Análise da implementação paralela do algoritmo Floyd Warshall

Diogo Alves de Almeida RA: 95108[1], **Narcizo Gabriel Freitas Palioto RA: 95380**[1], **Leonardo Puglia Assunção RA: 94518**[1]

[1]Centro de Tecnologia
Departamento de Informática – Universidade Estadual de Maringá (UEM)
CEP 87020-900 – Maringá – Paraná – Brasil

1. Introdução

O algoritmo de *Floyd-Warshall* tem o propósito de resolver o problema de caminho mínimo entre todos os nós de um grafo [Cormen et al. 2009]. O problema consiste em achar a menor distância entre todos os pares de vértices sendo que todos as arestas contem pesos e são direcionadas.

As métricas têm grande importância para a análise da implementação de um algoritmo na sua versão paralela, pois assim pode-se avaliar quão eficiente é esta versão. Para este trabalho, utilizaremos métricas básicas, tais como o speedup da aplicação, a eficiência do processador e a métrica de Karp-Flatt que normalmente possibilita diagnosticar porquê a aplicação obteve o speedup alcançado, sendo overhead um dos problemas mais comuns para esse tipo de implementação, portando, também iremos avaliá-lo.

O algoritmo de *Floyd-Warshall* é paralelizável e esse é o foco do trabalho, implementar uma versão clássica do algoritmo e uma versão paralela utilizando a biblioteca padrão da IEEE *pthreads*.

2. Referencial Teórico

Um processo tradicional possui um fluxo de execução único. No modelo *multithreads*, em cada processo pode haver diversos fluxos de execução ou *threads* [Lamport 1979]. As diversas *threads* de um processo compartilham o espaço de endereçamento, mas apresentam contextos de execução diferentes.

Como foi utilizado a divisão de cargas do problemas em diferentes *threads* o problema de sincronização deve ser considerado pois é um fator muito importante para a correta execução do algoritmo. Para resolver esse foi problema foi empregado o uso de barreiras. Uma barreira é um mecanismo de sincronização que determina um ponto na execução de uma aplicação onde vários processos ou *threads* esperam uns pelos outros e habilita o programador a sincronizar os dados da aplicação com facilidade [Solihin 2015].

3. Desenvolvimento

O desenvolvimento do algoritmo sequencial foi bem direto sendo necessário apenas três laços de repetição, os dois laços interiores para percorrer os vértices e o laço externo para verificar o caminho dos vértices intermediários. Nesse caso foi utilizado somente o grafo *path* para a realização da computação.

```
1 void fw(){
2     for(int k = 0; k < size; k++){
3         for(int i = 0; i < size; i++)
4             for(int j = 0; j < size; j++)
5                 if(path[i][k] + path[k][j] < path[i][j])
6                     path[i][j] = path[i][k] + path[k][j];
7     }
8 }
```

No caso do desenvolvimento da função paralela foram necessários alguns ajustes em relação ao algoritmo sequencial. A parte que mais difere entre as duas é a utilização de duas matrizes para a realização do algoritmo, uma contendo os caminhos mais curtos atualizados, *graph*, e a outra contendo os valores da iteração anterior, *path*. Isso é necessário pois como é um algoritmo paralelo não se há certeza de qual informação estará disponível em determinado momento e mesmo que ela esteja disponível o seu valor pode estar incorreto. Com uma segunda matriz guardando o estado anterior é possível fazer todas as leituras de dados simultaneamente e não ter que se preocupar com dados incorretos sendo computados.

Em relação a codificação foi criado um vetor de *threads* do tamanho do número de *threads* escolhido pelo usuário e a função *fwParallel* é chamada uma vez para cada elemento do vetor e é passada para a função o ID de cada *threads*.

Já dentro da função paralela a primeira linha que se difere em relação a versão sequencial é o laço que cuida das linhas da matriz, linha 6 da Função Paralela. Esse laço garante que cada *threads* compute uma linha da matriz sem outras *threads* usarem essa mesma linha para escrita. Dentro dos laços é feita a atribuição do menor caminho encontrado nessa interação para a matriz de estado atual *graph* e após isso vem a parte de sincronia que é crucial para o funcionamento de um algoritmo paralelo. Nesse estágio é utilizado uma barreira na qual a sua função é apenas barrar todos as *threads* naquela linha e quando todas atingirem esse ponto o fluxo de execução é liberado novamente. Isso é necessário pois logo em seguida ocorre a troca de ponteiros dos grafos *path* e *graph* e isso só pode ser feito quando todas as *threads* acabarem a execução para que não haja nenhum nenhuma leitura e escrita de dados incorretos. Após essa troca de ponteiros, que é realizada para atualizar a matriz de estado atual e a matriz de estado anterior, é necessária outra barreira pois, novamente, se uma *threads* começasse a próxima iteração antes da atualização das matrizes ela poderia computar dados errados.

Após o término da função *fwParallel* é feito um *join* para esperar todas as *threads* acabarem a execução para poder seguir com o programa.

```
1 void *fwParallel(void *args){
2     int tid = (int)args;
3     float **aux;
4
5     for(int k = 0; k < size; k++){
6         for(int i = tid; i < size; i+=nthreads)
7             for(int j = 0; j < size; j++)
8                 graph[i][j] = MIN(path[i][k] + path[k][j], path[
                      i][j]);
```

```
9           pthread_barrier_wait(&barrier);
10          if(tid == 0){
11              aux = graph;
12              graph = path;
13              path = aux;
14          }
15          pthread_barrier_wait(&barrier);
16      }
17  }
```

4. Ambiente experimental e experimentos realizados

4.1. Hardware e ambiente de desenvolvimento

- Sistema Operacional Linux Mint ”Tara”19 baseado no Ubuntu 16.04;
- Processador Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 1800 Mhz, 4 Núcleo(s) físicos, 8 Processador(es) Lógico(s);
- Notebook Dell Inspiron 7000, 8GB RAM, sempre conectado à tomada para evitar configuração de economia de energia.

4.2. Experimentos realizados

Ao total a aplicação foi executada 88 vezes, a fim de eliminar alta variância na extração dos resultados, ou seja, eliminar variância de tempo de execução. Os experimentos foram realizados seguindo a seguinte metodologia a fim de extrair seu tempo de execução:

- Foram considerados dois tamanhos de entrada, 3500 vértices e 5000 vértices;
- Foram rodados ao todo, 4 configurações;
    1. Aplicação sequencial;
    2. Aplicação paralela com 2 threads;
    3. Aplicação paralela com 4 threads;
    4. Aplicação paralela com 8 threads.
- Para cada configuração, incluindo a aplicação sequencia, foram executadas 11 vezes;
- A primeira execução de cada configuração foi descartada sendo considerada ”aquecimento da cache”;
- O tempo de execução considerado de cada configuração é uma média aritmética simples das outras 10 execuções.

5. Análise e discussão dos resultados

Para fins de análise, será estudado principalmente, o speedup da aplicação com a quantidade de threads disponíveis, a eficiência de cada configuração e o overhead de cada configuração, também será feito um estudo utilizando a métrica de Karp-Flatt.

5.1. Desempenho

Após a execução para dois tamanhos de entradas fixos, 3500 vértices e 5000 vértices, teve-se para cada configuração, a média do tempo de execução mostrada na Figura 1.

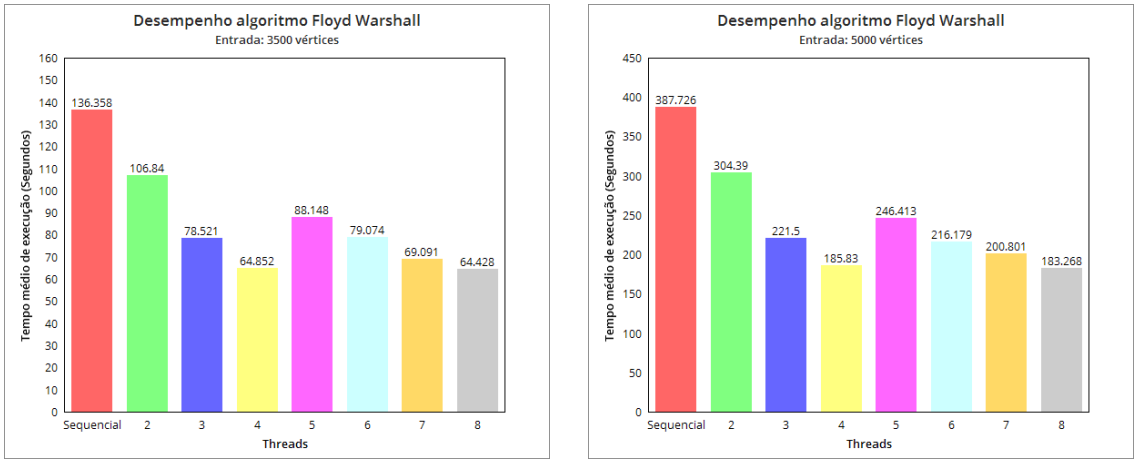

Figura 1. Tempo médio de execução para entradas de 3500 e 5000 vertices.

5.2. Speedup

Como previsto, a medida que aumentamos a quantidade de processadores, houve um aumento de speedup também. Porém, ao utilizar mais que 4 threads, lembrando que o processador utilizado possui 4 núcleos e com hyper-thread possibilitando uso de 8 threads, o speedup decai consideravelmente para ambos os tamanhos de entrada.

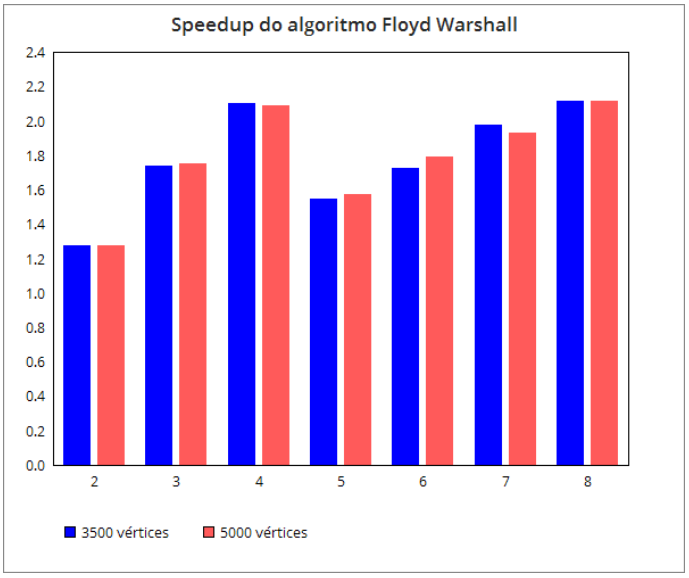

Figura 2. Gráfico de speedup.

A Figura 2 mostra os speedups para todas as instâncias executadas, nota-se um novo comportamento a partir de 5 threads que pode ter ocorrido basicamente por 2 motivos, aumento do overhead seja ele criado pela arquitetura ou pelo algoritmo, ou pela divisão de tarefas para cada threads quando excedido os limites físicos do processador, ou seja, pode acontecer de um núcleo ficar responsável por mais threads que, somadas precisam processar mais de uma linha da matriz, acarretando então num desbalanceamento de cargas para os núcleos.

5.3. Eficiência e Overhead

Segundo [Kumar 2002] um sistema paralelo ideal, para p processadores pode ser entregue um speedup igual a p. Porém o processador ainda sofre com ociosidade, tempo dedicado à comunicação entre outras atividades, para ilustrar isso, temos na figura 3 a eficiência do processador. Sendo o cálculo feito pela razão entre o speedup e a quantidade de threads utilizada.

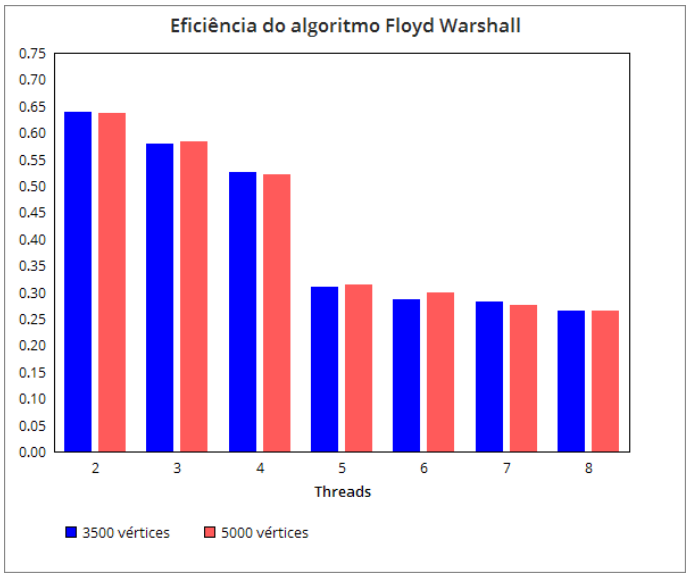

Figura 3. Gráfico de eficiência.

Com a criação de novos processos, é esperado também o aumento de overhead, uma vez que o overhead é diretamente proporcional ao número de threads/processos criados, sendo assim a soma de todo o tempo gasto pelos processos paralelos, assim, tem-se o gráfico abaixo, evidenciando que a proporção de overheads criados para cada configuração é mais atenuado de acordo com o tamanho da entrada do problema.

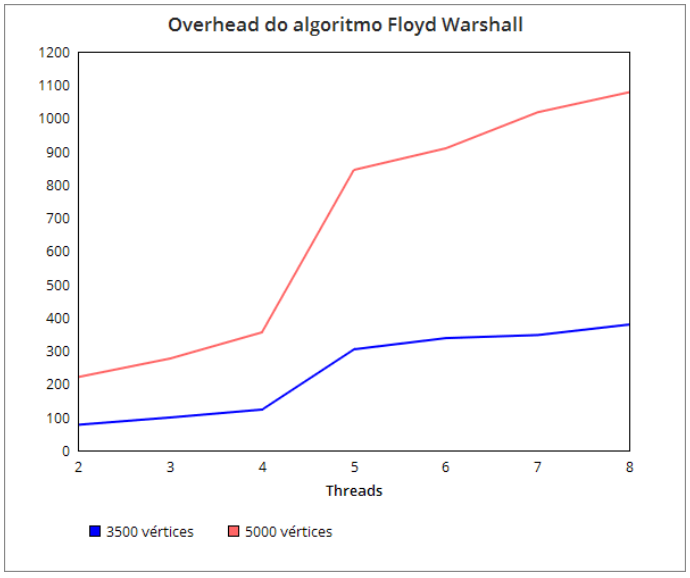

Figura 4. Gráfico do overhead.

5.4. Métrica de Karp-Flatt

Para aplicarmos a métrica de Karp Flatt, é necessário calcularmos a fração serial da aplicação, que é dada por:

$$e = \frac{1/S(p) - 1/p}{1 - 1/p}$$

Onde *S(p)* é o speedup alcançado com *p* threads. E segundo [Karp and Flatt 1990], espera-se dois possíveis comportamentos para os valores obtidos da fração serial. Sendo esses comportamentos, indicadores para diagnosticar o porquê da eficiência ter caído de acordo com o aumento das threads.

Para os casos em que a fração serial mantém-se contínua para qualquer quantidade de threads, indica-se que a eficiência caiu por motivos do paralelismo na aplicação ser determinado e não progredir. Para quando a fração serial aumenta de acordo com o número de threads, as literaturas indicam que o problema de eficiência existe pelo aumento significativo de overheads, sejam eles de comunicação, sincronização, arquiteturais entre outros.

No nosso experimento, obtivemos um resultado, inesperado de acordo com a literatura, como pode-se observar na figura 5, houve um decremento da fração serial enquanto aumentando a quantidade de threads em um único núcleo, e o mesmo comportamento para quando aumenta-se a quantidade de threads de 5 (pelo menos um núcleo responsável por mais de uma thread) até a utilização de 8 threads.

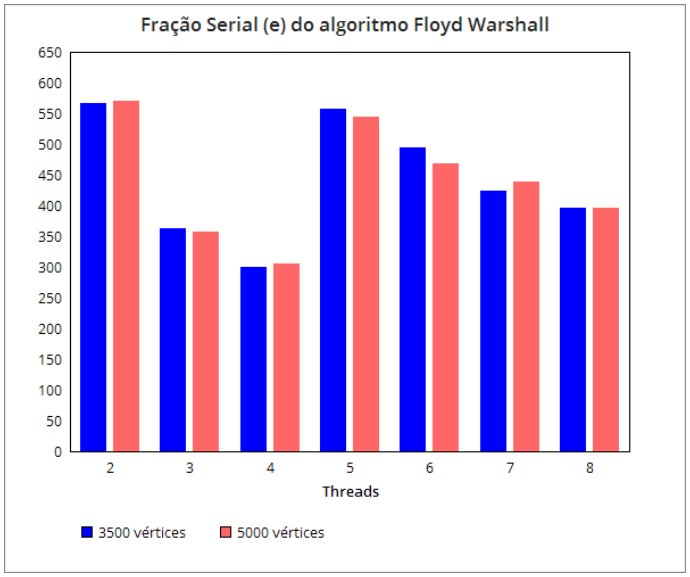

Figura 5. Gráfico da fração serial.

Este comportamento, pode indicar um paralelismo na aplicação quase que ideal, porém, sabe-se que existem partes sequenciais na aplicação que não justificam tal resul-

tado, não sendo possível então, justificar com exatidão tal comportamento pois podem ter problemas relacionados a sistema operacional a até problemas com hardware.

6. Conclusão

Após os estudos realizados chegamos a conclusão que houve um speedup desejado enquanto trabalhávamos com núcleos físicos, porém após os núcleos trabalharem individualmente houve uma diminuição do speedup, que pode ser relacionada pela diminuição do clock do processador, ao aumento de overhead ou até mesmo pelo sincronismo da memória cache.

Já para o comportamento da fração serial, fica para estudos futuros a explicação da mesma, pois na literatura a fração serial pode ser diagnosticada quando ela é constante ou crescente, o que não ocorreu no nosso estudo e tal estudo não cabe a este trabalho.

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.

Karp, A. H. and Flatt, H. P. (1990). Measuring parallel processor performance. *Commun. ACM*, 33(5):539–543.

Kumar, V. (2002). *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28*, 9:690–691.

Solihin, Y. (2015). *Fundamentals of Parallel Multicore Architecture*. Chapman & Hall/-CRC, 1st edition.