Explorando paralelismo em um algoritmo guloso para o problema do caixeiro-viajante

Vinicius Gabriel Machado[1]

[1]Bolsista de Pós Graduação em Informática (Mestrado)
Universidade Federal do Paraná (UFPR)

viniciusgabrielmachado@gmail.com

1. Introdução

Este documento apresenta um estudo realizado sobre o problema do caixeiro-viajante. A atividade, proposta pela disciplina de programação paralela, consiste em explorar paralelismo em nível de *threads*, utilizando OpenMP[1], para uma implementação gulosa do problema clássico. Experimentos realizados com dezenas de instâncias, de diferentes tamanhos e executadas múltiplas vezes, validam a solução implementada, demonstrando um comportamento fortemente escalonável até 6 núcleos físicos. A Seção 2 apresenta uma visão geral do problema e as suas implementações. Seção 3 detalha os experimentos realizados, metodologia utilizada, resultados práticos e teóricos obtidos. Por fim, Seção 4 apresenta as considerações finais.

2. O problema

O problema do caixeiro-viajante (*Travelling Salesman Problem*, TSP), na sua forma mais conhecida e mais simples, consiste em: dado um conjunto de cidades e as distâncias entre cada par de cidades, determinar o caminho de menor tamanho, que percorra todas as cidades exatamente uma vez e retorne a cidade de origem.

Na versão aqui discutida, as distâncias são calculadas em tempo de execução pelo programa, utilizando um algoritmo guloso que determina a distância entre todos os pares e as ordena em ordem crescente. Além disso, também trabalhamos com a suposição de que todas as cidades são alcançáveis por todas as outras. Por fim, utilizando as informações obtidas, o algoritmo recursivo TSP explora todos os possíveis caminhos e determina aquele que soluciona o problema. A Subseção 2.1 apresenta a implementação sequencial do TSP e a Subseção 2.2 explora a estratégia de paralelização utilizada neste algoritmo.

2.1. O problema sequencial

A Figura 1 apresenta o algoritmo sequencial para o TSP, provido pelo professor da disciplina e utilizado na nona edição da maratona de programação paralela da WSCAD[2].

O algoritmo é uma implementação recursiva da busca em profundidade, amplamente conhecida na área de ciência da computação, que dado um grafo e um nó inicial, irá explorar todos os decentes deste nó antes dos seus vizinhos. No problema aqui discutido, os nós são equivalentes as cidades, o grafo é um grafo completo, que representa as

[1]https://www.openmp.org
[2]http://wscad.sbc.org.br/edicoes/index.html

distâncias entre todos os pares de cidades. Dada uma cidade inicial[3], as cidades restantes serão exploradas por ordem crescente de distância, ignorando aquelas que já fizerem parte do caminho percorrido. A exploração de um caminho pode ser interrompida caso o comprimento do caminho seja maior do que o menor encontrado até o momento[4], ação que pode ser tomada graças a natureza de exploração por ordem crescente de distância do algoritmo. Ao atingir a profundidade máxima, na qual todas as cidades já foram percorridas, o comprimento total é somado com a distância até a cidade de origem e caso o valor resultante seja menor do que o menor caminho encontrado, ele será atualizado pelo novo valor.

Figura 1. Recorte do núcleo do algoritmo sequencial.

```
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 (!present (town, depth, path)) {
                path[depth] = town;
                dist = d_matrix[me][i].dist;
                tsp (depth + 1, current_length + dist, path);
            }
        }
    }
}
```

2.2. Paralelização realizada

A fim de paralelizar o código abordado na subseção anterior, diversas pesquisas e possibilidades foram feitas e testadas, respectivamente. Este caminho, que resultou na versão final, será abordado em seguida. Por hora, iremos discutir a estratégia vitoriosa.

Para a paralelização, o código sequencial foi dividido em duas versões, uma iterativa (Figura 2) e uma recursiva (Figura 3). A recursiva é idêntica a versão sequencial, com um pequeno adendo, a adição de uma diretiva para garantir a atribuição atômica na variável global que determina o comprimento do menor caminho encontrado até então. Esta modificação foi feita a fim de evitar situações de corrida, impedindo o acesso a variável enquanto estiver sendo atualizada. A iterativa por sua vez, constitui a ideia principal do algoritmo paralelo. Ela consiste no desentrelaçamento dos primeiros 3 níveis da recursão original (1 nível da cidade inicial e os 2 níveis seguintes). O código desta versão é semelhante ao código observado na versão sequencial, com ajustes a fim de garantir o mesmo comportamento na versão iterativa, como checagem do comprimento e atribuição da profundidade correta. A estratégia de paralelização pode ser resumida em: utilizando

[3]Aqui consideramos a cidade inicial como sendo a primeira fornecida.

[4]Inicialmente o comprimento do menor caminho é definido como o maior inteiro representável.

uma única *thread*, percorrer os primeiros 3 níveis da recursão, e ao atingir a terceira cidade dar sequência por meio da criação de *tasks* que irão percorrer o restante do caminho utilizando a versão recursiva. Como esperado, algumas modificações são necessárias para o processamento adequado das *tasks* pelas *threads* disponíveis. Mais especificamente, o caminho deixou de ser um ponteiro e se tornou um vetor próprio para cada *task*.

Figura 2. Recorte do procedimento principal do algoritmo paralelizado.

```
void tsp () {
    #pragma omp parallel default(none) shared(nb_towns, d_matrix, min_distance)
    #pragma omp single nowait
    for (int i = 0; i < nb_towns; i++) {
        int path_threaded[nb_towns], town, dist;
        int current_length = d_matrix[0][i].dist;

        int depth = 1;
        town = d_matrix[0][i].to_town;

        path_threaded[0] = 0;
        if (!present (town, depth, path_threaded) & (current_length < min_distance)) {
            path_threaded[1] = town;
            depth = 2;

            for (int j = 0; j < nb_towns; j++) {
                town = d_matrix[path_threaded[1]][j].to_town;
                if (!present (town, depth, path_threaded)) {
                    dist = d_matrix[path_threaded[1]][j].dist;

                    path_threaded[2] = town;

                    #pragma omp task firstprivate(path_threaded)
                    tsp_recursive (depth + 1, current_length + dist, path_threaded);
                }
            }
        }
    }
}
```

Figura 3. Recorte do procedimento alternativo do algoritmo paralelizado.

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

O código paralelo observado foi resultado de diversas pesquisas e experimentos realizados por parte do autor. Inicialmente, havia a ideia de modificar o algoritmo a fim de distribuir todas as possíveis permutações igualmente entre as *threads*. O que, ao fazer uma busca pela literatura relacionada, demonstrou ser um problema mais complexo do que se esperava, ainda mais considerando a perspectiva de paralelização. Sendo assim, optou-se pela realização de experimentos utilizando os conhecimentos adquiridos na disciplina.

Primeiramente, tentou-se paralelizar e dividir a cláusula *else* em 3 situações baseado na profundidade. A primeira criaria *threads* no primeiro nível, a segunda *tasks* no segundo nível e a terceira seria semelhante ao código já existente. Uma ideia parecida com a final. Entretanto, esta demonstrou ser altamente ineficiente, com as *threads* apresentando 60% de utilização ou menos. Não se sabe exatamente o motivo disto, mas acreditou-se que tenha sido causado pela estrutura do paralelismo, que paraleliza apenas uma metade da função recursiva, o que talvez fizesse as *threads* ficarem presas na barreira da região paralela da segunda metade. Desta forma, surgiu a ideia de envolver o código inteiro da TSP em uma única região paralela, tornando os primeiros 3 níveis iterativos, que na maior profundidade iriam chamar uma rotina distinta para realização da recursão.

3. Experimento Realizado

Objetivando realizar uma avaliação prática do algoritmo paralelo, diversos experimentos foram conduzidos. Estes variam o número de cidades (dificuldade do problema) e o número de *threads* utilizadas, a fim de testar a escalabilidade da solução desenvolvida. Desejando fornecer uma análise mais aprofundada e resultados mais significativos, cada experimento foi conduzido com dezenas de instâncias do problema do caixeiro-viajante e estes foram repetidos dezenas de vezes. Mais detalhes sobre a metodologia utilizada serão explorados na Subseção 3.1. Subseção 3.2 expõe os resultados encontrados na prática, analisando-os e comparando com resultados teóricos.

3.1. Metodologia de experimentação

Dada uma especificação do tamanho do problema, por exemplo 15 cidades, foram geradas 50 instâncias[5] aleatórias com tamanho 15, por meio de uma rotina em Python, com suas coordenadas de longitude limitadas entre 1 e 360 e de latitude entre 1 e 180. Os experimentos foram conduzidos desta maneira, e não com uma única instância com muitas cidades, a fim de analisar o impacto de cada instância no tempo de execução. Sendo um problema de espaço de busca, algumas instâncias podem ser solucionadas logo no começo da busca, já para outras pode ser necessário percorrer o espaço inteiro até que a solução ideal seja encontrada. Estas instâncias são todas solucionadas em uma execução do algoritmo, que por sua vez é repetida 10 vezes[6], a fim de analisar e descartar eventuais impactos por parte do Sistema Operacional (SO) e outros processos em execução na máquina de teste durante a experimentação, sobre os resultados observados. Finalmente, este conjunto de testes é realizado para o algoritmo sequencial e o paralelo, com número de *threads* variando entre $t = \{1, 2, 3, 6, 9, 12\}$[7], extraindo o tempo de execução (em segundos) dedicado a cada instância e as execuções ao todo, por meio de rotinas de medição de tempo real próprias da biblioteca OpenMP para C.

Os experimentos foram executados no seguinte ambiente:

- Sistema Operacional: Ubuntu 22.04.1 LTS;
- Kernel: 6.2.0-33-generic;
- Compilador: gcc 11.4.0;
- Flags de compilação: -O3 -fopenmp -lm;

[5]Número arbitrário, escolhido a fim de garantir pelo menos 20 segundos de tempo de execução total.
[6]Valor considerado suficientemente grande, validado pelos resultados.
[7]Valores escolhidos de acordo com o processador utilizado.

- Processador: Ryzen 5 3600;
- Memória RAM: 2x8 GB HyperX Fury 3000 MHz (DDR4).

A fim de remover possíveis fontes de interferência, o controlador de frequência (*scaling governor*) do SO foi definido para o modo *performance*, o acesso a internet foi removido e o SO foi executado sem interface gráfica. Os testes da versão sequencial foram fixados na primeira *thread* do processador, removendo o impacto negativo da mudança de *thread* que pode ser realizada pelo SO, e que pode resultar em um maior número de *cache misses*. Os testes da versão paralela também receberam um tratamento semelhante, com as *threads* sendo fixadas por meio da variável de ambiente OMP_PROC_BIND com uma abordagem distribuída (*spread*). Finalmente, os resultados da versão paralela foram verificados com a sequencial. Todos os *scripts*, código fonte, entradas utilizadas, saídas obtidas e informações mais detalhadas sobre o ambiente podem ser encontrados no github do trabalho[8].

3.2. Resultados práticos

A seguir são apresentados os resultados obtidos na prática. As Tabelas 1 e 2 mostram a média geométrica (para as instâncias)/média aritmética (para a execução completa), o desvio padrão e uma porcentagem comparando o desvio padrão com a média geométrica/média aritmética de, respectivamente, os valores observados para as instâncias e o tempo total das 50 instâncias. Para as instâncias, primeiramente realizou-se a média das 10 execuções de cada uma, com estes 50 valores foi então obtido a média geométrica e o desvio padrão. Já para o tempo total, foi feita a média aritmética e o desvio padrão das 10 execuções. A escolha da média geométrica para as instâncias está ligado ao fato de que os valores observados são altamente distintos, o que pode impactar significativamente na média aritmética, fazendo com que ela tenda aos *outliers*.

Como podemos observar, os resultados obtidos para as execuções completas são bem comportados, com desvio padrão menor que 1% da média aritmética em todos os testes. Isso é esperado, tendo em vista as precauções para remoção de eventuais interferências. Entretanto, há certas considerações a serem feitas. Primeiramente, a execução paralela com apenas uma *thread* precisou de mais tempo em todas as situações quando comparado com a versão sequencial. Isto também é um comportamento esperado em algoritmos paralelos, uma vez que há a introdução de *overhead* ao trabalhar com os mecanismos de paralelização e a modificação da lógica necessária para o funcionamento paralelo. Segundo, é possível notar uma redução do tempo de execução ao utilizar mais de 6 *threads*, porém, já é possível perceber ganhos não muito expressivos.

Em relação aos resultados obtidos por instância, é possível notar os mesmos adendos feitos para os resultados das execuções completas, com a exceção de uma observação, o comportamento. Este flutua drasticamente, com um desvio padrão muito maior do que a média geométrica. O que por sua vez nos diz que foi uma boa escolha executar o algoritmo para mais de uma instância, já que uma instância pode ser um *outlier* e consequentemente não ser representativa da média do tempo necessário para a computação do problema com o tamanho especificado. Neste sentido, seria possível imaginar uma instância com 15 cidades sendo solucionada mais rapidamente que uma de 14 cidades, mesmo que possa ser difícil de acontecer. Entretanto, isto também nos indica que talvez fosse necessário um

[8]https://github.com/viniciusgm000/ParallelGreedyTSP

estudo mais aprofundado sobre o caso médio do problema, buscando determinar qual seria o número de instâncias realmente necessárias para obter uma alta confiança estatística sobre os resultados, mas que provavelmente não seria possível realizar em tempo hábil.

Em seguida, as Tabelas 3 e 4 apontam as métricas *speedup* e eficiência extraídas com as médias anteriores, tanto para as instâncias quanto para a execução completa. Nessas tabelas é possível notar mais claramente o que foi mencionado anteriormente. A eficiência do algoritmo paralelo com uma única *thread* é 70% da do sequencial, um impacto expressivo e esperado, pelos motivos já discutidos. A eficiência com 9 e 12 *threads* também se degradou, algo que também é esperado devido a natureza *Simultaneous Multi-Threading* (equivalente ao *Hyper-Threading* da Intel) em que a *cache* e os recursos de processamento são compartilhados. Entretanto, há um pequeno ganho, o que pode ser consequência dos acessos a memória, suspendendo uma *thread* que esteja esperando pela memória por outra que já tenha os valores necessários disponíveis. Por outro lado, o comportamento observado com 2, 3 e 6 *threads* é altamente promissor, com uma eficiência que varia entre 86% e 91%, tanto para as instâncias quanto para a execução completa. Um comportamento fortemente escalonável nas circunstâncias aqui definidas. Porém, vale a ressalva de que esse comportamento pode não ser observado em todas as situações, por exemplo, é possível imaginar um tamanho de entrada suficientemente pequeno ou um número de núcleos físicos suficientemente grande, que não tenha os mesmos resultados.

Tabela 1. Média geométrica e desvio padrão por instância.

	Threads	1		2	3	6	9	12
	Versão	Sequencial	Paralelo	Paralelo	Paralelo	Paralelo	Paralelo	Paralelo
	# Cidades							
Média geométrica instância	13	0.382	0.543	0.218	0.145	0.074	0.060	0.052
	14	1.539	2.397	0.859	0.569	0.290	0.237	0.204
	15	7.627	11.162	4.204	2.767	1.405	1.160	1.013
Desvio padrão instância	13	0.397	0.515	0.217	0.145	0.073	0.058	0.049
	14	3.838	4.984	2.078	1.377	0.704	0.567	0.480
	15	9.265	13.572	5.060	3.322	1.694	1.385	1.201
% Desvio padrão instância (x100)	13	103.981	94.748	99.918	100.062	99.308	96.694	94.485
	14	249.376	207.937	241.978	242.231	243.127	239.159	234.643
	15	121.476	121.594	120.371	120.045	120.561	119.329	118.598

Comparação com resultados teóricos

Visando realizar uma análise mais aprofundada, foram utilizadas as leis de Amdahl e Gustafson-Barsis para mensurar o *speedup* máximo teórico. Para isso, o tempo dedicado ao trecho sequencial dos algoritmos sequencial e paralelo foi medido para 15 cidades e 50 instâncias, levando em conta todas as operações que são realizadas fora da rotina principal, a TSP. Isso inclui: a inicialização e liberação das matrizes e vetores, leitura dos dados e cálculo das distâncias mínimas. Este processo foi repetido 10 vezes, a média realizada e a porcentagem determinada por meio da comparação com o tempo

Tabela 2. Média aritmética e desvio padrão por execução.

	Threads	1		2	3	6	9	12
	Versão	Sequencial	Paralelo	Paralelo	Paralelo	Paralelo	Paralelo	Paralelo
	# Cidades							
Média aritmética tempo total	13	24.774	34.439	13.918	9.271	4.716	3.812	3.268
	14	129.097	185.243	71.255	47.070	23.963	19.489	16.661
	15	537.870	769.570	294.996	194.263	98.881	81.547	71.306
Desvio padrão tempo total	13	0.187	0.075	0.018	0.028	0.010	0.010	0.007
	14	0.391	0.248	0.123	0.112	0.055	0.048	0.028
	15	1.835	1.862	0.741	0.483	0.271	0.129	0.158
% Desvio padrão tempo total (x100)	13	0.757	0.217	0.133	0.304	0.216	0.273	0.204
	14	0.303	0.134	0.173	0.238	0.228	0.247	0.167
	15	0.341	0.242	0.251	0.248	0.274	0.158	0.222

Tabela 3. Speedup e eficiência por instância.

	Threads	1	2	3	6	9	12
	# Cidades						
Speedup instância	13	0.703	1.755	2.637	5.167	6.345	7.376
	14	0.642	1.792	2.707	5.315	6.493	7.526
	15	0.683	1.814	2.756	5.429	6.574	7.533
Eficiência instância	13	0.703	0.878	0.879	0.861	0.705	0.615
	14	0.642	0.896	0.902	0.886	0.721	0.627
	15	0.683	0.907	0.919	0.905	0.730	0.628

médio da execução completa. Abaixo segue os valores observados, entre 0 e 1:

- Execução sequencial: 8.368e-7;
- Execução paralela com 6 threads: 5.493e-6.

Como podemos notar, a porcentagem dedicada a porção sequencial do código é extremamente baixa. Neste documento não entramos em detalhes sobre a implementação da rotina de cálculo das distâncias entre as cidades, a parte mais significativa da porção sequencial, mas ela contém 3 laços de repetição aninhados. Considerando isso poderíamos dizer que ela tem custo próximo a $\Theta(n^3)$. Logo, é esperado que, dada a natureza NP-difícil do problema do caixeiro-viajante, o trecho sequencial não seja uma porção significativa do tempo de execução. Com estes valores, os *speedups* máximos teóricos não contribuem significativamente para a discussão (Tabelas 5 e 6). Entretanto, podemos pensar nos motivos pelos quais foi observado uma eficiência de aproximadamente 90% ao invés dos teóricos 100%. Dentre as desvantagens relacionadas a paralelização de um código, acredita-se que a manipulação e gerenciamento de *tasks* tenha tido o maior impacto, seguida pelo acesso a variável global que guarda o comprimento do menor caminho encon-

Tabela 4. Speedup e eficiência por execução.

	Threads	1	2	3	6	9	12
	# Cidades						
Speedup tempo total	13	0.719	1.780	2.672	5.253	6.499	7.581
	14	0.697	1.812	2.743	5.387	6.624	7.748
	15	0.699	1.823	2.769	5.440	6.596	7.543
Eficiência tempo total	13	0.719	0.890	0.891	0.875	0.722	0.632
	14	0.697	0.906	0.914	0.898	0.736	0.646
	15	0.699	0.912	0.923	0.907	0.733	0.629

trado até então. A primeira, além de ter custos intrínsecos ao gerenciamento de *tasks*, também realiza a cópia das variáveis utilizadas, mais especificamente, o vetor de caminho, o que pode ocasionar um maior número de *misses* na *cache* do processador. Já a segunda pode ocasionalmente barrar as *threads* que desejam acessar a variável. Também poderíamos pensar no desperdício de processamento como causa de *overhead*, por exemplo, uma *thread* pode começar o processamento de um caminho e no meio ser informada por outra de que aquele caminho não será o menor.

Tabela 5. Lei de Amdahl: Speedup máximo teórico (n processadores).

2	3	6	9	12	∞
1.999	2.999	5.999	8.999	11.999	1195029

Tabela 6. Lei de Gustafson-Barsis: Speedup máximo teórico (n processadores).

2	3	6	9	12
1.999	2.999	5.999	8.999	11.999

4. Conclusão

Neste documento abordamos uma solução gulosa para um dos problemas clássicos da computação, o caixeiro-viajante. Uma classe de problemas que intriga cientistas da computação até hoje, devido a sua natureza NP-difícil. A solução original fazia uso de algumas eurísticas simples para reduzir o custo computacional, como utilização de uma matriz ordenada com as distâncias mínimas de todos os pares de cidades, para realizar o corte de caminhos não promissores. A solução paralelizada, por sua vez, faz uso de uma *thread* para a criação de *tasks*, que são então processadas pelas *threads* ociosas. Esta solução demonstrou ser altamente eficaz, expressando um comportamento fortemente escalonável com 90% de eficiência, em 10 execuções de 50 instâncias de 13, 14 e 15 cidades, para 2, 3 e 6 *threads* (o número de núcleos físicos). Acima de 6 *threads* também houve ganhos, mas menos expressivos, devido ao compartilhamento de recursos em sistemas SMT. Finalmente, comparamos os resultados com os esperados na teoria, o que expandiu nossa discussão para possíveis geradores de *overhead* em algoritmos paralelos, como gerenciamento de *tasks* e competição por acesso a uma única variável.