Análise da implementação paralela do algoritmo Floyd Warshall utilizando MPI

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 *MPI*.

2. Referencial Teórico

Segundo [Gropp et al. 1999], [Foster 1995] e [Pacheco et al. 1998], o MPI é um padrão de interface para a troca de mensagens em máquinas paralelas com memória distribuída e não se devem confundi-lo com um compilador ou um produto específico. Sua aplicação é constituída por um ou mais processos que se comunicam e trocam mensagens entre si. Nas implementações, inicialmente tem-se a criação de um conjunto fixos de processos, onde esses processos podem executar diferentes programas. Por isso, o padrão MPI é algumas vezes referido como MPMD (*multiple program multiple data*).

De acordo com [Ignácio and Ferreira Filho 2002], elementos importantes em implementações paralelas são a comunicação de dados entre processos paralelos e o balanceamento da carga. Os processos podem usar mecanismos de comunicação ponto a ponto (operações para enviar mensagens de um determinado processo a outro). Um grupo de processos pode invocar operações coletivas (*collective*) de comunicação para executar operações globais. O MPI é capaz de suportar comunicação assíncrona e programação modular, através de mecanismos de comunicadores (*communicator*) que permitem ao usuário MPI definir módulos que encapsulem estruturas de comunicação interna.

Os algoritmos que criam um processo para cada processador podem ser implementados, diretamente, utilizando-se comunicação ponto a ponto ou coletivas. Os algoritmos que implementam a criação de tarefas dinâmicas ou que garantem a execução

concorrente de muitas tarefas, num único processador, precisam de um refinamento nas implementações com o MPI [Ignácio and Ferreira Filho 2002].

Entender que processos em um modelo de passagem de mensagem não compartilham a mesma memória é crucial para o desenvolvimento de uma aplicação usando esse tipo de recurso, saber quando os processos precisam atualizar seus dados e/ou mandar dados para outros processos se atualizarem faz parte do procedimento e foi a parte que mais foi levada em conta nesse trabalho. Para realizar essas trocas de mensagem foi empregado o uso de *broadcast* que age como um *send/receive* junto, um processo manda dados para todos os outros processos existentes e esses processos recebem esses dados facilitando assim as trocas de mensagens e a leitura do código.

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 }
```

A versão paralela com MPI difere muito da versão implementada com a biblioteca *pthread* pois uma se baseia em passagem de mensagem e a outra em memória compartilhada, ou seja, na versão de *pthread* não era necessário se preocupar muito sobre como os dados seriam compartilhados pois todos compartilhavam da mesma memória e com isso uma simples variável global poderia ser o suficiente. O MPI por outro lado não garante que todos os processos compartilharam da mesma memória até porque com ele é possível dividir a mesma aplicação num *cluster* através de uma rede e isso tira fisicamente o acesso à memória compartilhada. Para poder compartilhar dados em uma aplicação utilizando MPI é preciso fazer com que os processos troquem mensagens entre si, esse processo é muito lento e geralmente a maior causa de *overhead* nesses tipos de aplicações e por isso não pode ser usado sem cuidado.

A biblioteca de MPI para a linguagem C oferece várias formas de passagem de mensagem e a que foi utilizada neste trabalho foi a função de *broadcast*. O MPI *broadcast* basicamente habilita em uma única linha de código um processo a enviar determinada informação a todos os outros processos concorrentes e a todos os processos fora o processo raiz do *broadcast* a receber essa informação, para isso é necessário somente que ambos os processo raiz e os processos concorrentes tenham acesso a linha do *broadcast* no código. Como o MPI *broadcast* espera os processos enviarem e receberem os dados não foi necessário uma ferramente de sincronismo como uma barreira por exemplo pois a forma com que o MPI faz essa comunicação dispensa essa necessidade, pelo menos para esse caso.

Mais especificamente sobre a implementação da função paralela, o laço paralelizado foi o laço referente as linhas da matriz garantindo assim que a matriz seja dividida entre os processos existentes e dentro dos laços é feita a atribuição do menor caminho encontrado nessa interação como normalmente é feito. A parte que cabe à passagem de mensagem do MPI ocorre na primeira linha do laço exterior onde há um *broadcast* da linha k pois essa vai ser a linha com qual o valor atual do caminho do grafo vai ser comparada afim de achar o menor caminho, teoricamente a coluna da matriz também seria necessária mas como os valores das linhas k-1 já estão atualizadas não foi necessário essa troca de mensagem. Ao final ocorre um *broadcast* da matriz inteira à fim de guardar o resultado final com os menores caminhos do grafo na matriz *graph*.

```
1  void fw(){
2      for(int k = 0; k < size; k++){
3          MPI_Bcast(graph + (k * size), size, MPI_FLOAT, k %
               nprocs, MPI_COMM_WORLD);
4          for(int i = rank; i < size; i += nprocs)
5              for(int j = 0; j < size; j++)
6                  graph[i*size + j] = MIN(graph[i * size + j],
                       graph[i * size + k] + graph[k * size + j]);
7      }
8      for(int i = 0; i < size; i++)
9          MPI_Bcast(graph + (i * size), size, MPI_FLOAT, i%nprocs,
               MPI_COMM_WORLD);
10 }
```

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 198 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, 9 configurações;
    1. Aplicação sequencial;
    2. Aplicação paralela de 1 até 8 processos;
- Para cada configuração, incluindo a aplicação sequencial, 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 processos disponíveis, a eficiência de cada configuração e o overhead de cada configuração, também será feito um estudo utilizando a métrica de Karp-Flatt a fim de identificar o motivo da eficiência alcançada pelos processadores.

5.1. Desempenho

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

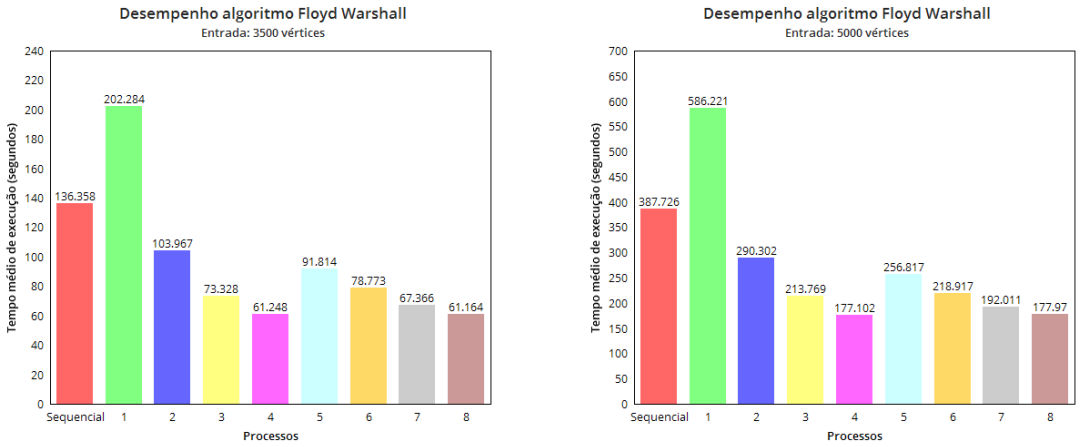

Figura 1. Tempo médio de execução para entradas de 3500 e 5000 vertices.

5.2. Speedup

Como previsto segundo a Lei de Amdahl, a medida que aumentamos a quantidade de processadores, houve um aumento de speedup também. Porém, ao utilizar mais que 4 processos, lembrando que o processador utilizado possui 4 núcleos e com hyper-thread possibilitando uso de 8 slots, o speedup decai quase 30% para ambos os tamanhos de entrada, como mostra a figura2.

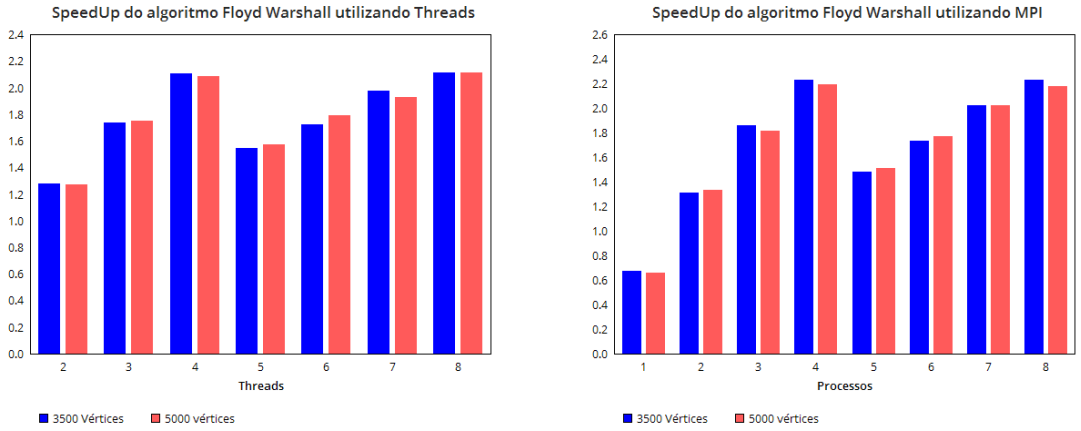

Figura 2. Gráficos de speedup para Threads e MPI.

Uma singularidade do MPI, é a necessidade de um processo gerenciador, que fica responsável pelo gerenciamento dos processos que estão trocando mensagem. Na utilização de 1 processo apenas para a aplicação, nota-se a importância desse processo gerenciador, para entradas maiores (como a de 5000 vértices) fica ainda mais evidente essa importância. Além de gerenciar a responsabilidade dos processos, este mesmo processo fica responsável por enviar e receber as mensagens para ele mesmo, causando um trabalho desnecessário e muito custoso, uma evidência que reforça essa teoria é o tempo gasto de execução com 1 processos mostrada na figura 1. Por isso, não obtivemos um aumento de 36% de speedup como obtido utilizando a biblioteca *pthreads* e sim de 94% de 1 para dois processos, indicando então a necessidade de pelo menos 2 processos para um speedup efetivo

O novo comportamento a partir de 5 processos que pode ter ocorrido basicamente por 2 motivos, aumento do overhead seja ele criado pela arquitetura ou pelo algoritmo (com a métrica de Karp-Flatt podemos confirmar essa hipótese), ou pela divisão de tarefas para cada processos quando excedido os limites físicos de recursos do processador, ou seja, pode acontecer de um núcleo ficar responsável por mais processos 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 processos utilizado.

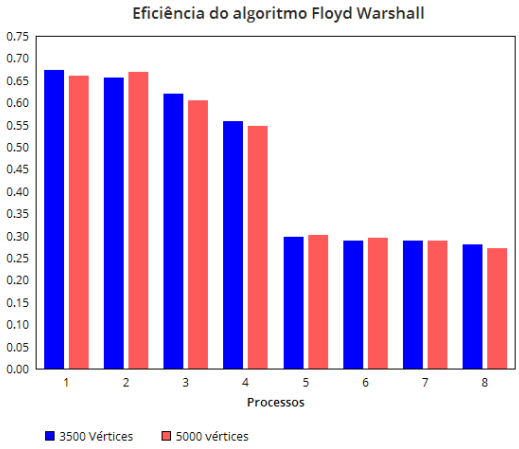

Figura 3. Gráfico de eficiência.

Com a criação de novos processos, é esperado também o aumento de overhead, uma vez que o overhead é diretamente proporcional ao número de processos criados; sendo assim a soma de todo o tempo gasto pelos processos paralelos, 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.

Esses overheads, podem ou não influenciar a eficiência dos processadores e também no speedup para p processadores.

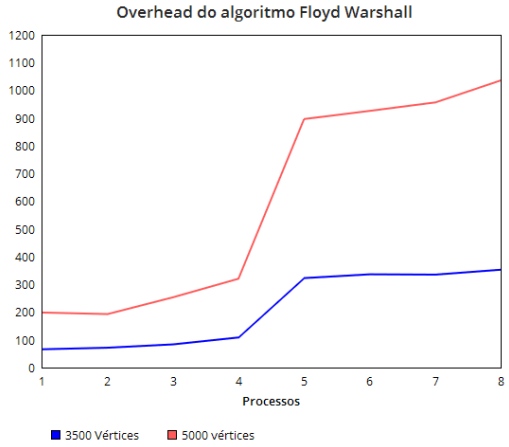

Figura 4. Gráfico do overhead.

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 processadores. 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 dos processos.

Para os casos em que a fração serial mantém-se contínua para qualquer quantidade de processos, indica-se que a eficiência caiu por motivos de limitações de oportunidade de paralelismo. Para quando a fração serial aumenta de acordo com o número de processos, as literaturas indicam que o problema de eficiência existe pelo aumento significativo de overheads, sejam eles de comunicação, sincronização, arquiteturais, implementação entre outros.

No nosso experimento, obtivemos um resultado, diferente do esperado de acordo com a literatura, como pode-se observar na figura 5, houve um decremento da fração serial enquanto aumentando a quantidade de processos nos núcleos físicos, e o mesmo comportamento para quando aumenta-se a quantidade de processos de 5 (pelo menos um núcleo responsável por mais de um processo) para a utilização de 8 processos. Vale ressaltar que não há o cálculo da fração serial para 1 processo, pois com essa configuração, um único processo é responsável por todo o processamento do algoritmo, sendo quase uma aplicação serial o que na fórmula, 1 processador acarreta em divisão por 0.

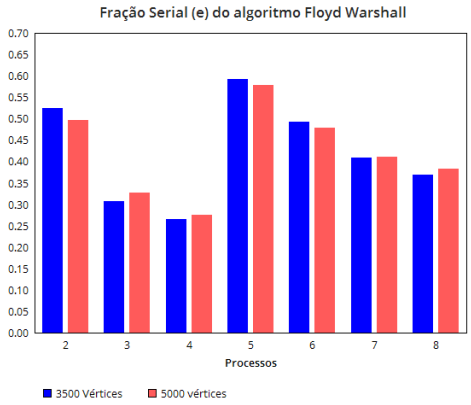

Figura 5. Gráfico da fração serial.

Este comportamento, pode indicar um paralelismo na aplicação quase que ideal, porém, sabe-se que aliado a custosa comunicação entre os processos, existem partes sequenciais na aplicação que não justificam tal resultado, 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, a partir de 5 processos, 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.

Vale ressaltar também a importância de que quando utilizarmos a biblioteca MPI, tem-se a necessidade de trabalhar com no mínimo 2 processadores, pois dessa forma um fica responsável pelo processo *gerenciador* e os demais processadores ficam responsáveis pelo processamento do algoritmo, envio e recebimento das mensagens. Caso contrário, se houver apenas um processador, este ficará responsável por gerenciar os processos e também executar a aplicação causando um trabalho desnecessário e muito custoso. Diferenciando então da utilização de *threads* para paralelizar uma aplicação, pois com uma thread, ela não ficará responsável pelo gerenciamento e processamento de si mesma, já utilizando a biblioteca *MPI*, esta sobrecarga de responsabilidade existe.

A comparação de desempenho e speedups entre o primeiro trabalho realizado (paralelização utilizando *pThreads*) e este desenvolvido, não pode ser feita diretamente, pois houve alterações na implementação do algoritmo que acarretou em otimizações. A melhora de desempenho utilizando MPI comparado a *pThreads*, não é uma realidade para as aplicações que possuem a mesma implementação, pois sabe-se que a comunicação entre os processos é muito custosa quando comparada a paralelização utilizando threads, esperando assim, uma queda de desempenho quando utiliza-se a biblioteca *MPI*.

E 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.

Foster, I. (1995). *Designing and building parallel programs*, volume 78. Addison Wesley Publishing Company Reading.

Gropp, W. D., Gropp, W., Lusk, E., Skjellum, A., and Lusk, A. D. F. E. E. (1999). *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press.

Ignácio, A. A. V. and Ferreira Filho, V. J. M. (2002). Mpi: uma ferramenta para implementação paralela. *Pesquisa Operacional*, 22(1):105–116.

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.

Pacheco, P. S. et al. (1998). A user's guide to mpi. *University of San Francisco*, 56.