o algoritmo, ainda assim, não se mostra proporcional, portanto, não é fracamente escalável nem fortemente escalável.

9 Discussão

A implementação paralela do algoritmo para o Problema do Caixeiro Viajante (TSP) usando OpenMP demonstrou uma melhoria significativa no desempenho em relação à implementação sequencial. No entanto, a análise dos resultados revelou algumas nuances importantes a serem consideradas:

1. A variação considerável nos tempos de execução, especialmente nos casos de teste com desvio-padrão alto, levanta questões sobre a representatividade dos casos de teste e a estabilidade do algoritmo em diferentes cenários.

2. A escolha de um caso trivial como referência pode não ser adequada para avaliar o desempenho do algoritmo, uma vez que os resultados variam consideravelmente em relação aos casos médios.

3. A análise da escalabilidade indica que o algoritmo não é fortemente escalável, uma vez que o speedup e a eficiência diminuem à medida que mais CPUs são adicionadas.

Para uma análise mais completa e precisa do desempenho do algoritmo paralelo, seria necessário considerar uma variedade de cenários de teste, incluindo diferentes tamanhos de entrada e distribuições de cidades. Além disso, a coleta de dados para os casos de 2 e 4 CPUs no cenário médio seria importante para uma avaliação mais abrangente da escalabilidade.

Em resumo, a implementação paralela do TSP usando OpenMP apresenta melhorias significativas no desempenho em comparação com a versão sequencial. No entanto, a interpretação dos resultados requer uma análise cuidadosa das características dos casos de teste e da escalabilidade do algoritmo em diferentes cenários.

Repositório: https://github.com/RichardHeise/TSP-OpenMP