UMA ABORDAGEM PARALELA DO ALGORITMO DE FLOYD PARA SOLUÇÃO DO PROBLEMA DO CAMINHO MÍNIMO

Rafael Castro de Souza
Programa de Pós-Graduação em Ciência da Computação - UERN/UFERSA
Av. Francisco Mota, 572 - Bairro Costa e Silva, Mossoró RN
rafaelcastro@ufersa.edu.br

Alexsandro Trindade Sales da Silva
Programa de Pós-Graduação em Ciência da Computação - UERN/UFERSA
Av. Francisco Mota, 572 - Bairro Costa e Silva, Mossoró RN
alextrindade.com.br@gmail.com

Wilamis Kleiton Nunes da Silva
Programa de Pós-Graduação em Ciência da Computação - UERN/UFERSA
Av. Francisco Mota, 572 - Bairro Costa e Silva, Mossoró RN
wilamiskleiton@yahoo.com.br

Rennê Stephany Ferreira dos Santos
Programa de Pós-Graduação em Ciência da Computação - UERN/UFERSA
Av. Francisco Mota, 572 - Bairro Costa e Silva, Mossoró RN
renne@ifpi.edu.br

Francisco Milton Mendes Neto
Programa de Pós-Graduação em Ciência da Computação - UERN/UFERSA
Av. Francisco Mota, 572 - Bairro Costa e Silva, Mossoró RN
miltonmendes@ufersa.edu.br

RESUMO

O algoritmo de Floyd foi concebido para resolver o problema do caminho mínimo entre todos os pares de vértices dado um grafo orientado. Todavia, a abordagem clássica desse algoritmo propõe um conjunto de passos a serem seguidos de forma sequencial, visto que na época em que foi proposto ainda não existia a possibilidade execução simultânea de múltiplas instruções. Tal possibilidade só se tornou possível com a inserção das arquiteturas paralelas de processadores nos computadores, que permitiu a execução de tarefas em um menor intervalo de tempo por meio do processamento paralelo de diversas instruções computacionais. Esse modelo de arquitetura contribuiu para o surgimento de um novo paradigma na programação de computadores, que diferentemente do modelo tradicional, onde as instruções são executadas todas de modo sequencial, este permite que o programador possa definir fluxos paralelos de instruções que serão executadas simultaneamente. Diante do cenário exposto, este trabalho apresenta uma variação do algoritmo convencional de Floyd utilizando uma abordagem paralela no fluxo de execução deste algoritmo, de modo que seus custos e complexidade computacional venham ser reduzidos em comparação com o algoritmo original.

PALAVRAS CHAVE. Algoritmo de Floyd, Computação Paralela, Problema do Caminho Mínimo.

Área Principal: Teoria e Algoritmos em Grafos

ABSTRACT

The Floyd algorithm was designed to solve the shortest path problem among all pairs of vertices given a directed graph. However, the classic approach of this algorithm proposes a set of steps to be followed in sequence because in the years of its creation did not exist the possibility of simultaneous execution of multiple instructions. This possibility was made possible with the inclusion of parallel processor architectures on computers, which allowed the execution of tasks in a shorter period of time by means of parallel processing of several computational instructions. This architecture model contributed to the emergence of a new paradigm in computer programming, that unlike of the traditional model, where all the instructions are executed in a sequential mode, this allows that the programmer can define parallel streams of instructions to be executed simultaneously. Given the above scenario, this paper presents a variation of Floyd conventional algorithm using a parallel approach in the execution flow of this algorithm, so that its cost and computational complexity will be reduced compared with the original algorithm.

KEYWORDS. Floyd algorithm. Parallel Computing. Shortest Path problem.

Main Area: Theory and Algorithms in Graphs.

1. Introdução

O Problema do Caminho Mínimo ou Caminho mais Curto (em inglês *Shortest Path Problem*) objetiva encontrar o caminho mínimo entre dois vértices dado um grafo orientado G. Esse é um tipo de problema que pode ser aplicado em muitas situações reais, a exemplo de problemas relacionados a tráfego aéreo e terrestre, roteamento de pacotes de redes, sistemas distribuídos, ou qualquer outro problema que possa ser representado por um grafo no qual as arestas sejam ponderadas e cujos valores sejam linearmente acumulados à medida que a rede é percorrida (Dehne, 2010).

Dentre os vários algoritmos utilizados na resolução de problemas do caminho mínimo, destaca-se o Algoritmo de Floyd, que basicamente recebe como entrada uma matriz de adjacência representando um grafo conexo e com pesos em suas arestas, e tem como saída o menor caminho entre todos os vértices de um determinado grafo. Seja (*x,y*) dois vértices quaisquer, o valor do caminho mínimo dentre estes pode passar também por outros vértices, sendo que, para se chegar a *y* vindo de *x* com um custo mínimo, primeiramente deve-se ir de *x* aos vértices intermediários, que são aqueles que se encontram entre *x* e *y*. Como visto em (Cormen,2009), sua ordem de complexidade é $O(n^3)$.

O presente trabalho propõe uma variação deste algoritmo por meio da paralelização de fluxos de instruções que serão executados de forma simultânea a fim de obter ganhos computacionais em comparação com sua abordagem clássica proposta por Floyd. A abordagem utilizada neste trabalho se beneficia da decomposição do problema em subproblemas nos quais podem ser seguramente executados simultaneamente. Para que isso possa ser feito são necessárias técnicas que irão dividir fluxos de execução do código-fonte, nas quais irão possibilitar a estruturação dos subproblemas de tal forma que sejam executados em máquinas com arquiteturas paralelas (Mattson, 2004).

Vale ainda ressaltar que a paralelização de um algoritmo originalmente concebido para ser executado de forma sequencial não é uma tarefa trivial, visto que não são todos os trechos de um algoritmo sequencial em que cabe o paralelismo, e, se isto for feito sem um estudo sistemático do funcionamento do algoritmo, a paralelização pode resultar em perca de performance e também na saída de resultados conflitantes.

Este artigo está organizado da seguinte forma: a Seção 2 apresenta a fundamentação teórica sobre a computação paralela, a Seção 3 explana sobre o algoritmo de Floyd, a Seção 4 apresenta os trabalhos relacionados, a Seção 5 descreve as implementações da aplicação, bem como os resultados alcançados, e, por fim, a Seção 6 traz as considerações finais do trabalho e as perspectivas de trabalhos futuros.

2. Computação Paralela

A introdução dos circuitos integrados no *hardware* ampliou as possibilidades no processamento de aplicações pelos computadores, que consequentemente permitiu que tarefas até então consideradas não realizáveis fossem possíveis de serem efetuadas. Segundo Tanenbaum (2003), o principal artifício para o aumento da capacidade de processamento dos computadores tem sido o incremento de transistores e o aumento na frequência das operações por segundo. No entanto, esse esquema de aumentar o desempenho do *hardware* encontrou algumas limitações, tais como: superaquecimento dos chips, alto consumo de energia, e até mesmo o limite físico de fabricar resistores menores do que os atuais (Cardozo e Santiago, 2013).

Diante disso, os projetistas de *hardware* começaram então a estudar meios alternativos para aumentar o desempenho da CPU. Um dos métodos adotados para conceber esse aumento computacional foi a inserção de múltiplos processadores em um único chip. A ideia por trás desse método é que, no lugar de ser produzido um único processador com alto poder computacional que irá executar todo o fluxo de instruções, múltiplos processadores com um menor poder computacional irão dividir o fluxo de instrução a ser executado, de modo que o desempenho final obtido por meio da divisão de tarefas supere o tempo de processamento com apenas um único processador.

Com essa nova abordagem da utilização de múltiplos processadores por CPU, surgiu então uma nova área na ciência da computação denominada de computação paralela, que tem por objetivo o estudo da divisão das instruções computacionais a serem executadas entre várias unidades de processamento em um mesmo intervalo de tempo (Silva, 2005).

Segundo Flynn (1972), as arquiteturas paralelas podem ser categorizadas baseadas no modo execução de uma sequência de instruções sobre uma sequência de dados. Apesar desse modelo de classificação de arquiteturas paralelas ter sido proposto em 1972, ele ainda é válido, muito difundido e bastante referenciado na literatura (Rose e Navaux, 2008). Os quatros tipos de arquiteturas paralelas propostas por Flynn são:

1. *Single Instruction Single Data* (SISD): Quando um único fluxo de instruções atua sobre um único fluxo de dados;

2. *Single Instruction Multiple Data* (SIMD): Quando uma única instrução é executada ao mesmo tempo sobre múltiplos dados;

3. *Multiple Instruction Single Data* (MISD): Quando múltiplos fluxos de instruções atuam sobre um único fluxo de dados;

4. *Multiple Instruction Multiple Data* (MIMD): Cada unidade de processadores recebe um fluxo próprio de instruções e de dados.

A Figura a seguir apresenta de forma simplificada cada uma dessas arquiteturas, onde C representa uma unidade de controle, P uma unidade de processamento e M a memória principal.

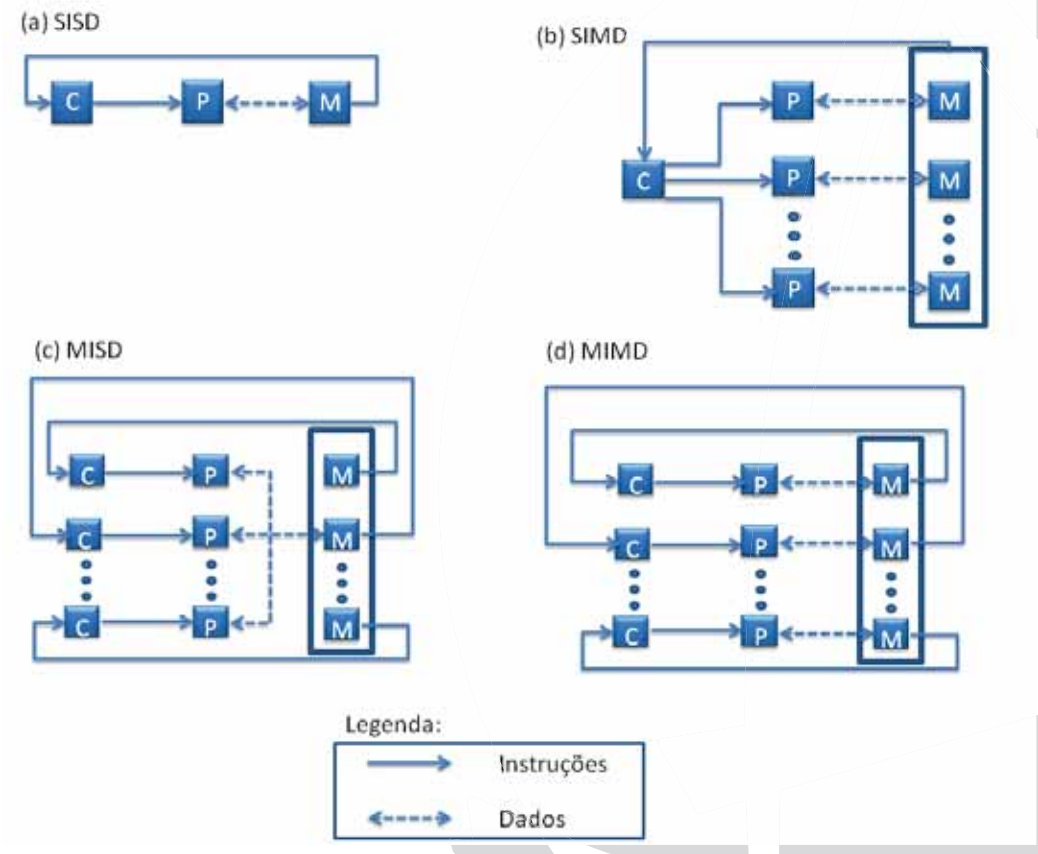

Figura 1: Arquitetura de Flynn (Rose e Naxau, 2008). Figura adaptada pelo autor.

Embora as arquiteturas paralelas possam melhorar significativamente no que se refere ao tempo de processamento das instruções, sua dificuldade de utilização está atrelada à forma como o *software* é desenvolvido. Pois o objetivo da programação paralela não está apenas limitada à escrita de programa corretos, eficientes e portáveis, mas também na contemplação de programas escaláveis de acordo com o número de núcleos por processador (Nascimento, 2011). Esta não é uma tarefa trivial de ser realizada, visto que ambientes paralelos e distribuídos devem contemplar uma série requisitos, tais como: gerenciamento de concorrência, balanceamento de carga, consistência de memória, granularidade da decomposição e sincronia de fluxos de execução (Asanovic et. al., 2009).

3. Algoritmo de Floyd

O algoritmo de Floyd foi concebido para solucionar o problema do caminho mínimo entre vértices dado um grafo orientado G. Este algoritmo recebe basicamente como entrada uma matriz de adjacência de um grafo orientado e multivalorado. Em seguida, o algoritmo calcula para cada par de vértices o menor caminho entre estes. A complexidade deste algoritmo é da ordem de $O(n^3)$, onde *n* é a quantidade de vértices (Cormen, 2009).

Além de encontrar o peso do caminho mais curto, o algoritmo de Floyd também calcula a rota do caminho mais curto. A fim de facilitar o entendimento, iremos seguir um exemplo de como o algoritmo funciona. Considere o grafo G ilustrado na Figura 2:

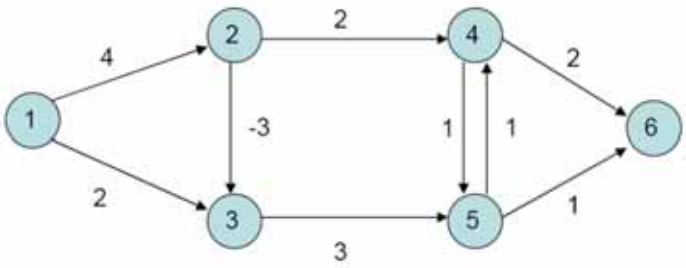

Figura 2: Grafo G.

Após definido o grafo, deve-se construir a matriz de adjacência (MA) e a matriz de roteamento (R). Segundo Boaventura Neto e Jurkiewicz (2009) deve-se utilizar os seguintes critérios para atribuição dos valores:

1. Valor Infinito: Quando as posições dos arcos não existirem;
2. Valor Nulo: Nas posições que se encontram na diagonal principal;
3. Peso das arestas: Que correspondem ao valor correspondente dos arcos que existem.

Já para a construção da matriz de roteamento, todos os elementos de uma coluna devem ter o valor de seus índices, com exceção dos elementos que correspondem ao valor infinito da matriz de adjacência, que neste caso, receberão o valor zero. A Figura 3 apresenta a matriz de adjacência, e a matriz de roteamento correspondente para o grafo G ilustrado na Figura 2.

Calculada a matriz de adjacência e a matriz de roteamento, o algoritmo irá verificar se existem caminhos mais curtos do que os que estão nos arcos utilizando cada vértice como sendo um intermediário. Dado *MA[n][n]* e *R[n][n]* sendo a matriz de adjacência e a matriz de roteamento respectivamente, essa verificação é feita por meio da análise da seguinte condição: Se *MA[i][k] + MA[k][j] < MA[i][j]*, onde *k* é o vértice utilizado como intermediário, *i* é o índice da linha e *j* como sendo o índice da coluna. Essa verificação deve ser feita para todos os elementos da *MA*, utilizando os vértices de forma ordenada como intermediários, ou seja, a análise deve começar a

Matriz de Adjacência

0	4	2	∞	∞	∞
∞	0	-3	2	∞	∞
∞	∞	0	∞	3	∞
∞	∞	∞	0	1	2
∞	∞	∞	1	0	1
∞	∞	∞	∞	∞	0

Matriz de Roteamento

0	2	3	∞	∞	∞
∞	0	3	4	∞	∞
∞	∞	0	∞	5	∞
∞	∞	∞	0	5	6
∞	∞	∞	4	0	6
∞	∞	∞	∞	∞	0

Figura 3: Matriz de Adjacência e Matriz de Roteamento correspondente do grafo ilustrado na Figura 2.

partir do vértice de número um e prosseguir para os vértices sucessores. Caso a condição *MA[i][k] + MA[k][j] < MA[i][j]* seja verdadeira, o elemento na posição *MA[i][j]* recebe o valor *MA[i][k] + MA[k][j]* e o elemento *R[i][j]* recebe o valor de *R[k][j]*. Caso contrário, as matrizes de adjacência e de roteamento não sofrerão alterações.

A Figura 4 apresenta as linhas e as colunas bases que são utilizadas para cada vértice intermediário no algoritmo de Floyd para o grafo ilustrado na Figura 2.

(a) Utilizando o vértice 1 como intermediário

0	2	3	∞	∞	∞
∞	0	3	4	∞	∞
∞	∞	0	∞	5	∞
∞	∞	∞	0	5	6
∞	∞	∞	4	0	6
∞	∞	∞	∞	∞	0

(b) Utilizando o vértice 2 como intermediário

0	2	3	∞	∞	∞
∞	0	3	4	∞	∞
∞	∞	0	∞	5	∞
∞	∞	∞	0	5	6
∞	∞	∞	4	0	6
∞	∞	∞	∞	∞	0

(c) Utilizando o vértice 3 como intermediário

0	2	3	∞	∞	∞
∞	0	3	4	∞	∞
∞	∞	0	∞	5	∞
∞	∞	∞	0	5	6
∞	∞	∞	4	0	6
∞	∞	∞	∞	∞	0

(d) Utilizando o vértice 4 como intermediário

0	2	3	∞	∞	∞
∞	0	3	4	∞	∞
∞	∞	0	∞	5	∞
∞	∞	∞	0	5	6
∞	∞	∞	4	0	6
∞	∞	∞	∞	∞	0

(e) Utilizando o vértice 5 como intermediário

0	2	3	∞	∞	∞
∞	0	3	4	∞	∞
∞	∞	0	∞	5	∞
∞	∞	∞	0	5	6
∞	∞	∞	4	0	6
∞	∞	∞	∞	∞	0

(f) Utilizando o vértice 6 como intermediário

0	2	3	∞	∞	∞
∞	0	3	4	∞	∞
∞	∞	0	∞	5	∞
∞	∞	∞	0	5	6
∞	∞	∞	4	0	6
∞	∞	∞	∞	∞	0

Figura 4: Linhas e Colunas bases de cada vértice intermediário da Matriz de Adjacência ilustrada na Figura 2.

Ao final da verificação, a saída do algoritmo será a matriz de adjacência contendo o menor peso para um caminho entre dois vértices, e a matriz de roteamento indicando qual o menor caminho a ser percorrido. A seguir é apresentada a formalização do pseudocódigo do algoritmo de Floyd.

Algorithm 1 Algoritmo de Floyd

1: **Entrada:** Matriz de Adjacência $MA_{nxn}(G)$; Matriz de Roteamento R_{nxn}; *n* como sendo o número de vértices.
2: **função** FLOYD ALGORITHM
3: **para** k $\leftarrow$ 1 **até** n **faça**
4: **para** i $\leftarrow$ 1 **até** n **faça**
5: **para** j $\leftarrow$ 1 **até** n **faça**
6: **se** $MA[i][k] + MA[k][j] < MA[i][j]$] **então**
$MA[i][j] \leftarrow MA[i][k] + MA[k][j]$
$R[i][j] \leftarrow R[k][j]$
7: **fim se**
8: **fim para**
9: **fim para**
10: **fim para**
11: **fim função**

4. Trabalhos Relacionados

A computação paralela vem oferecendo contribuições significativas na otimização computacional de várias áreas de pesquisa na ciência da computação. Garcia et al. (2013) apresentam um estudo sobre a paralelização do algoritmo de ordenação *MergeSort* utilizando a linguagem de programação JAVA. A paralelização foi feita por meio do uso de *Threads*, que a linguagem implementa de forma nativa. Os resultados apresentaram uma redução de até 74% do tempo de execução do algoritmo paralelo em comparação com sua abordagem sequencial.

Já Comati et. al. (2013) apresentam uma paralelização do algoritmo Zhang-Suen, que é um algoritmo de esqueletização de imagens que vem sendo aplicado no processamento digital de imagens médicas para identificação da ramificação dos vasos sanguíneos nos olhos. Essa paralelização foi por meio da arquitetura CUDA, que permite uma divisão das instruções que serão executadas pelo processador com a placa de vídeo. Com a paralelização proposta foi obtido um desempenho de até 45 vezes mais rápido na execução do algoritmo em comparação com sua abordagem sequencial.

Dantas e Cárceres (2014) propuseram uma implementação paralela para o problema da mochila multidimensional utilizando CUDA, MPI e redes neurais aumentadas para o modelo de memória compartilhada e memória distribuída. O problema da mochila multidimensional consiste em, dado um conjunto de *n* diferentes itens, cada qual com um valor associado, e *m* recursos, decidir quais dos itens devem ser colocados na mochila visando maximizar o valor sem exceder a capacidade desta. Os resultados da implementação utilizando CUDA e redes neurais aumentadas se mostraram bastante eficientes em relação ao algoritmo sequencial, o que não pôde ser visto para a implementação utilizando o MPI.

O fator que difere este trabalho dos demais é que o algoritmo proposto a ser paralelizado é o algoritmo de Floyd utilizando o paradigma da memória compartilhada por meio do uso da biblioteca *OpenMP*.

5. Resultados Computacionais

Para que fosse possível realizar uma paralelização do algoritmo de Floyd, foi feito um estudo bem detalhado do funcionamento deste algoritmo, pois a paralelização de trechos sequenciais que dependem do resultado de uma iteração anterior, na maioria das vezes resulta em erros na saída final do algoritmo.

Conforme explicado na Seção 3, o algoritmo de Floyd contém três laços de repetição *for* aninhados. O primeiro laço *for*, onde temos { *k=1, ..., até n* }, é a estrutura de repetição responsável pela definição do vértice que será utilizado como base para realização do cálculo. Como o cálculo

do vértice subsequente depende da iteração de seu vértice antecessor, em outras palavras, a iteração para *k+1* depende do resultado calculado em *k*, a paralelização deste laço se torna inviável.

No entanto, a paralelização pode ser aplicada nos dois laços de repetição mais internos do algoritmo, onde temos { *i=1, ..., até n* } e { *j=1, ..., até n* } respectivamente, sendo o *i* representante da linha e o *j* da coluna que serão utilizados para o cálculo. Como o cálculo somente será realizado para um dado elemento da matriz de adjacência e da matriz de roteamento por vez, e como o cálculo do elemento na posição *P=(i,j)*, para a iteração *k*, independe do cálculo das posições das outras linhas e colunas para esta mesma iteração, pode-se então aplicar a paralelização destes dois laços mais internos do algoritmo sem prejudicar a saída final do algoritmo.

A Figura 5 ilustra de forma gráfica a paralelização destes dois laços. Os quadros que estão em azul são os elementos do vértice que corresponde à iteração *k*, e { $P_1, P_2, P_3, ..., P_n$ } correspondem aos processadores. O item (a) expõe a forma em que o algoritmo sequencial executa cada elemento, e o item (b) apresenta essa mesma execução sendo realizado por *n* processadores.

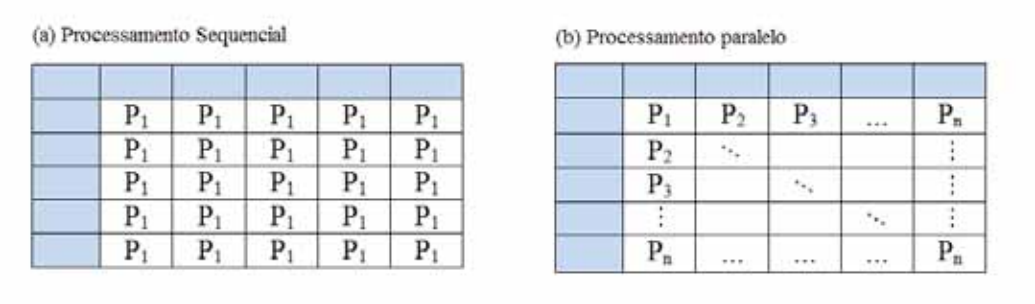

Figura 5: Exemplo da divisão dos elementos da matriz de adjacência a serem executados pelo processador. (a) Divisão utilizando a abordagem sequencial, onde somente um único processador realiza todo o processamento. (b) Divisão utilizando a abordagem paralela, onde *n* processadores realizam esta tarefa.

A fim de validar a abordagem proposta, foi implementado o algoritmo de Floyd de forma sequencial e paralela. Para sua implementação paralela foi utilizada a biblioteca *OpenMP*, que é uma API multiplataforma de memória compartilhada para programação paralela em C/C++ e Fortran (OpenMP, 2015). Para a realização do estudo de caso, foram utilizadas instâncias retiradas da TSPLIB, que é uma biblioteca com exemplos de instâncias do problema do caixeiro viajante (TSPLIB, 2015).

A implementação sequencial e paralela do algoritmo de Floyd foi executada em uma máquina com 4 GB de memória RAM, sistema operacional Linux Ubuntu, processador Intel Core i5-2400 (3.1 GHz). Os experimentos foram realizados sobre cinco instâncias que correspondem respectivamente a: 100 cidades, 200 cidades, 299 cidades, 439 cidades e 666 cidades. Portanto, se uma instância tem 100 cidades, esta contém 100 vértices e sua matriz de adjacência comporta 100 x 100 elementos, visto que a matriz de adjacência é definida com sendo MA_{nxn}, onde *n* é o número de vértices.

A Tabela 1 apresenta os resultados do desempenho computacional da abordagem paralela e sequencial para este algoritmo. Para cada instância o algoritmo foi executado cinco vezes, e foram computados sua média aritmética, mediana, variância, desvio padrão e *speedup*.

A Equação 1 exibe o cálculo da média aritmética do tempo de execução das cinco iterações, onde *T* é o tempo de execução, *i* é a iteração e *n* é o número de iterações. A média aritmética é utilizada para que se possa determinar um ponto central no tempo de execução do programa, pois, devido ao escalonamento de processos do sistema operacional, esse tempo de execução tende a variar alguns milisegundos.

$$\frac{\sum_{i=1}^{n} T(i)}{n} \tag{1}$$

A Equação 2 ilustra o cálculo da mediana para um número ímpar de elementos em uma dada amostra, onde *n* é o número de elementos de uma determinada amostra. A mediana é utilizada para identificar o elemento central de uma determinada amostra de dados.

$$\frac{n+1}{2} \tag{2}$$

A Equação 3 demonstra o cômputo da variância, onde *x* é o valor do tempo de processamento da iteração, $\bar{x}$ é a média do tempo de processamento de todas as iterações, e *n* é o número de iterações. A variância é utilizada para verificar se existe uma discrepância expressiva entre as amostras.

$$\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1} \tag{3}$$

A Equação 4 apresenta a equação do desvio padrão, onde *x* é o valor do tempo de processamento da iteração, $\bar{x}$ é a média do tempo de processamento de todas as iterações, e *n* é o número de iterações. O desvio padrão é utilizado para que se possa verificar qual o nível de variação ou dispersão existente em relação à média ou de um determinado valor esperado, neste caso, o nível de variação será calculado em relação à média do tempo de execução.

$$\sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}} \tag{4}$$

A Equação 5 expõe a fórmula para o cálculo do *speedup*, que consiste na divisão do tempo de execução sequencial dividido pelo tempo de execução paralelo . O *speedup* é uma métrica utilizada para saber o quanto um algoritmo paralelo é mais rápido em comparação com sua implementação sequencial correspondente (Coelho, 2013). Na equação abaixo, *T(s)* representa o tempo de execução sequencial e T(p) é o tempo de execução paralelo.

$$\frac{T(s)}{T(p)} \tag{5}$$

Tabela 1: Comparação do desempenho computacional utilizando a abordagem sequencial e paralela do algoritmo de Floyd para instâncias com: 100, 200, 299, 439 e 666 cidades.

	Média	Mediana	Variância	Desvio Padrão	*speedup*
100 Cidades					
Sequencial	0,007473206	0,0076563	1,76464 $.10^{-07}$	0,000420076	
Paralelo	0,002255312	0,0024472	3,62267 $.10^{-07}$	0,000601886	3,313601843
200 Cidades					
Sequencial	0,03156458	0,032291	1,28084 $.10^{-06}$	0,001131741	
Paralelo	0,008487732	0,00838465	1,42692 $.10^{-07}$	0,000377746	3,718847391
299 Cidades					
Sequencial	0,09091868	0,0927604	8,96029 $.10^{-06}$	0,002993375	
Paralelo	0,02775238	0,0282711	1,13132 $.10^{-06}$	0,001063635	3,276067854
439 Cidades					
Sequencial	0,26931432	0,268244	1,15566 $.10^{-05}$	0,003399495	
Paralelo	0,08445788	0,0846013	4,8952 $.10^{-07}$	0,000699657	3,188741181
666 Cidades					
Sequencial	0,9406126	0,941021	7,91927 $.10^{-06}$	0,002814119	
Paralelo	0,2905618	0,290701	1,57623 $.10^{-07}$	0,000397017	3,237220447

A Figura 6 apresenta de forma gráfica o resultado da média de execução da abordagem sequencial em comparação com a paralela. Podemos perceber que a medida em que o número de cidades aumenta, o algoritmo sequencial demanda bem mais tempo para solucionar o problema do caminho mínimo. Enquanto que, na abordagem paralela deste algoritmo, é perceptível que o mesmo apresentou um tempo de execução de pelo menos 3 vezes mais rápido em comparação com a abordagem sequencial.

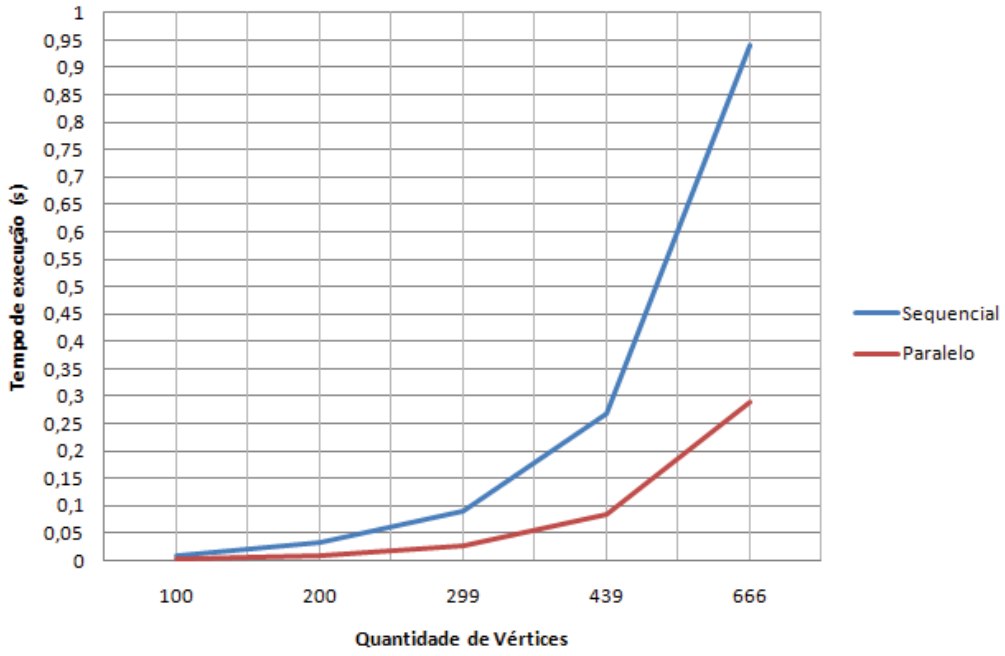

Figura 6: Comparação do tempo de execução do algoritmo de Floyd sequencial em relação com sua abordagem paralela.

Vale ressaltar ainda que, na implementação paralela, o fluxo de execução a ser divido foi definido para uma arquitetura paralela com 4 processadores devido ao número de *cores* da máquina em que o algoritmo foi testado. Sendo assim, pressupõe-se que se o mesmo fosse executado em uma máquina com um maior número de processadores, como por exemplo em um *cluster*, o ganho computacional poderia ter sido ainda maior.

6. Considerações Finais

Foi apresentado neste artigo a implementação e testes do algoritmo de Floyd utilizando o paradigma da computação paralela para máquinas com memória compartilhada. A eficácia dessa abordagem pôde ser percebida pela divisão do processamento para um número de p processadores. Com os resultados obtidos foi possível perceber que a abordagem proposta neste trabalho é pelo menos 3 vezes mais eficiente que a versão sequencial do algoritmo.

Vale ressaltar que, com a otimização no tempo de solução do caminho mínimo apresentado por este trabalho, sua abordagem poderia ser utilizada principalmente por setores que trabalham na busca e no redirecionamento de rotas em tempo real, como por exemplo no tráfego aéreo, cuja velocidade na transmissão das informações de rotas e o estabelecimento de caminhos alternativos tendem a interferir na segurança dos voos.

Como trabalhos futuros, pretende-se realizar novos estudos para a paralelização do algoritmo de Floyd utilizando outras arquiteturas, como por exemplo, a da memória distribuída. E também um estudo em relação à viabilidade da paralelização de outros algoritmos que propõem a solução para o problema do caminho mínimo a fim de otimizar os já existentes.

Agradecimentos

Os autores agradecem a agência brasileira de pesquisa CAPES pelo apoio financeiro concedido a este trabalho, a Universidade Federal Rural do Semi-Árido (UFERSA) e a Universidade do Estado do Rio Grande do Norte (UERN) pelo apoio e infraestrutura disponibilizada

Referências Bibliográficas

Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K., (2009), A View of the Parallel Computing Landscape. Commun. ACM, ACM, New York, NY, USA, v. 52, p. 56-67.

Boaventura Neto, P. O., Jurkiewicz, S., (2009), Grafos: Introdução e prática. Editora Blucher.

Cardoso, E. A., Santiago, R., (2013), Análise Comparativa de Algoritmos NP-Completo Executados em CPU E GPU Utilizando CUDA. *Computer on the Beach 2013 - Artigos Completos*, 79-87.

Coelho, S. A., (2013), Introdução a Computação Paralela com o OpenMPI. In: Simpósio Mineiro de Computação, Escola Regional de Informática de Minas Gerais, p. 24-44.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C., (2009), Introduction to algorithms. MIT press.

Dantas, B. A., Cáceres E. N., (2014), A Parallel Implementation to the Multidimensional Knapsack Problem Using Augmented Neural Networks, *2014 XL Latin American Computing Conference (CLEI)*, p. 1-9.

Dehne, F.,Y Ogaratnam, K., (2010), Exploring the limits of gpus with parallel graph algorithms. CoRR, abs/1002.4482.

Flynn, M. J., (1972), Some Computer Organizations and their Effectiveness, *IEEE TOC 21*, pp. 948- 960.

Garcia, A. M., Cera, M. C., Mergen, S. L. S., (2013), Analisando o Desempenho da Paralelização no Algoritmo de Ordenação *Mergesort In-place*, *Anais do ERAD-RS*, ISSN 2177-0085.

Komati, K. S., Souza, F. S. L., Guimarães, J. A., Andrade, Andrade, J. O., (2013), PARALELIZAÇÃO DE ESQUELETIZAÇÃO DE IMAGENS DE FUNDO DE RETINA NA ARQUITETURA CUDA, *Revista Eletrônica do Alto Vale do Itajaí*, V. 2, N. 1, 75-85.

Mattson, T. G., Sanders, B. A., Massingill, B. L., (2004), A pattern language for parallel programming.

Nascimento, J. P. B., (2011), Um Algoritmo Paralelo para Cálculo de Centralidade em Grafos Grandes, Dissertação (Mestrado) - Mestrado em Modelagem Matemática e Computacional, Centro Federal de Educação Tecnológica de Minas Gerais.

OpenMP, Disponível em: <http://openmp.org/wp/ > Acesso em: 02 de abril de 2015.

Rose, César A. F., Navaux, Philippe O. A., (2008), Arquiteturas Paralelas, Editora Sagra-Luzzatto, 1ª Edição, 152 p.

Silva, A. J. M., 2005, Implementação De Um Algoritmo Genético Utilizando O Modelo De Ilhas, Dissertação (Mestrado) - Curso de Engenharia Civil, Departamento de Engenharia Civil, Universidade Federal do Rio de Janeiro.

Tanenbaum, A. S., (2003), Sistemas operacionais modernos, *Pearson Prentice Hall, 2ª edição*, ISBN: 8587918575.

TSPLIB Disponível em: <http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95z > Acesso em: 02 de abril de 2015.