UNIVERSIDADE ESTADUAL DO OESTE DO PARANÁ
Curso de Bacharelado em Ciência da Computação
UNIOESTE - *Campus* de Cascavel

Análise de paralelização de um algoritmo Dijkstra utilizando o OpenMP

Roberval Requião Junior

1. Introdução

O problema escolhido para este trabalho foi o do Dijkstra, utilizado na 9ª Maratona de Programação Paralela, de 2014. A função de um algoritmo Dijkstra é encontrar o caminho de menor custo entre os nós de um grafo, que é definido como uma representação abstrata de um conjunto de objetos e das relações existentes entre eles. Em um grafo, os objetos são representados como nós e as suas ligações são representadas como arestas, que ligam pares de nós.

2. Funcionamento do programa

O código disponibilizado na maratona realiza a criação de um grafo de acordo com os parâmetros de entrada, calculando após isso a distância média entre o nó inicial e todos os outros nós.

A execução do programa demanda três entradas, onde a primeira se refere ao número de nós do grafo, a segunda ao número médio de saídas dos nós e a terceira a um número inteiro qualquer, que é utilizado como uma *seed* durante a geração das conexões dos nós, de forma a permitir que as mesmas entradas em diferentes execuções gerem sempre os mesmos grafos e, consequentemente, as mesmas saídas.

2. Potencial de paralelização

Para a identificação dos trechos com potencial de paralelização do algoritmo, foi realizada uma análise do seu funcionamento, buscando identificar laços de repetição que ocupassem boa parte do tempo de execução total do programa.

De forma visual, foram identificados quatro laços de repetição que poderiam ser relevantes para a paralelização, sendo dois laços for na função de criação do grafo e dois laços for na função de busca do Dijkstra, com um deles possuindo mais dois laços em seu interior. Após esta identificação visual, foram aplicadas funções para o cálculo do tempo gasto por cada laço, onde foi possível confirmar que estes são os trechos que mais ocupam o tempo de execução do programa, ocupando cerca de 99% do seu tempo de execução total. A imagem abaixo mostra os resultados de um dos testes realizados, imprimindo na tela o tempo gasto por cada ponto calculado.

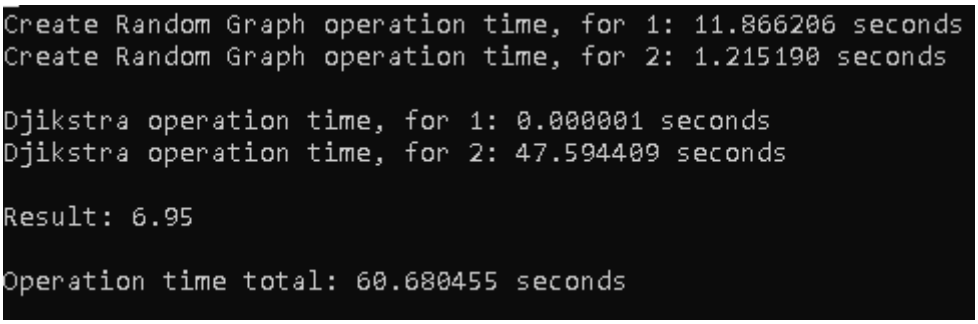


Figura 1 – Tempo de execução dos trechos analisados

Pelos resultados obtidos, observou-se que a operação mais relevante foi o segundo laço da função de busca do Dijkstra, ocupando cerca de 78% do tempo de execução do programa, enquanto o seu primeiro laço ocupou um tempo muito pequeno. Além disso, a segunda operação mais relevante foi a do primeiro laço da função de criação do grafo, ocupando cerca de 19.5%, enquanto o seu segundo laço ocupou cerca de 0.2%.

3. Implementação da paralelização

Após a identificação dos trechos com potencial de paralelização, iniciou-se a etapa de testes, com tentativas de paralelizar os trechos identificados e observar os seus resultados. Os testes foram realizados na ordem em que os trechos aparecem no código, iniciando-se então pelo primeiro laço da função de criação do grafo.

```
int k, v;
#pragma omp parallel for firstprivate(nNodes)
for (v = 0; v < nNodes; v++) {
                graph->edges[v] = (int *) malloc(sizeof(int) * nNodes);
                graph->w[v] = (int *) malloc(sizeof(int) * nNodes);
                graph->nEdges[v] = 0;
}
```

Figura 2 – Laço 1 da função de criação do grafo

Este laço tem por finalidade alocar dinamicamente os espaços em memória do grafo, definido como uma struct, conforme a imagem abaixo:

```
struct Graph {
        int nNodes;
        int *nEdges;
        int **edges;
        int **w;
};
```

Figura 3 – Struct do grafo

A tentativa foi feita aplicando um *pragma omp parallel for* com apenas um parâmetro na cláusula *firstprivate,* que é o único parâmetro utilizado na função que não foi criado como um ponteiro, necessitando então ser inicializado antes de entrar no processo. A aplicação desta função não prejudicou os resultados do programa, porém não foi vantajosa em questões de tempo de execução, com o código sequencial sendo mais rápido que o paralelo.

Após isso, foi tentado paralelizar o segundo laço desta função, que é responsável por gerar o grafo. Foi possível paralelizá-lo a partir do seu laço interno, definindo uma seção critica para duas variáveis que causavam concorrência entre as threads:

```
int source = 0;
for (source = 0; source < nNodes; source++) {
        int nArestasVertice = (double) nEdges / nNodes * (0.5 + my_rand() / (double) RAND_MAX);
        #pragma omp parallel for firstprivate(source) private(k) shared(nNodes)
        for (k = nArestasVertice; k >= 0; k--) {
                int dest, w;
                #pragma omp critical
                {
                        dest = my_rand() % nNodes;
                        w = 1 + (my_rand() % 10);
                }
                graph->edges[source][graph->nEdges[source]] = dest;
                graph->w[source][graph->nEdges[source]++] = w;
        }
}
```

Figura 4 – Laço 2 da função de criação do grafo

Porém, observou-se que a utilização de paralelismo nessa função acaba resultando na geração de grafos totalmente aleatórios em todas as vezes, quebrando um dos objetivos do programa que é o de, com as mesmas entradas em diferentes execuções, gerar os mesmos grafos e, portanto, as mesmas saídas. Isso ocorreu devido ao fato de que a seed precisa ser calculada e incrementada sequencialmente dentro desse laço de repetição, chamando em toda a rodada, as duas funções abaixo:

```
int my_rand(void) {
        return ((next = next * 1103515245 + 12345) % ((u_long) RAND_MAX + 1));
}

void my_srand(unsigned int seed) {
        next = seed;
}
```

Figura 5 – Funções de calculo da seed

O próximo laço analisado foi o primeiro laço da função de busca do Dijkstra, que armazena as distâncias entre os nodos. Porém, além deste laço ocupar muito pouco tempo de execução do programa, a tentativa de paralelizá-lo resultou em um tempo maior de execução do que o código sequencial.

```
for (k = 0; k < graph->nEdges[source]; k++)
        distances[graph->edges[source][k]] = graph->w[source][k];
```

Figura 6 – Laço 1 da função de busca do Dijkstra

Por fim, analisou-se o segundo laço desta função, que tem como objetivo realizar toda a busca para encontrar as menores distâncias entre o nó principal e os demais. Foi possível paralelizá-lo a partir dos seus laços internos, que trouxeram melhores resultados. Para o primeiro *sub laço,* precisou-se apenas de dois parâmetros na clausula *shared,* pois os demais valores utilizados dentro dele são todos referentes à ponteiros. O segundo laço de nenhum parâmetro, pois todos os valores trabalhados dentro dele ou foram definidos internamente ou são ponteiros.

```
for (v = 1; v < nNodes; v++) {
        int min = 0;
        int minValue = INT_MAX;
        #pragma omp parallel for shared(minValue,min)
        for (k = 0; k < nNodes; k++)
                if (visited[k] == 0 && distances[k] < minValue) {
                        minValue = distances[k];
                        min = k;
        }

        visited[min] = 1;

        #pragma omp parallel for
        for (k = 0; k < graph->nEdges[min]; k++) {
                int dest = graph->edges[min][k];
                if (distances[dest] > distances[min] + graph->w[min][k])
                        distances[dest] = distances[min] + graph->w[min][k];
        }
}
```

Figura 7 – Laço 1 da função de busca do Dijkstra

3. Testes no Santos Dumont

Após realizada a implementação da paralelização, foram realizados os testes de execução no Supercomputador Santos Dumont. O programa foi executado com os parâmetros de entrada “500000 100 1”, ou seja, 500mil nós, 100 arestas saindo por nó e seed de valor 1. Esta entrada, no código sequencial, gerou um tempo total de execução de 657.84 segundos, ou seja, de quase 11 minutos.

```
Create Random Graph operation time, for 1: 9.465226 seconds
Create Random Graph operation time, for 2: 0.876481 seconds

Djikstra operation time, for 1: 0.000001 seconds
Djikstra operation time, for 2: 647.495447 seconds

Result: 6.95

Operation time total: 657.841107 seconds
```

Figura 8 – Execução do código sequencial

Após isso, executou-se o código paralelizado, com as mesmas entradas, para 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22 e 24 threads, conforme os dados abaixo:

```
Create Random Graph operation time, for 1: 11.142503 seconds
Create Random Graph operation time, for 2: 1.185742 seconds

Djikstra operation time, for 1: 0.000001 seconds
Djikstra operation time, for 2: 341.362952 seconds

Result: 6.95

Operation time total: 353.695836 seconds

Threads: 2
```

```
Create Random Graph operation time, for 1: 11.145890 seconds
Create Random Graph operation time, for 2: 1.241827 seconds

Djikstra operation time, for 1: 0.000001 seconds
Djikstra operation time, for 2: 173.352815 seconds

Result: 6.95

Operation time total: 185.745445 seconds

Threads: 4
```

```
Create Random Graph operation time, for 1: 11.420238 seconds
Create Random Graph operation time, for 2: 1.205561 seconds

Djikstra operation time, for 1: 0.000001 seconds
Djikstra operation time, for 2: 121.993638 seconds

Result: 6.95

Operation time total: 134.624422 seconds

Threads: 6
```

```
Create Random Graph operation time, for 1: 11.462175 seconds
Create Random Graph operation time, for 2: 1.196127 seconds

Djikstra operation time, for 1: 0.000001 seconds
Djikstra operation time, for 2: 95.563437 seconds

Result: 6.95

Operation time total: 108.226358 seconds

Threads: 8
```

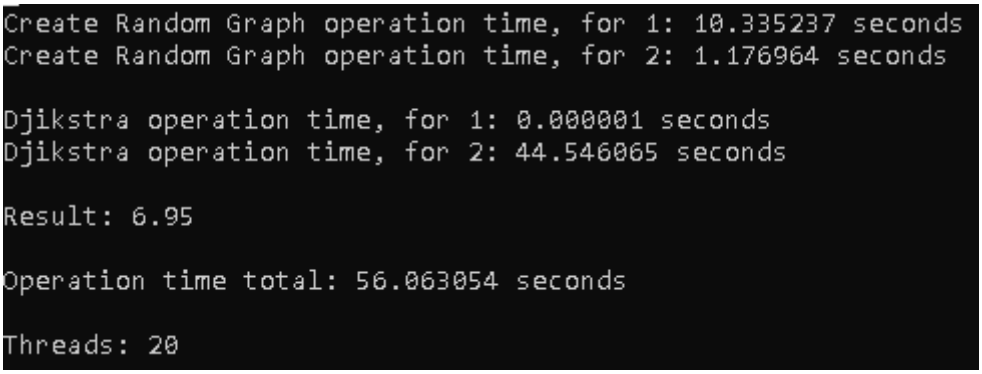
```
Create Random Graph operation time, for 1: 11.454898 seconds
Create Random Graph operation time, for 2: 1.235024 seconds

Djikstra operation time, for 1: 0.000001 seconds
Djikstra operation time, for 2: 79.611127 seconds

Result: 6.95

Operation time total: 92.305917 seconds

Threads: 10
```

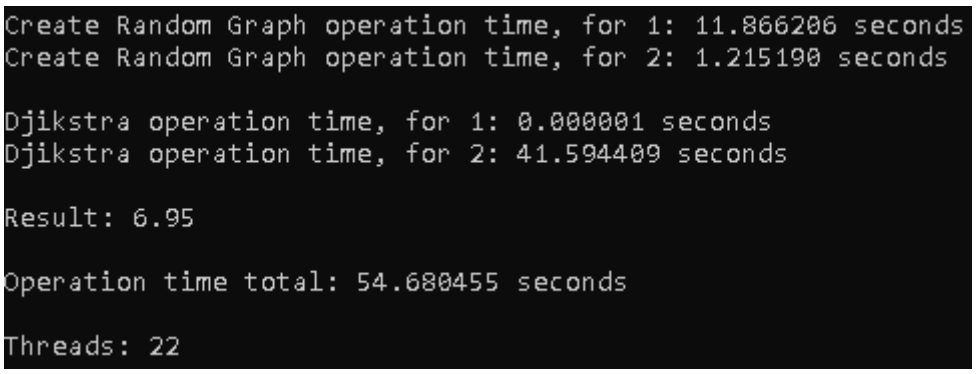
```
Create Random Graph operation time, for 1: 11.402495 seconds
Create Random Graph operation time, for 2: 1.196361 seconds

Djikstra operation time, for 1: 0.000001 seconds
Djikstra operation time, for 2: 67.913080 seconds

Result: 6.95

Operation time total: 80.516780 seconds

Threads: 12
```

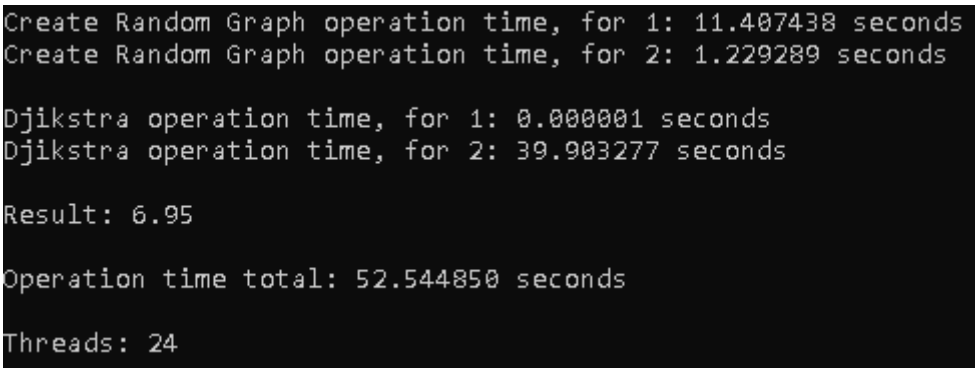
```
Create Random Graph operation time, for 1: 11.313476 seconds
Create Random Graph operation time, for 2: 1.231393 seconds

Djikstra operation time, for 1: 0.000001 seconds
Djikstra operation time, for 2: 59.649118 seconds

Result: 6.95

Operation time total: 72.199032 seconds

Threads: 14
```

```
Create Random Graph operation time, for 1: 11.554907 seconds
Create Random Graph operation time, for 2: 1.218270 seconds

Djikstra operation time, for 1: 0.000001 seconds
Djikstra operation time, for 2: 53.208380 seconds

Result: 6.95

Operation time total: 65.986586 seconds

Threads: 16
```

```
Create Random Graph operation time, for 1: 11.283801 seconds
Create Random Graph operation time, for 2: 1.179018 seconds

Djikstra operation time, for 1: 0.000001 seconds
Djikstra operation time, for 2: 48.765772 seconds

Result: 6.95

Operation time total: 61.233323 seconds

Threads: 18
```







Figuras 9 – Execução do código paralelo

4. Resultados

Após os testes serem realizados, os resultados foram centralizados em uma planilha, separando os dados de acordo com o número de threads. Também foi incluído na planilha o cálculo do SpeedUp, que tem a finalidade de medir o desempenho relativo de dois sistemas processando o mesmo problema. Este calculo é feito dividindo o tempo de execução do código paralelo pelo tempo de execução do código sequencial, obtendo-se assim uma relação de qual foi a taxa de ganho de tempo de execução. Além disso, foi realizado o cálculo da Eficiência, que é uma medida do grau de aproveitamento dos recursos computacionais, medindo a razão entre o grau de desempenho e os recursos computacionais disponíveis. O

cálculo da eficiência se dá pela divisão do SpeedUp com o número de threads utilizadas.

Threads	1	2	4	6	8	10	12	14	16	18	20	22	24
Tempo	657,84	353,7	185,75	134,62	108,23	92,31	80,52	72,2	65,99	61,23	56,06	54,68	52,54
SpeedUp	1	1,8599	3,5415	4,8866	6,0782	7,126	8,17	9,111	9,969	10,74	11,73	12,03	12,52
Eficiência	1	0,9299	0,8854	0,8144	0,7598	0,713	0,681	0,651	0,623	0,597	0,587	0,547	0,522

Tabela 1 – Tempo, SpeedUp e Eficiência

Abaixo estão os dados em formato de gráfico:

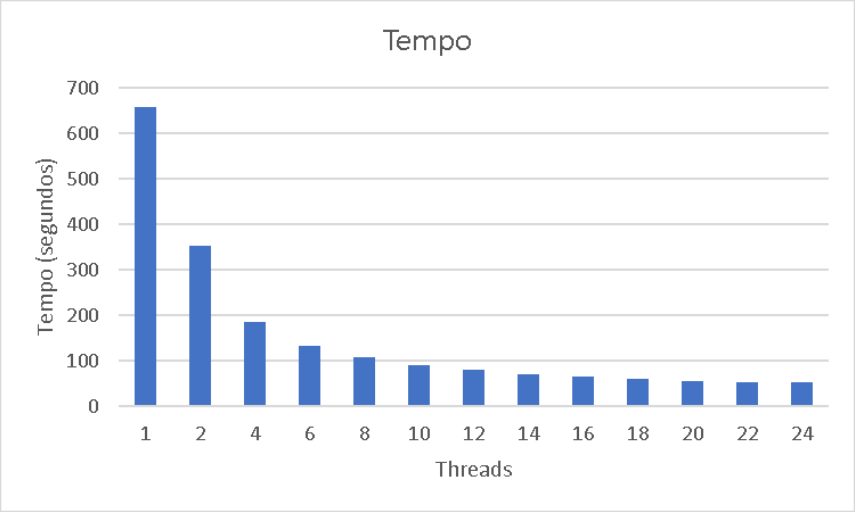


Gráfico 1 – Tempo de execução

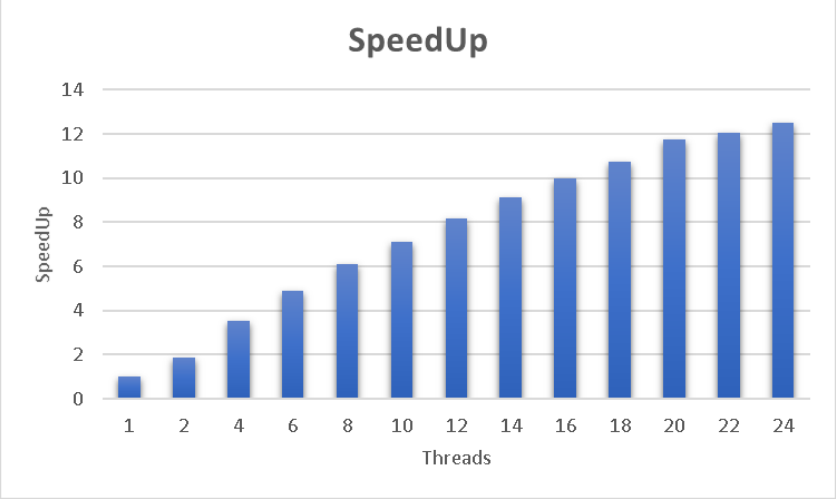


Gráfico 2 – SpeedUp

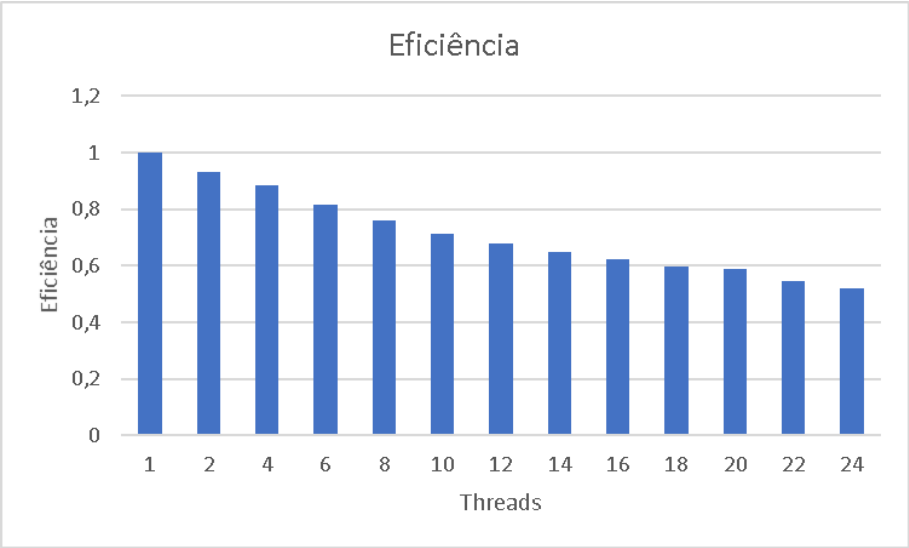


Gráfico 3 – Eficiência

5. Conclusões

Após a paralelização do algoritmo do Dijkstra da 9ª Maratona de Programação Paralela de 2014, pode-se perceber um grande ganho em tempo de execução, principalmente nas primeiras variações do número de threads, onde com 2 threads o tempo caiu quase que pela metade.

Porém, observou-se que, conforme o número de threads vai aumentando, o algoritmo não consegue manter a sua eficiência em relação ao uso das threads, onde, com 24 threads, obteve-se uma eficiência de 0.522, ou seja, um aproveitamento de quase que apenas metade do poder computacional das threads. Isso indica que o algoritmo trabalhado possui uma escalabilidade forte, ou seja, não possui a capacidade de manter a eficiência conforme o número de threads aumenta.

Apesar disso, os resultados se mostraram muito interessantes durante o uso de até 8 threads, diminuindo um tempo considerável no processamento do código e ainda apresentando uma eficiência aproveitável.