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

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 processes, utilizando OpenMPI[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 fracamente escalonável até 5 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.open-mpi.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, o autor utilizou uma abordagem semelhante à utilizada no trabalho anterior da disciplina[5], que consistia em paralelizar o mesmo código mas utilizando OpenMP ao invés de OpenMPI. A abordagem divide o código sequencial recursivo em dois segmentos, um iterativo e um recursivo. O segmento recursivo é idêntico ao código sequencial. Já o segmento iterativo consiste no desentrelaçamento dos primeiros 3 níveis da recursão original (1 nível da cidade inicial e os 2 níveis seguintes). A interação desses dois segmentos se dá por meio de uma estrutura produtor-consumidor. O código sequencial é executado por um processo produtor (Figura 2), que caminha até o terceiro nível de profundidade e aguarda um processo consumidor (Figura 3) solicitar um trabalho para realizar. Recebendo a solicitação, o produtor envia para o consumidor uma mensagem com informações de profundidade, comprimento e caminho até então, juntamente com a distância mínima atual. O consumidor que recebe a mensagem, atualiza a sua distância mínima e chama a função recursiva, com os dados

[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.
[5]https://github.com/viniciusgm000/ParallelGreedyTSP/blob/main/reports/openmp.pdf

recebidos, para dar continuidade no caminho. Ao final do processamento do caminho, o consumidor envia a sua distância mínima atual e solicita um novo trabalho para o produtor, que analisa se a distância mínima foi reduzida, e o ciclo se repete. Ao fim, quando não houver mais trabalho a ser realizado, o processo produtor aguarda novas solicitações e envia uma mensagem final com profundidade igual a -1, um sinal que o consumidor deve finalizar a sua execução.

Figura 2. Recorte do núcleo do processo produtor da versão paralela.

```
void tsp () {
    int *path, depth, current_length;
    int message[nb_towns + PATH_START];

    // Avoid copying by mapping path into the message
    path = message + PATH_START;

    if (process_rank == 0) {
        int town, dist, depth;
        for (int i = 0; i < nb_towns; i++) {
            current_length = d_matrix[i].dist;

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

            path[0] = 0;

            if (!present (town, depth, path)) {
                path[1] = town;
                depth = 2;

                for (int j = 0; j < nb_towns; j++) {
                    town = d_matrix[path[1] * nb_towns + j].to_town;
                    dist = current_length + d_matrix[path[1] * nb_towns + j].dist;

                    if (!present (town, depth, path) && dist < min_distance) {
                        path[2] = town;

                        // Receive process min_distance and send more work (length/depth/path) with current min_distance
                        MPI_Recv (&message[MIN_DISTANCE], 1, MPI_INT, MPI_ANY_SOURCE, STD_TAG, MPI_COMM_WORLD, &status);

                        if (message[MIN_DISTANCE] < min_distance)
                            min_distance = message[MIN_DISTANCE];

                        message[DEPTH] = depth + 1;
                        message[CURRENT_LENGTH] = dist;
                        message[MIN_DISTANCE] = min_distance;
                        // path already mapped into the message

                        MPI_Ssend (&message, nb_towns + PATH_START, MPI_INT, status.MPI_SOURCE, STD_TAG, MPI_COMM_WORLD);
                    }
                }
            }
        }

        // No more work to be done, await for processes to send a finalize signal
        message[DEPTH] = -1;
        for (int i = n_process - 1; i > 0; i--) {
            MPI_Recv (&message[MIN_DISTANCE], 1, MPI_INT, MPI_ANY_SOURCE, STD_TAG, MPI_COMM_WORLD, &status);

            if (message[MIN_DISTANCE] < min_distance)
                min_distance = message[MIN_DISTANCE];

            MPI_Ssend (&message, nb_towns + PATH_START, MPI_INT, status.MPI_SOURCE, STD_TAG, MPI_COMM_WORLD);
        }
    } else {
```

Como citado anteriormente, o código paralelo é resultado de uma combinação da ideia utilizada no trabalho anterior da disciplina e da ideia de processos produtores e consumidores. A primeira foi reutilizada já que havia o objetivo de realizar uma paralelização semelhante, mantendo a estrutura recursiva que cria tarefas no percorrer do caminho, que são computadas paralelamente por *threads* ociosas. Originalmente, um mecanismo de baixo nível do OpenMP se encarregava da criação e distribuição dessas tarefas (*tasks*) entre *threads* ociosas. Entretanto, o autor não encontrou algo semelhante para OpenMPI. Dessa forma, esse mecanismo foi implementado de forma explicita pelo autor, utilizando algumas ideias de produtor-consumidor.

Figura 3. Recorte do núcleo dos processos consumidores da versão paralela.

```
    } else {
        while (1) {
            // Send current min_distance (asking for more work) and receive more work (length/depth/path) with current min_distance
            MPI_Ssend (&min_distance, 1, MPI_INT, 0, STD_TAG, MPI_COMM_WORLD);
            MPI_Recv (&message, nb_towns + PATH_START, MPI_INT, 0, STD_TAG, MPI_COMM_WORLD, &status);

            // No more work to be done
            if (message[DEPTH] == -1)
                break;

            min_distance = message[MIN_DISTANCE];

            // No need to copy path
            tsp_recursive (message[DEPTH], message[CURRENT_LENGTH], message + PATH_START);
        }
    }
}
```

Até chegar na versão final, diversos testes e modificações foram realizadas. Começando pela inicialização do algoritmo, a leitura e cálculo da matriz de distâncias mínimas. A leitura é realizada pelo processo produtor e transmitida para os outros processos em um único vetor, evitando *overhead* de mensagens desnecessárias. O cálculo inicialmente era feito pelo produtor e enviado, porém, foi possível reduzir o tempo de execução realizando o cálculo em cada processo, removendo a comunicação de uma matriz N^2 e comunicando apenas o vetor $2N$ de entrada. Em relação aos segmentos produtores e consumidores, a fim de reduzir o número de comunicações, todas as informações necessárias para a atribuição de trabalho para um processo foram resumidas em um único vetor, a mensagem. Nesta mensagem é mapeada diretamente o caminho (o caminho aponta para um elemento da mensagem), visando não realizar a cópia do vetor a cada nova solicitação.

Alguns experimentos foram feitos com essa estrutura, a fim de aumentar ainda mais o desempenho. Por exemplo, o autor tentou agrupar diversas mensagens e realizar apenas um grande envio com muitas mensagens, com a ideia de que seria melhor lidar com o *overhead* de uma única mensagem do que de muitas. Entretanto, experimentos com essa estrutura de agrupamento de mensagens demonstraram que o desempenho era reduzido. Acredita-se que apesar dessa ideia ser interessante, o impacto na vazão das mensagens (atendimento a múltiplos consumidores) foi maior do que os ganhos na redução do *overhead* da comunicação. Outra ideia rejeitada foi a realização da otimização da rotina de teste da presença de uma cidade no caminho até então. Esta rotina percorria o vetor do caminho todas as vezes, o que aumenta drasticamente seu tempo de execução, se comparado com um vetor *hash*. Sendo assim, a rotina foi substituída por um vetor, com a suposição de que a vazão do produtor poderia ser aumentada desta forma. Apesar do tempo de execução ser reduzido pela metade, a eficiência continuou parecida, provavelmente porquê os consumidores também fazem uso da mesma rotina, logo, foi descartada essa ideia e mantida a versão original, para ser possível realizar uma comparação com os resultados obtidos no OpenMP. Por fim, também foi analisado o impacto de diferentes métodos de comunicação, como *MPI_Bsend, MPI_Send* e *MPI_Ssend*, mas não foi possível notar diferenças significativas.

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 processos consumidores criados, a fim de testar a escalabilidade da solução desenvolvida. Desejando fornecer uma análise mais aprofundada e resultados mais sig-

nificativos, 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 com 1 *host*. Subseção 3.3 expõe os resultados encontrados na prática com 3 *hosts*.

3.1. Metodologia de experimentação

Dada uma especificação do tamanho do problema, por exemplo 15 cidades, foram geradas 50 instâncias[6] 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[7], 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 processos consumidores variando entre $t = \{1, 2, 3, 4, 5, 10\}$[8], 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 OpenMPI 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;
- Mpirun: 4.1.2;
- Flags de compilação: -O3 -lm;
- Flags de execução: –map-by l2cache –bind-to hwthread;
- 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 os processos sendo fixados em *threads* de forma a evitar *oversubscribing* da cache L2, compartilhada pelo *multithreading*. Finalmente, os resultados da versão paralela foram verificados com a sequencial.

Além disso, o autor também decidiu realizar os mesmos experimentos em múltiplos *hosts*, utilizando 3 computadores do Departamento de INFormática da UFPR

[6] Número arbitrário, escolhido a fim de garantir pelo menos 20 segundos de tempo de execução total.
[7] Valor considerado suficientemente grande, validado pelos resultados.
[8] Valores escolhidos de acordo com o processador utilizado e levando em conta o processo extra produtor.

(DINF)[9]. Os testes foram conduzidos de forma a dividir o número de processos igualmente, adicionando processos incrementalmente nas máquinas mais ociosas. A metodologia foi semelhante a utilizada nos experimentos no computador pessoal do autor, com a diferença de que não foram feitas modificações no mapeamento dos processos nas *threads* do processador, já que os processadores utilizados não possuem *Hyperthreading* ou *Simultaneous Multithreading*.

- Número de *hosts*: 3;
- Sistema Operacional: LMDE 5;
- Kernel: 6.1.0-0.deb11.7-amd64;
- Compilador: gcc 10.2.1;
- Mpirun: 4.1.0;
- Flags de compilação: -O3 -lm;
- Processador: Intel Core i5-7500;
- Memória RAM: 1x8 GB.

Todos os *scripts*, código fonte, entradas utilizadas, saídas obtidas e informações mais detalhadas sobre o ambiente podem ser encontrados no github do trabalho[10], juntamente com o trabalho anterior em OpenMP.

3.2. Experimentos - 1 host

A seguir são apresentados os resultados obtidos na prática com 1 único *host*. Subseção 3.2.1 apresenta os resultados propriamente ditos. Subseção 3.2.2 compara os valores do OpenMP do trabalho anterior com os do OpenMPI. Subseção 3.2.3 compara as medições observadas com estimativas teóricas.

3.2.1. Resultados práticos - 1 host

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 um processo consumidor 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

[9]https://web.inf.ufpr.br/dinf/

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

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 10 processos consumidores (*oversubscribing*), porém, é possível perceber ganhos não muito expressivos, devido ao compartilhamento de recursos computacionais de cada núcleo entre as *threads* e da luta pela cache L2, resultando em *cache trashing*.

Em relação aos resultados por instância, algo semelhante ocorre, com a diferença sendo o comportamento do tempo de execução de cada instância. Este flutua drasticamente, atingindo desvios padrões de 250%. Algo que mostra o impacto no tempo de execução que uma instância pode ter em um problema de busca em um espaço. Estes problemas podem ter tempos de execuções extremamente rápidos ou lentos, caso a instância seja um *oulier*. Nestes casos, o melhor a se fazer é percorrer todo o espaço de busca (inviável em um problema fatorial), encontrar uma instância média, que sua solução seja representativa do caso médio do problema ou então, percorrer o máximo possível de instâncias e realizar análises estatísticas de significância. Neste trabalho tentou-se realizar algo semelhante a última opção com o tempo que o autor tinha disponível, e como foi possível notar pelo desvio padrão, um número muito maior de instâncias seria necessário para se atingir conclusões mais satisfatórias sobre o tempo de execução de uma única instância.

Em seguida, as Tabelas 3 e 4 apontam as métricas *speedup* e eficiência[11] 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 um único processo consumidor é aproximadamente 94% da do sequencial. A eficiência com 10 processos consumidores se degradou devido ao compartilhamento de recursos. Por outro lado, o comportamento observado com 2, 3, 4 e 5 consumidores é notável, com eficiências que variam entre 80% e 92% e que é fracamente escalonável entre 2, 3 e 4 consumidores e 3, 4 e 5 consumidores, com valores entre uma margem de erro de 5%. Na concepção do autor, a solução não é fortemente escalonável devido a natureza produtora-consumidora, com apenas um único produtor responsável por atender todos os consumidores. Conforme o número de consumidores aumenta, o número de solicitações aumenta, enquanto que a vazão continua a mesma. Desta forma, os consumidores passam a ficar inativos, em um estado *busy-waiting* cada vez mais frequentemente.

3.2.2. Comparação com resultados obtidos com OpenMP

A Tabela 5 apresenta os resultados de *speedup* e eficiência obtidos no trabalho anterior, que paralelizava o mesmo código utilizando uma estrutura semelhante, com segmento iterativo e recursivo, utilizando OpenMP e o mecanismo de baixo nível de criação de *tasks*. Tanto os resultados do OpenMPI quanto os do OpenMP foram obtidos comparando os tempos de execução com os mesmos resultados sequenciais (o algoritmo sequencial não foi executado outra vez). Sendo assim, é possível realizar uma comparação entre eles.

Primeiramente, é possível notar que ambos os desempenhos degradaram conforme

[11] A métrica de eficiência não considera o processo produtor, uma vez que o trabalho de busca realizado por ele é extremamente reduzido se comparado com os consumidores.

Tabela 1. Média geométrica e desvio padrão por instância.

	Processos	1		2	3	4	5	10
	Versão	Sequencial	Paralelo	Paralelo	Paralelo	Paralelo	Paralelo	Paralelo
	# Cidades							
Média geométrica instância	13	0.382	0.427	0.218	0.150	0.116	0.097	0.075
	14	1.539	1.679	0.860	0.591	0.460	0.382	0.299
	15	7.627	8.122	4.189	2.888	2.260	1.888	1.533
Desvio padrão instância	13	0.397	0.419	0.211	0.144	0.110	0.090	0.065
	14	3.838	3.995	2.010	1.354	1.028	0.834	0.583
	15	9.265	9.674	4.909	3.329	2.564	2.101	1.559
% Desvio padrão instância (x100)	13	103.981	98.148	96.722	96.117	94.874	93.433	86.666
	14	249.376	237.882	233.767	229.130	223.432	218.507	195.326
	15	121.476	119.111	117.183	115.265	113.449	111.235	101.719

o número de processos ou *threads* superou o número de núcleos físicos, conforme esperado. Em relação a mais de um processo/*thread*, o OpenMP teve um comportamento fortemente escalonável, com eficiência entorno de 90%, diferente da versão com OpenMPI que foi fracamente escalonável. O autor acredita que isso esta relacionado ao problema de vazão do processo produtor e pelo fato do mecanismo de baixo nível de criação de *tasks* do OpenMP ser extremamente otimizado para o objetivo que busca atingir, diferentemente da estratégia de consumidor-produtor implementada em alto nível pelo autor. Por outro lado, comparando a execução com apenas uma *thread*/um processo, é possível notar que a versão do OpenMPI se sai melhor. Algo esperado, já que nesta versão o processo único apenas realiza as computações necessárias, enquanto que no OpenMP, a mesma *thread* deve gerar uma lista de tarefas para depois processá-la. Esta característica provavelmente resulta em um maior número de *cache misses*, reduzindo a eficiência.

3.2.3. Comparação com resultados teóricos

Também foram realizadas predições teóricas de *speedup* máximo utilizando as leis de Amdahl e Gustafson-Barsis. Para isso, foi extraído o tempo dedicado aos trechos sequenciais utilizando a média de 10 execuções das 50 instâncias de 15 cidades. Foi considerado como trecho sequencial a inicialização do algoritmo (leitura, inicialização de matrizes e cálculo de distâncias mínimas), excluindo comunicação. Apesar da inicialização ser feita por todos os processos na versão paralela, ela não tira vantagem do número de processos para realizar a tarefa mais rápido, logo, se considerou como sendo sequencial. Abaixo segue as porcentagens de tempo dedicado a tarefas sequenciais, entre 0 e 1:

- Execução sequencial: 8.368e-7;
- Execução paralela com 5 processos: 3.472e-6.

Como esperado, devido a natureza NP-difícil do problema, o tempo dedicado aos trechos sequenciais é extremamente baixo se comparado com o tempo total. Conside-

Tabela 2. Média aritmética e desvio padrão por execução.

	Processos	1		2	3	4	5	10
	Versão	Sequencial	Paralelo	Paralelo	Paralelo	Paralelo	Paralelo	Paralelo
	# Cidades							
Média aritmética tempo total	13	24.774	27.170	13.819	9.478	7.321	6.050	4.623
	14	129.097	137.736	70.018	47.761	36.810	30.220	22.664
	15	537.870	567.930	291.044	199.377	155.082	128.659	101.341
Desvio padrão tempo total	13	0.187	0.215	0.082	0.046	0.024	0.018	0.116
	14	0.391	1.083	0.351	0.200	0.124	0.121	0.091
	15	1.835	5.112	1.445	1.096	0.847	0.458	0.651
% Desvio padrão tempo total (x100)	13	0.757	0.790	0.596	0.483	0.324	0.290	2.507
	14	0.303	0.786	0.502	0.419	0.336	0.401	0.400
	15	0.341	0.900	0.496	0.550	0.546	0.356	0.642

Tabela 3. Speedup e eficiência por instância.

	Processos	1	2	3	4	5	10
	# Cidades						
Speedup instância	13	0.895	1.750	2.547	3.283	3.952	5.069
	14	0.916	1.790	2.605	3.344	4.031	5.152
	15	0.939	1.821	2.641	3.375	4.039	4.976
Eficiência instância	13	0.895	0.875	0.849	0.821	0.790	0.507
	14	0.916	0.895	0.868	0.836	0.806	0.515
	15	0.939	0.910	0.880	0.844	0.808	0.498

rando que a rotina mais significativa do trecho sequencial resume-se em 3 laços aninhados calculando as distâncias mínimas (custo $\Theta(n^3)$), faz sentido que ela seja insignificante se comparada com a rotina que soluciona o problema. As Tabelas 6 e 7 apresentam os *speedups* máximos teóricos, mas como também já esperado, acabam não contribuindo para a discussão, com *speedups* praticamente iguais ao número de processos consumidores. Neste sentido, pensando no motivo para esses valores não terem sido atingidos, podemos pensar no *overhead* necessário para a troca de mensagens ou então na já citada redução de desempenho conforme novos consumidores são adicionados, um sinal de que possivelmente 1 único produtor não tenha vazão suficiente para atender todos os consumidores. Isto é, pelo menos não com uma implementação de alto nível. Também podemos pensar que talvez seja possível um processo realizar um trabalho desnecessário, já que a distância mínima só é atualizada quando ele recebe um novo trabalho.

3.3. Experimentos - 3 hosts

Por fim, iremos discutir os resultados observados em múltiplos *hosts*, utilizando 3 computadores do DINF. Segue os resultados obtidos (Tabelas 8 e 9).

Tabela 4. Speedup e eficiência por execução.

	Processos	1	2	3	4	5	10
	# Cidades						
Speedup tempo total	13	0.912	1.793	2.614	3.384	4.095	5.359
	14	0.937	1.844	2.703	3.507	4.272	5.696
	15	0.947	1.848	2.698	3.468	4.181	5.308
Eficiência tempo total	13	0.912	0.896	0.871	0.846	0.819	0.536
	14	0.937	0.922	0.901	0.877	0.854	0.570
	15	0.947	0.924	0.899	0.867	0.836	0.531

Tabela 5. Speedup e eficiência por execução - OpenMP.

	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

Como podemos observar, a Tabela 8 nos apresenta resultados semelhantes aos observados em ambiente pessoal, com algumas observações. Como esperado, execução em um único *host* possui melhores resultados graças a comunicação por meio de *localhost* e pelos avanços arquiteturais do processador. Entretanto, ela não escalona bem acima do número de núcleos físicos (*6 cores/12 threads*), o que o ambiente com múltiplos *hosts* possui vantagem (*12 cores/12 threads*), sendo mais rápido com 10 consumidores. Por outro lado, observando a Tabela 9, percebemos que o padrão fracamente escalonável se mantém mesmo neste ambiente, o que, novamente, pode ser um indício de a vazão de 1 único produtor não é suficiente para atender todos os consumidores. Além disso, também é possível perceber uma maior eficiência se comparado com o ambiente com um único *host* com 2, 3, 4 e 5 consumidores, o que é difícil de explicar, possivelmente a implementação da comunicação em rede é diferente da utilizada localmente e mais otimizada, já que esse é o foco do OpenMPI. Finalmente, também é possível perceber uma eficiência superior a 100% com um processo consumidor, o que faz sentido se de fato a comunicação em rede for mais rápida, o que permitiria ela tirar vantagem do fato de que até a terceira cidade, o trabalho é feito pelo produtor, que não está sendo considerado nestes cálculos. Esta vantagem não deve ser muito grande, já que ela explora apenas uma pequena porcentagem do espaço de busca, porém, alinhada com o possível ganho na comunicação, possivelmente gerou a eficiência observada.

Tabela 6. Lei de Amdahl: Speedup máximo teórico (n consumidores).

2	3	4	5	10	∞
1.999	2.999	3.999	4.999	9.999	1195029

Tabela 7. Lei de Gustafson-Barsis: Speedup máximo teórico (n consumidores).

2	3	4	5	10
1.999	2.999	3.999	4.999	9.999

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 outro lado, implementa um mecanismo produtor-consumidor em alto nível utilizando as rotinas de comunicação do OpenMPI. Esta apresentou um comportamento fracamente escalonável tanto com 1 *host* quanto com 3 *hosts*. Também argumentamos que os resultados teóricos máximos são inatingíveis, considerando *overheads* associados a troca de mensagens, mas também graças a perda de desempenho associado ao aumento do número de consumidores, o que pode indicar uma vazão insuficiente por parte do produtor.

Tabela 8. Média aritmética e desvio padrão por execução - DINF.

	Processos	1		2	3	4	5	10
	Versão	Sequencial	Paralelo	Paralelo	Paralelo	Paralelo	Paralelo	Paralelo
	# Cidades							
Média aritmética tempo total	13	29.897	31.323	15.948	10.953	8.471	6.918	4.142
	14	156.585	155.417	78.626	54.707	41.609	34.061	19.939
	15	657.006	645.328	329.756	227.268	177.402	145.578	88.867
Desvio padrão tempo total	13	0.064	0.040	0.021	0.031	0.023	0.015	0.016
	14	0.112	0.118	0.025	2.027	0.031	0.021	0.085
	15	0.499	0.429	1.683	0.152	2.005	0.148	0.048
% Desvio padrão tempo total (x100)	13	0.215	0.128	0.135	0.283	0.276	0.220	0.384
	14	0.071	0.076	0.032	3.705	0.074	0.063	0.425
	15	0.076	0.066	0.510	0.067	1.130	0.101	0.054

Tabela 9. Speedup e eficiência por execução - DINF.

	Processos	1	2	3	4	5	10
	# Cidades						
Speedup tempo total	13	0.954	1.875	2.730	3.529	4.321	7.219
	14	1.008	1.992	2.862	3.763	4.597	7.853
	15	1.018	1.992	2.891	3.703	4.513	7.393
Eficiência tempo total	13	0.954	0.937	0.910	0.882	0.864	0.722
	14	1.008	0.996	0.954	0.941	0.919	0.785
	15	1.018	0.996	0.964	0.926	0.903	0.739