

## TSP PARALELO

Gabriel Lüders - GRR20190172

<https://github.com/ludersGabriel/parallel-tsp>

<https://docs.google.com/spreadsheets/d/11RnrdB1DEmXjR7ZjJIYtn0Ku1Os5FmA51GGSj9yJzKY/edit?usp=sharing>

### O Kernel

Após análises do algoritmo fornecido pelo professor, constatei que o seguinte é o “kernel” do programa:

```
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);
            }
        }
    }
}
```

Como podemos ver, essa é a parte que resolve o TSP propriamente dito. É aqui que checamos quais cidades já foram visitadas, visitamos novas cidades e calculamos a distância mínima de viagem para o conjunto de cidades fornecido. O trecho começa checando os cortes de viabilidade: se a distância atual é maior ou igual a distância mínima já calculada, o ramo é cortado. Se a profundidade é igual ao número de cidades quer dizer que chegamos ao fim da recursão e precisamos voltar à cidade original, portanto a distância é incrementada da distância da cidade atual à origem e, se for menor que a distância mínima previamente calculada, a distância mínima é atualizada. Caso não seja a última cidade da recursão, pegamos a última cidade visitada (me) e iteramos por cada cidade procurando candidatas para a próxima visita. Para cada cidade que não está presente no caminho atual, uma chamada recursiva a tsp é executada, efetivamente visitando essa cidade. A chamada adiciona 1 à profundidade, soma à distância atual a distância da cidade sendo visitada com relação à última cidade (me) e passa o array de caminhos para frente. Ao final da execução do algoritmo, temos a distância do caminho mínimo a ser percorrido em min\_distance.

### Estratégia de Paralelização

A intuição aqui foi simples e direta: como precisamos paralelizar o TSP em si, o foco da paralelização deve ser a própria função tsp, o kernel do algoritmo. Dessa forma, a ideia foi paralelizar o for que executa as chamadas recursivas para os candidatos:

```

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);
    }
}

```

De início, tentei com *openmp parallel for*. Com um schedule dinâmico e mais algumas diretivas para privar certas variáveis e um memcpy para dar uma cópia de path para cada ramo, obtive sucesso, apesar de um speedup pequeno: cerca de 1,6 apenas.

Como achei 1,6 não muito significativo e ainda tinha outras opções para explorar antes de tentar otimizar o que eu havia feito, segui para uma implementação com tasks criadas para as chamadas recursivas até certa profundidade:

```

void tsp(int depth, int current_length, int *path) {
    int i;
    int local_min_distance;

    #pragma omp atomic read
    local_min_distance = min_distance;

    if (current_length >= local_min_distance) return;
    if (depth == nb_towns) {
        current_length += dist_to_origin[path[nb_towns - 1]];

    #pragma omp atomic read
        local_min_distance = min_distance;

        if (current_length < local_min_distance) {
    #pragma omp atomic write
            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;

                if (depth < 3) {
                    int *new_path = (int *)malloc(sizeof(int) * nb_towns);
                    memcpy(new_path, path, sizeof(int) * nb_towns);

    #pragma omp task firstprivate(new_path, depth, current_length, dist)
                    {
                        tsp(depth + 1, current_length + dist, new_path);
                        free(new_path);
                    }
                } else {

```

```

        tsp(depth + 1, current_length + dist, path);
    }
}
}
if (depth < 3) {
#pragma omp taskwait
}
}
}
}

```

Após alguns testes rápidos, optei por permitir criação de tasks apenas até a profundidade 3 da recursão. Profundidades muito grandes causavam muito overhead e tornavam o código extremamente lento. Essa foi a implementação final que utilizei para gerar os testes. A região paralela é definida na primeira chamada a `tsp`, dentro de `run_tsp`:

```

#pragma omp parallel proc_bind(spread)
{
#pragma omp single
    { tsp(1, 0, path); }
}

```

A implementação funciona da seguinte forma: nas checagens iniciais, utilizei *omp atomic read* e *omp atomic write* para ler e atualizar o caminho mínimo encontrado até o momento, visto que todos os ramos têm acesso a essa variável e, portanto, ela gera uma região crítica no código. Após, na entrada do `else` responsável por procurar as cidades candidatas, gerei uma task utilizando

```
omp task firstprivate(new_path, depth, current_length, dist)
```

para cada uma das chamadas recursivas que ocorrem antes da profundidade 3 da árvore. Privei `depth`, `current_length` e `dist` para que uma thread não impactasse noutras e fui obrigado a fazer uma cópia de `path` para `new_path` para que cada thread tivesse a sua cópia única do caminho a ser analisado. Não consegui privar essa variável via primitivas do `openmp`. Acredito que não tenha funcionado porque `path` é um ponteiro para uma região de memória e, ao privar via primitivas, eu estava apenas gerando uma cópia do ponteiro. Dessa forma toda thread tinha uma variável ponteiro única, mas todas apontavam para a mesma região de memória. O motivo disso ser um problema é que o caminho é atualizado com as cidades já visitadas por aquele ramo, porém elas podem não ser as mesmas visitadas por outro ramo. Dessa forma não queremos que atualizações de visita em um ramo alterem as possíveis cidades candidatas para outros ramos. Por fim, gerei uma barreira explícita para que todas as tasks geradas até a profundidade da criação esperassem e sincronizassem com as outras no momento da saída da função.

Por último, a região paralela é criada na primeira chamada de `tsp` com uma diretiva `single` para que apenas uma das threads spawne a função `tsp` e gere as tasks iniciais para que as outras threads auxiliem na carga de trabalho. Também utilizei `proc_bind(spread)` na tentativa de espalhar as threads da melhor maneira possível entre os processadores existentes.

É importante ressaltar que apesar do `single`, tasks criadas pela primeira thread após a primeira execução do `tsp` também podem criar novas tasks depois caso sua profundidade

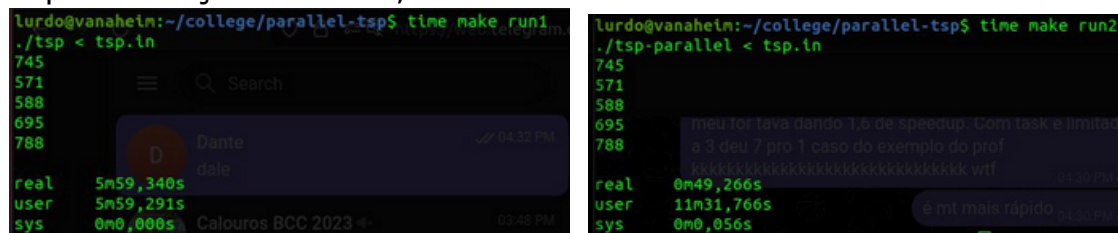
seja  $< 3$ . Vale também apontar que a criação das tasks até o ramo 3 faz com que após essa profundidade o código siga com a busca da maneira que já estava fazendo, mas agora analisando vários ramos em paralelo. A ideia era balancear o overhead de criar tasks com a maior velocidade que o código sequencial pode obter conforme mais profundo nos ramos, visto que cortes podem ocorrer com mais frequência e faltam menos nós para analisar.

Por fim, gostaria de deixar claro que a utilização de tasks aqui foi devido a natureza desbalanceada do problema. Como há cortes de ramos baseado na distância mínima já encontrada e na distância atual do caminho sendo seguido, certas threads poderiam acabar muito antes que outras dependendo da maneira que o input do problema foi apresentado. Portanto, a solução com tasks me pareceu boa já que gera um escalonamento dinâmico implícito via task queue.

## Testes

Os testes foram executados no meu notebook. O SO utilizado foi Ubuntu 20.04.6 LTS; a versão do kernel é 5.15.0-84-generic; a versão do compilador é gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0; a cpu é um AMD Ryzen 7 5800H, possui 8 cores físicos e 16 threads por padrão. Os flags de compilação foram `FLAGS=-O3 -fopenmp -lm` e estão presentes no makefile. O computador ficou em modo avião, com o mínimo possível de processos do SO rodando além do terminal que rodava o script de automação.

De início, fiz apenas testes simples para decidir como iria prosseguir com a implementação. Utilizei o input inicial fornecido pelo professor para decidir algumas coisas. Como citado acima, o parallel for me deu um speedup inicial de 1,6. Quando rodei com a nova implementação de tasks, obtive:



The image contains two side-by-side terminal screenshots. The left terminal shows the execution of a program with the command `time make run1`. The output lists a list of numbers (745, 571, 588, 695, 788) and then shows timing statistics: `real 5m59,340s`, `user 5m59,291s`, and `sys 0m0,000s`. The right terminal shows the execution of the same program with the command `time make run2`. The output lists the same list of numbers and shows timing statistics: `real 0m49,266s`, `user 11m31,766s`, and `sys 0m0,056s`. There is some text in the background of the right terminal that is partially obscured.

Metric	Sequential (run1)	Parallel (run2)
real	5m59,340s	0m49,266s
user	5m59,291s	11m31,766s
sys	0m0,000s	0m0,056s

podemos notar um speedup de  $359,34 / 49,266 = 7,2$ . Como teste inicial, fiquei bem feliz, pois isso apontava para um possível speedup quase linear, visto que a minha máquina tem 8 cores e o algoritmo paralelo estava rodando com o número default de threads.

Após, fiz uma breve análise de inputs para ver o impacto que a entrada tem no código sem alterar mais nada. Gerei três inputs: um com um caminho obviamente mínimo (uma linha reta), um caminho normal (praticamente aleatório) e um caminho horrível no qual as possíveis rotas tinham valores de `min_distance` muito parecidos (diminuindo assim os cortes por viabilidade). Todos os arquivos estão no diretório do trabalho (`tsp-best.in`, `tsp-average.in`, `tsp-worst.in`), foram feitos em cima de 17 cidades e me deram os seguintes resultados:

Sequencial

```
lurdo@vanaheim:~/college/parallel-tsp$ make run1Best
time ./tsp < tsp-best.in
320
69.51user 0.00system 1:09.52elapsed 99%CPU (0avgtext+0
```

```
lurdo@vanaheim:~/college/parallel-tsp$ make run1Average
time ./tsp < tsp-average.in
546
271.13user 0.00system 4:31.16elapsed 99%CPU (0avgtext+0
```

```
lurdo@vanaheim:~/college/parallel-tsp$ make run1Worst
time ./tsp < tsp-worst.in
409
446.72user 0.00system 7:26.47elapsed 100%CPU (0avgtext+0
```

Paralelo

```
lurdo@vanaheim:~/college/parallel-tsp$ make run2Best
time ./tsp-parallel < tsp-best.in
320
115.95user 0.00system 0:12.87elapsed 900%CPU (0avgtext+0
```

```
lurdo@vanaheim:~/college/parallel-tsp$ make run2Average
time ./tsp-parallel < tsp-average.in
546
465.19user 0.04system 0:36.04elapsed 1290%CPU (0avgtext+0
```

```
lurdo@vanaheim:~/college/parallel-tsp$ make run2Worst
time ./tsp-parallel < tsp-worst.in
409
805.96user 0.06system 0:58.74elapsed 1372%CPU (0avgtext+0
```

Como podemos ver, apenas mudar a forma como o input é fornecido altera drasticamente o tempo de execução dos algoritmos, tanto para o sequencial quanto para o paralelo. Isso se dá ao fato do algoritmo ser de complexidade fatorial e ao corte de ramos por viabilidade. Se há poucos cortes, o número de ramos a serem explorados é significativamente maior devido ao  $n!$ . Dessa forma, decidi utilizar inputs que fossem mais rápidos para ambos visto que o tempo de execução dos testes já seria enorme e a única coisa que importa é que ambos algoritmos executem em cima da mesma entrada. No mais, o enunciado do trabalho pede 20 execuções por teste, mas segui isso apenas como guideline. Devido à natureza fatorial, mudar o tamanho do input minimamente causa aumentos significativos no tempo de execução. Como exemplo, a turma como um todo encontrou que ao mudarmos de 19 para 20 cidades, o tempo variava de aproximadamente 3h para 19 cidades para 22h com 20.

Tendo decidido qual algoritmo usar e o formato das entradas, escrevi um script em shell para executar os programas várias vezes por iteração (considere iteração um par num proc x num cidades), executando o sequencial para 1 core e o paralelo para 2, 4 e 8 cores. Como acima de 19 cidades o tempo para rodar o algoritmo era muito absurdo, rodei para  $14 \leq n \leq 19$ . O número de execuções repetidas por iteração variou de acordo com a seguinte heurística:

```
if [ $(echo "$initial_time < 60" | bc) -eq 1 ]; then
    runs=20
elif [ $(echo "$initial_time >= 60 && $initial_time < 300" | bc) -eq 1 ]; then
    runs=15
elif [ $(echo "$initial_time >= 300 && $initial_time < 600" | bc) -eq 1 ]; then
    runs=10
elif [ $(echo "$initial_time >= 600 && $initial_time < 1800" | bc) -eq 1 ]; then
    runs=5
else
    runs=1
fi
```

A ideia por trás é que conforme o tempo de execução aumenta, possíveis interferências do SO tem impactos menos significativos no run time. Além disso, demoraria muito para executar 20x para todos os casos, então fui reduzindo o número de runs conforme o tempo encontrado na primeira iteração. Os tempos foram medidos utilizando a `system.h` e `time.h` para o sequencial e as chamadas `omp_get_wtime()`; para o paralelo. Além disso, como a ideia era paralelizar para o tsp em si e não para resolver múltiplos tsp em sequência (apesar de um causar redução no outro), limitei todas as entradas para apenas 1 problema. Com isso definido, utilizei o primeiro número, antes indicador do número de problemas a serem resolvidos, da entrada para decidir quantos processadores utilizar com a diretiva `omp_set_num_threads(num_threads)`; . Esse algoritmo produz 4 tabelas: uma com os tempos médios de execução, uma com os tempos apenas da parte paralela, outra com o tempo apenas da parte puramente sequencial e outra com o desvio padrão dos tempos de execução. Os tempos são sempre médias da quantidade de runs que o script executa para aquela iteração e são expressos em segundos. Tudo isso pode ser encontrado no repositório ou no link da planilha. O tempo puramente sequencial foi calculado checando quanto tempo a execução do alg sequencial levava descontando a parte do tsp que foi paralelizada. Com essas tabelas, é possível extrair speedup e eficiência com facilidade.

betas	puramente seq
Size	% seq
14	0.01252103151
15	0.003322396345
16	0.0007878171829
17	0.0002100424965
18	0.00004951118996
19	0.00001295946703

eficiência	número de cpus (seq = 1)			
Size	Seq	2	4	8
14	1	0.7583597991	0.7609632862	0.5699372469
15	1	0.7626543288	0.7344248944	0.5482265398
16	1	0.7626204477	0.682254568	0.5298231816
17	1	0.7548240903	0.6843775759	0.5165192102
18	1	0.8454148753	0.6873105005	0.5188933841
19	1	0.9231878657	0.7175393247	0.5263160813

amdahl	número de cpus (seq = 1)		
Size	2	4	8
14	1.97526761 2	3.855187237	7.35532611
15	1.993377211	3.960524702	7.81817448
16	1.99842560 6	3.990568485	7.956124201
17	1.99958000 3	3.997481077	7.988254889
18	1.99990098 3	3.999405954	7.997228334
19	1.99997408 1	3.999844492	7.999274336
avg	1.99442091 6	3.967168658	7.852397058

speedup	número de cpus (seq = 1)			
Size	Seq	2	4	8
14	1	1.516719598	3.043853145	4.559497975
15	1	1.525308658	2.937699577	4.385812318
16	1	1.525240895	2.729018272	4.238585453
17	1	1.509648181	2.737510303	4.132153681
18	1	1.690829751	2.749242002	4.151147072
19	1	1.846375731	2.870157299	4.210528651

Aqui estou demonstrando apenas as tabelas principais. As outras estão no link da planilha listado no topo deste artigo. Análises seguirão abaixo.

## Análises

O tempo puramente sequencial foi calculado, como explicado acima, utilizando a diferença entre o tempo total de execução e o tempo de execução da parte que foi paralelizada. Os cálculos foram feitos em cima do algoritmo sequencial visto que Amdahl precisa do tempo puramente sequencial de acordo com o algoritmo sequencial. Depois, peguei esse tempo e o dividi pelo tempo total de execução do sequencial e multipliquei por 100 para termos a tabela de betas. O tempo paralelo foi meramente calculado utilizando  $(1 - \text{beta})$ . Com isso, a lei de amdahl foi calculada.

Escalabilidade forte ocorre quando o algoritmo demonstra capacidade de manter a mesma eficiência conforme o número de processadores aumenta. Aqui notamos que após o quarto core a eficiência cai drasticamente, portanto não é fortemente escalável. De outra maneira, escalabilidade fraca ocorre quando demonstra capacidade de manter a mesma eficiência conforme a dimensão do problema e o número de processadores aumentam proporcionalmente. Podemos ver que com 2 cores isso ocorre, mas com 4 e 8 não. Para esses, a eficiência se mantém praticamente constante. Por consequência o algoritmo não é fracamente escalável. Dessa forma, o algoritmo da maneira presente não é escalável.

Analisando a tabela de speedups e amdahl ao mesmo tempo, notamos que para 2 e 4 cores, o speedup teórico foi razoavelmente parecido com o encontrado, mas cai drasticamente com 8 cores. Com isso, é possível determinar que os recursos não estão sendo utilizados da maneira mais eficiente. A tabela de eficiência mostra isso diretamente ao mostrar a queda significativa de 4 para 8 cores na eficiência.

Acredito que para obter escalabilidade fraca poderíamos aumentar a profundidade máxima da criação de tasks, visto que a partir da profundidade definida temos uma quantidade de ramos sequenciais sendo tratados em paralelo e essa quantidade aumenta conforme a profundidade máxima aumenta. A minha hipótese é que da maneira como implementei, o problema continua desbalanceado para um número muito grande de cidades, na qual os ramos se tornam cada vez maiores e não há extra paralelismo sendo incorporado. Dessa forma, quando a quantidade de cpus aumenta, ainda não há uma quantidade suficiente de tasks sendo geradas para acarretar no balanceamento entre as threads.

Da maneira como o código está, os resultados aparentam ser esperados. É muito difícil saber a quantidade de tasks a serem criadas quando não sabemos quantas chamadas recursivas podem ocorrer já que, como demonstrado acima, o input muda drasticamente a quantidade de ramos sendo processados. Testes com o caso médio e o pior caso deveriam ser feitos também para que uma profundidade máxima ideal pudesse ser encontrada para que atingamos o balanceamento do número de tasks x número de processadores sem que o overhead seja significativo a ponto de afetar o tempo de execução.

Aqui vale também discutir sobre hyperthreading e a diferença de speedup mostrada acima. Comentei que fiquei bem feliz quando encontrei speedups  $\geq 7$  rodando no default da minha máquina. Acontece que o meu processador possui 8 cores físicos, porém 16 threads, o que significa que 8 dessas são "virtuais". Assim, o default da minha máquina é 16 threads e foi isso que constatei após análises de uso de threads com a ferramenta htop. Ao rodar o programa no default, eu obtinha um uso de 1600% de cpu aproximadamente. Quando



rodava com 8 fixas e settadas para spread, o uso de cpu mal passava de 900%, pouco mais da metade apresentada no default. Além disso, a documentação do openmp diz que durante o runtime, quando apenas `proc_bind(spread)` é utilizado, o SO pode mudar as threads de lugar da maneira que bem entender. Pude notar isso também no htop. Quando o número de threads era fixo, elas ficavam pulando entre os núcleos mesmo sem nenhum outro processo significativo rodando. Essas mudanças que o SO faz com certeza trazem consigo certo overhead, o que ajuda a justificar a perda de eficiência.

Tentei fixar as threads em núcleos específicos utilizando affinity, mas os resultados foram muito similares, não obtendo nenhum ganho significativo. O uso de cpu ficava entre 800 ~ 900% e em alguns casos notei pioras no speedup, inclusive.

No mais, para testes realmente conclusivos sobre a escalabilidade do meu algoritmo, hyperthreading deveria ter sido desligado por completo para que com toda certeza uma thread no núcleo utilize 100% daquele núcleo quando possível. Como cada núcleo possui 2 threads, processos em background do sistema podem estar utilizando tais threads já que elas estão livres ou o SO pode apenas não estar liberando todo o núcleo já que outros processos podem vir a querer utilizar a outra thread livre. Como obtive speedups muito maiores com as 16 threads, penso que as 8 threads realmente não estavam utilizando os cores por completo. Porém, infelizmente a opção para desativar hyperthreading não existia na bios da minha máquina, provavelmente devido a falta de atualizações ou a impossibilidade de fazer isso em máquinas AMD, não encontrei nada conclusivo sobre.

## **Conclusão**

Apesar do algoritmo implementado não ser escalável de primeira mão, acredito ter atingido todos os objetivos propostos pelo trabalho. Entendi o que deveria ser paralelizado e como paralelizar tal parte. Executei testes com mais de uma forma antes de decidir por qual seguir, demonstrando conhecimento do conteúdo e fiz as minhas análises com o rigor necessário, no tempo provido, para saber se a solução encontrada é boa ou não e o que afeta a velocidade do problema apresentado, além de ter encontrado possíveis respostas para a baixa eficiência com alto número de cores, como má utilização da cpu devido a hyperthreading e o desbalanceamento do problema não estar sendo contemplado de maneira ótima devido ao número de tasks sendo criadas.

No mais, acredito que para análises mais precisas no curto período de tempo que temos para resolver trabalhos na faculdade, um problema de complexidade menor teria sido mais interessante. Com complexidades polinomiais, quadráticas ou cúbicas, por exemplo, seria possível realizar testes num range muito maior do tamanho da entrada, o que diminuiria possíveis fontes de erro. Como os testes seriam mais rápidos também seria possível testar mais possibilidades de paralelismo ou até mesmo tentar apertar parafusos, como encontrar o valor ideal da profundidade máxima para geração de tasks ou procurar outras formas de garantir o balanceamento da carga.